

Testing Framework Migration with Large Language Models

Altino Alves

DCC, UFMG

Belo Horizonte, Brazil

altinojunior@dcc.ufmg.br

João Eduardo Montandon

DCC, UFMG

Belo Horizonte, Brazil

joao@dcc.ufmg.br

Andre Hora

DCC, UFMG

Belo Horizonte, Brazil

andrehora@dcc.ufmg.br

Abstract

Python developers rely on two major testing frameworks: unittest and Pytest. While Pytest offers simpler assertions, reusable fixtures, and better interoperability, migrating existing suites from unittest remains a manual and time-consuming process. Automating this migration could substantially reduce effort and accelerate test modernization. In this paper, we investigate the capability of Large Language Models (LLMs) to automate test framework migrations from unittest to Pytest. We evaluate GPT 4o and Claude Sonnet 4 under three prompting strategies (Zero-shot, One-shot, and Chain-of-Thought) and two temperature settings (0.0 and 1.0). To support this analysis, we first introduce a curated dataset of real-world migrations extracted from the top 100 Python open-source projects. Next, we actually execute the LLM-generated test migrations in their respective test suites. Overall, we find that 51.5% of the LLM-generated test migrations failed, while 48.5% passed. The results suggest that LLMs can accelerate test migration, but there are often caveats. For example, Claude Sonnet 4 exhibited more conservative migrations (e.g., preserving class-based tests and legacy unittest references), while GPT-4o favored more transformations (e.g., to function-based tests). We conclude by discussing multiple implications for practitioners and researchers.

CCS Concepts

- Software and its engineering → Software testing and debugging.

Keywords

Software Testing, Test Migration, Large Language Models, Python, Unittest, Pytest

ACM Reference Format:

Altino Alves, João Eduardo Montandon, and Andre Hora. 2026. Testing Framework Migration with Large Language Models. In *7th ACM/IEEE International Conference on Automation of Software Test (AST 2026) (AST '26), April 13–14, 2026, Rio de Janeiro, Brazil*. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3793654.3793749>

1 Introduction

Software testing is fundamental to avoiding regressions and catching bugs. Currently, Python developers can rely on two main testing frameworks: unittest [55] and Pytest [41]. Pytest provides some advantages compared to unittest, including simpler assertions, reuse of fixtures, and interoperability [4, 41].



This work is licensed under a Creative Commons Attribution 4.0 International License.
AST '26, Rio de Janeiro, Brazil

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2476-3/2026/04

<https://doi.org/10.1145/3793654.3793749>

```
unittest (before migration)
9 - class BufferTest(unittest.TestCase):
10 -     def setUp(self):
11 -         b = Buffer()
12 -         eb = EditorBuffer('b1', b)
13 -         self.window = Window(eb)
14 -         self.tabpage = TabPage(self.window)

16 -     def test_initial(self):
17 -         self.assertIsInstance(self.tabpage.root, VSplit)
18 -         self.assertEqual(self.tabpage.root, [self.window])

28 -     def test_vsplit(self):
29 -         # Create new buffer.
30 -         b = Buffer()
31 -         eb = EditorBuffer('b1', b)
32 -         # Insert in tab, by splitting.
33 -         self.tabpage.vsplit(eb)
34 -         self.assertIsInstance(self.tabpage.root, VSplit)
35 -         self.assertEqual(len(self.tabpage.root), 2)

pytest (after migration)
9 + @pytest.fixture
10 + def prompt_buffer():
11 +     return Buffer()
12 +
13 +
14 + @pytest.fixture
15 + def editor_buffer(prompt_buffer):
16 +     return EditorBuffer(prompt_buffer, 'b1')
17 +
18 +
19 + @pytest.fixture
20 + def window(editor_buffer):
21 +     return Window(editor_buffer)
22 +
23 +
24 + @pytest.fixture
25 + def tab_page(window):
26 +     return TabPage(window)

7 + def test_initial(window, tab_page):
8 +     assert isinstance(tab_page.root, VSplit)
9 +     assert tab_page.root == [window]

12 + def test_vsplit(tab_page):
13 +     # Create new buffer.
14 +     b = Buffer()
15 +     eb = EditorBuffer(b, 'b1')
16 +
17 +     # Insert in tab, by splitting.
18 +     tab_page.vsplit(eb)
19 +
20 +     assert isinstance(tab_page.root, VSplit)
21 +     assert len(tab_page.root) == 2
```

Figure 1: Migration from unittest to Pytest (pyvim).

Multiple Python projects—such as Pandas, NumPy, ScikitLearn, Requests, and Flask—have migrated to or are currently migrating to Pytest due to these benefits [4]. For example, Figure 1 presents a migration from unittest to Pytest in pyvim.¹ This migration consists of three major modifications. The first chunk shows that the unittest setUp method is being split into four @Pytest.fixtures properties. The second and third chunks modify test methods to receive these fixtures and use assert statements during the test verification. Section 2 explains this migration in more detail.

Pytest allows projects to run tests written in unittest, thus enabling a gradual migration process. However, this migration can be time-consuming given the complexity of test suites: a recent study found that some projects may take months or years to conclude the migration or simply never conclude it [4]. Besides, using two different testing frameworks at the same time can increase the effort needed to maintain the test suite. Such projects could benefit from an automated solution to assist developers in migrating these tests faster and effectively.

Recently, Large Language Models (LLMs) have been evaluated in multiple software engineering tasks, including source code and tests generation, bug fix, code smells detection, and to support the code review process [16, 20, 22, 24, 27, 36, 38, 40, 52, 54]. Other studies investigated the use of LLMs to support API migration [1, 29]. However, to the best of our knowledge, no study has explored the migration of the test frameworks.

¹<https://github.com/prompt-toolkit/pyvim/commit/7e1c7bfb505cefba468>

In this paper, we investigate the capability of LLMs to assist developers during the migration of test frameworks. For this purpose, we built a dataset with over 900 real-world migrations from `unittest` to `Pytest`, manually implemented by developers [3]. Next, we selected a subset of migrated tests that still exist by September 2025, and thus remain executable. This process resulted in a final set of 40 isolated migrations that are investigated in this paper. In our experiment, we rely on GPT-4o and Claude Sonnet 4 to migrate Python tests from `unittest` to `Pytest`. We applied three prompting strategies—Zero-shot, One-shot, and Chain-of-Thought, under two temperature settings (0.0 and 1.0), generating a total of 480 migration variants.

To ensure the migration proposed by the LLMs was correct, we reproduced the test environment of each migration and actually *executed* both LLM-migrated and developer-migrated tests. We then compared the results of both executions in terms of test correctness (i.e., test was successfully executed) and test coverage (i.e., test coverage remained the same). Both models achieved an overall correctness rate of 48.54%, while successful migrations maintained the same test coverage, indicating that LLMs did not modify runtime behaviour. However, 25 out of 40 migrations presented at least one failing configuration (247 total failures), mostly related to dependency handling, fixture adaptation, or setup inconsistencies. While Claude Sonnet 4 preserved class-based structures and `unittest` patterns, GPT-4o favored function-based rewrites aligned with `Pytest`, revealing distinct yet complementary migration styles. We conclude by discussing implications for researchers and practitioners.

Contribution. The contributions of this paper are twofold: (1) we provide the first study to explore test migration with LLMs and (2) we discuss multiple implications for researchers and practitioners.

2 Background and Motivation

`Unittest` [55] and `Pytest` [41] are the most popular testing frameworks in Python [30]. `Unittest` belongs to the Python standard library, while `Pytest` is a third-party testing framework. `Unittest` relies on classes and inheritance to create tests (i.e., the test class needs to extend the `unittest` class `TestCase`), whereas `Pytest` tests can be regular functions, with the `test` prefix. Consequently, `Pytest` tests tend to be less verbose than `unittest` ones. Another difference is the assertions: `unittest` provides `self.assert*` methods (e.g., `assertEqual`, `assertTrue`, etc.), while `Pytest` allows developers to use the regular Python `assert` statement for verifying expectations and values. There are many other differences; for example, `Pytest` facilitates the creation of parameterized tests and the reuse of fixtures.

Due to the advantages of `Pytest`, many Python projects have migrated to this framework. A prior study discovered that 27% of top-100 most popular Python projects migrated or were migrating to `Pytest` [4]. To gain more insights into the relevance of this problem, we replicated this study in the same set of popular projects, found that the migration rate has increased to 37%.

Migrating from `unittest` to `Pytest` may involve at least the following major steps: (1) removing test from class and moving to regular functions; (2) replacing assertions with `Pytest` asserts; and (3) moving setup/teardown operations to `Pytest` fixtures. Steps 1 and 2 is relatively simple to apply because the migration between `unittest`

and `Pytest` is almost direct. For example, the test `test_vsplt` in Figure 1 only replaces the `unittest` assertions `assertIsInstance` and `assertEqual` by the `Pytest` `assert` statement.

On the other hand, migrating the remaining steps is harder to accomplish because there is no direct mapping between `unittest` and `Pytest`. For example, the `unittest` `setUp` method is split into four `Pytest` fixture functions in Figure 1: `prompt_buffer`, `editor_buffer`, `window`, and `tab_page`. The two tests (`test_initial` and `test_vsplt`) are then adapted to receive the fixtures via parameters in `Pytest`. When `Pytest` runs a test, it looks at the parameters of the test function and then searches for fixtures with the same names as those parameters [4, 41]. Once `Pytest` finds them, it runs those fixtures, captures what they returned, and passes those objects into the test function as arguments.

The number of popular Python projects that decided to migrate from `unittest` to `Pytest` increased from 27% to 37%, reinforcing the need for an automated solution to assist this migration process.

3 Study Design

3.1 Overview

Figure 2 summarizes our study design. The methodology begins with selecting real-world Python projects and detecting framework migrations. Next, migration commits are analyzed to build the `TestMigrationsInPy` dataset. We then selected migrations that were still available in the latest commit of their projects. These migrations were used in our experiment, where we asked LLMs to generate the migration for each scenario, replaced the original migration with the one produced by the LLM, and evaluated its correctness and coverage. Next, we detail each step.

3.2 Case Study

In this study, we analyze real-world and relevant software systems. We analyze the top-100 Python projects with the most stars on GitHub. This metric is widely adopted as a measure of the popularity of software projects [7, 51]. These 100 projects came from a prior study that empirically analyzed the migration from `unittest` to `Pytest` [4]. The set includes projects broadly adopted worldwide, such as Pandas (data analysis library), Flask (web development framework), Requests (library for performing HTTP requests), and Ansible (open-source automation software).²

3.3 Detecting Migrations from `Unittest` to `Pytest`

We relied on the tool proposed by Barbosa and Hora [4] to detect projects that migrated from `unittest` to `Pytest`. Basically, the tool traverses throughout the commit history and analyzes the *removed* and *added* lines of each commit. One commit is considered *migration commit* if at least one of the following rules applies:

- (1) **Assert migration:** the commit removes `unittest self.assert*` and adds `assert` keyword.
- (2) **Fixture migration:** the commit removes `unittest fixtures` (e.g., `setUp` and `tearDown`) and adds `Pytest fixtures` (e.g., `@pytest.fixture`).

²The complete list of systems can be found in the original dataset: <https://doi.org/10.5281/zenodo.5847361>.

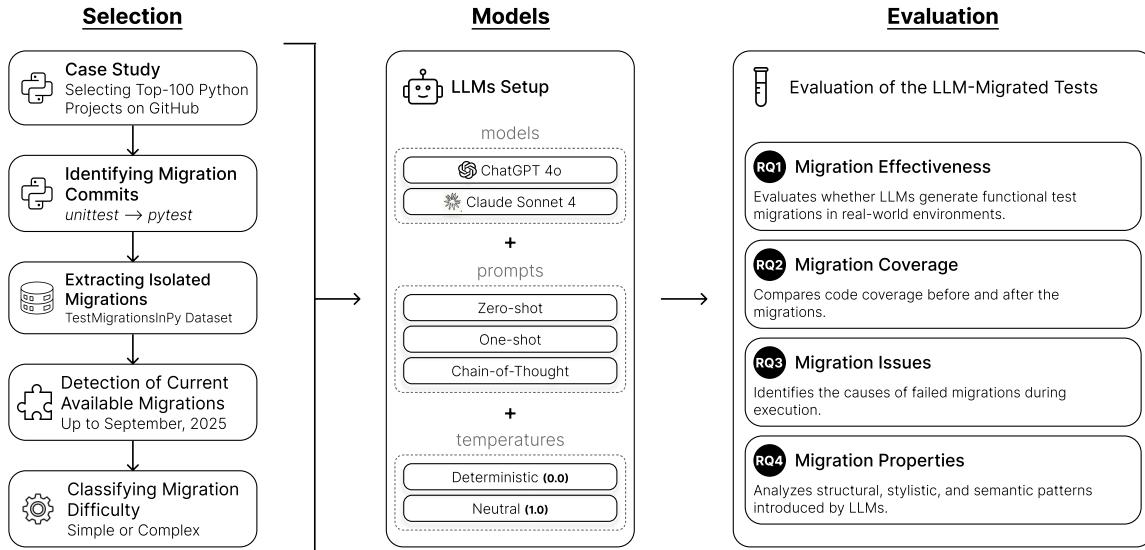


Figure 2: Overview of the study design.

- (3) **Import migration:** the commit removes `import unittest` and adds `import Pytest`.
- (4) **Skip migration:** the commit removes `unittest` test skips (e.g., `@unittest.skipIf`) and adds `Pytest` test skips (e.g., `@Pytest.mark.skipif`).
- (5) **Expected failure migration:** the commit removes `unittest` expected failure (i.e., `@unittest.expectedFailure`) and adds `Pytest` expected failure (i.e., `@Pytest.mark.xfail`).

For our study, we executed the migration detection tool on the top-100 selected projects, which detected 690 migration commits in 37 projects. This is a significant increase when compared to the original study, which found 330 migration commits in 27 projects [4].

3.4 Extracting Isolated Migrations

The next step in our study design is to create a dataset of migrations that can be used in our research as a ground truth.

We briefly describe the framework used to construct our *TestMigrationsInPy* dataset [3].³ The dataset was built based on the manual analysis of the migration commits collected in the previous step. It is important to notice that a migration commit may have one or more migrations from `unittest` to `Pytest`. However, it is well-known that commits may include unrelated (i.e., tangled) changes [17], e.g., it may perform migration and add/remove/update assertions. To avoid this problem, we focused on detecting *isolated migrations*, that is, migrations that simply replace `unittest` with `Pytest`, and no other unrelated changes are involved. Moreover, to avoid noise caused by large commits, we filtered commits that modified more than 5 files. Of the 690 migration commits collected in the previous step, we manually detected 923 isolated migrations that are used to create our dataset *TestMigrationsInPy*. This dataset is adopted as

the ground truth in this research, but it can be used by any other research in the context of framework migration.

Dataset: *TestMigrationsInPy* contains 923 real-world migrations from `unittest` to `Pytest` (<https://github.com/altinoalvesjunior/TestMigrationsInPy>).

3.5 Detection of Current Available Migrations

We applied a multi-step filtering process to keep only migrations that can be validated, i.e., can be executed at the time this study is conducted. From the 923 migrations available in the *TestMigrationsInPy* dataset, we first checked whether the migrated test files were present in the latest project versions. We then manually verified whether the migrated tests were still present in those files. Finally, we reproduced the development environment of each project to ensure it could be built and executed locally. For this paper, we only included for analysis the migrations satisfying all these criteria, considering project states up to September 30, 2025.

After applying these filtering steps, 883 out of the 923 examined migrations were excluded. In most cases (53%), the test files still existed, but the migrated methods were removed or their logic had changed substantially. Another 38% showed partial preservation, with only some methods remaining, often refactored or simplified. The remaining 9% referred to test files that had been deleted.

Finally, we selected 40 migrations to assess in this research. From these 40 migrations, 30 are classified as simple and 10 as complex cases according to our classification in Section 3.6. These 40 migrations originated from seven projects, as presented in Table 1.

3.6 Classifying Migration Difficulty

As discussed in Section 2, not all migrations have the same level of difficulty. For example, migrations involving only changes of

³The real dataset name is omitted due to the double-blind review.

(a) Zero-Shot prompt (base prompt used for all prompts).

Zero-Shot Prompt

Given the Python test code below, migrate it from the unittest framework to the Pytest framework. The migration should simply adapt the test code from unittest to Pytest, without adding new functionalities. The output should contain only the migrated code, without additional explanations. Ensure that the output is complete, keeping all original lines of the code that were not changed, including comments.

```
### TEST BEFORE MIGRATION ###
```

(b) Chain-of-Thought prompt

Chain-of-Thoughts Prompt

Use the steps below as a guide for the migration. You do not need to follow them exactly as described, but they should be able to help with the migration:

1. Analysis and Planning:
 - Identify the tests written with unittest.
 - Catalog the unittest assertion methods (assertEqual, assertTrue, assertIn, etc.).
 - Spot the unittest fixture methods such as setUp, tearDown, setUpClass, and tearDownClass.
2. Migration Procedure:
 - Remove the inheritance of unittest.TestCase.
 - Convert unittest fixture methods to Pytest fixtures.
 - Transform unittest test methods into Pytest test functions.
 - Replace unittest assertion methods (self.assertEqual*) with Pytest assert statements.
 - Adapt the test of exceptions in unittest (self.assertRaises) to the Pytest syntax (Pytest.raises).
 - Implement additional Pytest fixtures if necessary to optimize resource setup and teardown.
 - Use advanced Pytest features such as test parameterization if applicable.

```
### TEST BEFORE MIGRATION ###
```

(c) One-Shot prompt

One-Shot Prompt

Use the following test code blocks, enclosed in triple double quotes, as examples of migrations, and follow the same patterns used in it.

Code before migration (unittest):

```
"""
import unittest

class TestMyAPI(unittest.TestCase):

    def setUp(self):
        self.user_data = {"name": "John Doe", "email": "johndoe@example.com"}

    def test_create_user(self):
        response = create_user(self.user_data)
        self.assertEqual(response.status_code, 201)
        self.assertEqual(response.json()['name'], self.user_data['name'])

    def test_create_user_invalid_data(self):
        with self.assertRaises(ValueError):
            create_user({"name": ""})

"""

Code after migration (Pytest):
```

```
"""
import Pytest

@Pytest.fixture
def user_data():
    return {"name": "John Doe", "email": "johndoe@example.com"}

def test_create_user(user_data):
    response = create_user(user_data)
    assert response.status_code == 201
    assert response.json()['name'] == user_data['name']

def test_create_user_invalid_data():
    with Pytest.raises(ValueError):
        create_user({"name": ""})

"""


```

Figure 3: Prompt strategies adopted in this work.**Table 1: Origin of the selected migrations.**

owner/name	Stars	URL
ansible/ansible	66,900	https://github.com/ansible/ansible
apache/airflow	43,000	https://github.com/apache/airflow
ray-project/ray	39,600	https://github.com/ray-project/ray
httpie/httpie	36,900	https://github.com/httpie/httpie
cookiecutter/cookiecutter	22,200	https://github.com/cookiecutter/cookiecutter
beetbox/beets	14,100	https://github.com/beetbox/beets
redis/redis-py	13,300	https://github.com/redis/redis-py

assertions are potentially easier to perform than migrations of fixtures. To further explore this variation, we classify the migration into *simple* or *complex*.

Simple migrations involve only direct changes from unittest to Pytest, such as converting classes that inherit from `unittest.TestCase` into standalone test functions and replacing unittest assertions

(e.g., `self.assertEqual(a,b)`) with native assert statements. An example of a simple migration is `test_candle_int_4` from the termgraph project.⁴ *Complex migrations*, on the other hand, require structural adaptations and the use of additional Pytest resources. They typically involve fixtures, refactoring of `setUp/tearDown`, mocks, asynchronous tests, or interactions with external components such as I/O and/or network resources. They may also include adjustments to assertion logic. An example of a complex migration is `setupAttrs` from the airflow project.⁵

3.7 LLMs Setup

We selected OpenAI's GPT 4o because it represents the state of the art in code generation [40]. We also selected Anthropic's Claude Sonnet 4 due to its consistent superiority over other large language

⁴<https://github.com/sgeisler/termgraph/commit/d5665248b7d596cab0a5>

⁵<https://github.com/prompt-toolkit/pyvim/commit/7e1c7bf505cefba468>

models—including OpenAI’s, Google’s Gemini, and DeepSeek—in activities involving code understanding and generation [5, 21].

The prompt used to evaluate both models was inspired by prior research on library migration [1]. Specifically, we explore three different prompting methods: Zero-Shot [47], One-Shot [12] and Chain Of Thought [57]. While the Zero-Shot approach contains only the task description, the One-Shot strategy adds a single example, allowing the model to have access to a desired structure before generating the answer. The Chain-of-Thought approach introduces a step-by-step reasoning process that leads the model toward the outcome. Figure 3 details the prompts used in our research.

To ensure a balanced evaluation between determinism and creativity, we considered two temperature values: 0.0 (deterministic) and 1.0 (neutral). The deterministic value produces reproducible outputs, reflecting the model’s reliability in constrained scenarios [43]. The neutral configuration preserves the model’s default probability distribution and enables a balanced degree of creativity and diversity in the generated code [35].

Finally, for each of the 40 selected migrations, we created three prompts (Zero-Shot, One-Shot, and Chain-of-Thoughts), and executed each twice (under deterministic and neutral configurations) for both models (GPT 4o and Claude Sonnet 4). As a result, we performed 480 requests, i.e., $40 \text{ migrations} \times 3 \text{ prompts} \times 2 \text{ temperatures} \times 2 \text{ models}$. For this purpose, we developed a script to access GPT 4o and Claude Sonnet 4 APIs. The results were stored in structured format containing the migration returned as response and the request setup, i.e., model, prompt strategy, and temperature.

3.8 Evaluation of the LLM-Migrated Tests

To evaluate the effectiveness of the migration process, we executed all the *LLM-migrated test* on the original projects’ test suites. For each project, we cloned the latest available tag or release on GitHub (up to September 30th, 2025) into a local environment to reproduce its original setup. For example, we installed all required dependencies, downloaded necessary Docker images, and configured auxiliary components. The goal was to *ensure* a fully functional testing environment.

Next, we manually edited the original test files and replaced the migrated source code with one of the LLM-migrated versions; the rest of the project remained unchanged. For each replacement, we executed the test suite and recorded whether the test case had *PASSED* or *FAILED*. We also collected the project-level coverage for every execution using `coverage.py` [14]. This allowed us to quantify the structural impact of each migration and compare it against the baseline coverage of the original tests.

We examined the differences between the migration performed by the LLMs and the original one performed by the developers. Specifically, we first looked for duplicated answers to quantify how often the LLMs generated equivalent code despite variations in prompting configuration. Next, we extracted structural elements important to implement tests in Pytest, such as `assert`, `@pytest.fixture`, `pytest.raises`.

3.9 Research Questions

The goal of this study is to evaluate the effectiveness of LLMs at migrating tests from `unittest` to `Pytest`. We assessed the migrations

generated by the LLMs with respect to their correctness, coverage, issues, and changes. We propose four research questions:

RQ1: Can LLMs generate correct test migrations? This question explores whether the LLMs generate valid migrations. We consider that LLM-generated migrations are correct when they can be successfully executed in the test environment of their projects. We explore the migration in five perspectives: (i) model, (ii) temperature, (iii) strategy, (iv) difficulty, and (v) project.

RQ2: Do LLMs-generated migrations change test coverage? This question evaluates whether LLM-generated migrations not only produce valid tests, but if they keep their original intent. We hypothesize that changes in code coverage mean that a correct but different test was generated by the LLM.

RQ3: What are the errors in LLMs-generated migrations tests? Here, we investigate the cases where LLM-generated migrations failed to execute. This analysis aims to understand the underlying causes of these failures, such as structural inconsistencies, missing methods, or logical inconsistencies introduced during migration.

RQ4: Which changes are mostly introduced in LLMs-generated migrations tests? Finally, this RQ analyzes multiple properties of LLM-generated migrations, including changes in code structure and duplicated migrations.

4 Results

4.1 RQ1: Correctness of Test Migrations

We find that 247 out of 480 (51.5%) LLM-generated migrations have failed, i.e., returned *FAILED*. Conversely, 233 migrations were successfully executed, i.e., returned *PASSED*. Although the number of successful migrations almost matches the unsuccessful ones, most migrations did not preserve the functional behavior of the original tests. The remainder of this section focuses on analyzing the successful migrations in five perspectives: (i) model, (ii) temperature, (iii) strategy, (iv) difficulty, and (v) project.

4.1.1 By Model. Table 2 shows that Claude Sonnet 4 achieved slightly higher correctness, with 124 (51.66%) migrations returned *PASSED*, while GPT-4o reported 109 (45.41%). The difference of 6.25 points between the models is modest, but suggests that Claude may handle certain contextual and semantic aspects of the migration process more reliably. This benefit might be due to differences in how the models are built and fine-tuned. Recent research shows how these differences can affect the way LLMs find errors and change code [8, 42].

4.1.2 By Temperature. Claude Sonnet 4 maintains the same correctness at neutral and deterministic approaches, with 62 migrations *PASSED* in each case (124 in total, 53.21%), indicating low sensitivity to temperature, as shown in Table 2. GPT-4o shows only a minimum increase from 54 (23.17%) in *deterministic* to 55 (23.60%) in *neutral*. These results suggest that temperature has a negligible impact on correctness. Prior studies report similar stability across temperature ranges in reasoning and code-generation tasks [44, 59].

4.1.3 By Strategy. Overall, the Chain-of-Thought strategy yielded the best overall correctness, with 81 migrations marked as *PASSED*

Table 2: Successful migrations by model, temperature, strategy, and difficulty.

Migration Difficulty	#Migrations	Prompt Strategy	Model				Total	
			GPT 4o		Claude Sonnet 4			
			Deterministic (0.0)	Neutral (1.0)	Deterministic (0.0)	Neutral (1.0)		
Simple <i>(assertions and raises)</i>	30	Zero-shot	15	15	16	16	62	
		One-shot	14	15	15	15	59	
		Chain-of-Thought	16	16	16	16	64	
Complex <i>(fixtures, mocks, integrations, I/O)</i>	10	Zero-shot	2	2	6	6	16	
		One-shot	4	3	4	4	15	
		Chain-of-Thought	3	4	5	5	17	
Total			54	55	62	62	233	

(34.76%) across both models, as summarized in Table 2. Claude Sonnet 4 performed better under the Zero-shot approach, with 44 migrations returned *PASSED* (35.48%) out of the model’s 124 successful executions. GPT-4o, on the other hand, achieved its best correctness with the Chain-of-Thought strategy, with 39 (35.78%) migrations successfully executed out of the model’s 109 successful cases. Conversely, the One-shot approach yielded lower success rates for both models. Prior studies have shown that example-based prompting can induce representational bias or over-specialization toward the provided instance, limiting output diversity and generalization [6, 53]. A single example may lead LLMs to anchor their results on that specific instance and generate migrations consistent with the example but misaligned with other valid cases.

4.1.4 By Difficulty. *Simple* migrations presented higher success rates in both models, confirming that tasks requiring just syntactic adjustments are more easily captured by LLMs, as shown in Table 2. The best results were achieved under the Chain-of-Thought strategy, with 64 (27.46%) migrations performed successfully for both models. In contrast, the results vary more in *Complex* migrations. While Claude Sonnet was able to provide proper answers for 44 of the migration scenarios (Zero-shot), GPT-4o achieved its best (One-Shot and Chain-of-Thought). Nevertheless, both models struggled to preserve test logic structures such as fixtures and mocks.

4.1.5 By Project. Lastly, Table 3 summarizes the results by project. We detail the number of analyzed migrations, the distribution between simple and complex migrations, the total number of test executions—based on the 480 migration requests—and the percentage of successful outcomes.

The *ansible/ansible* project achieved the highest effectiveness, with 168 (99.4%) of successful executions; all successful migrations were classified as simple. Interestingly, *redis/redis-py* had a similar number of simple migrations, but none of them executed successfully. The *httpie/httpie* project obtained the second-best performance (156, 83.3%). Finally, *apache/airflow* and *ray-project/ray* reported no successful executions.

Table 3: Successful migrations by project.

Project	Number of Migrations		Exec.	Success	%
	#	Simple			
ansible/ansible	14	13	1	168	167
httpie/httpie	2	1	1	24	20
beetbox/beets	4	1	3	48	30
cookiecutter/cookiecutter	3	1	2	36	16
redis/redis-py	13	13	0	156	0
apache/airflow	3	0	3	36	0
ray-project/ray	1	1	0	12	0
Total	40	30	10	480	233
					-

RQ1. Overall, 48.5% (233 out of 480) of all migrations executed successfully. Chain-of-Thought yielded the best overall correctness. Claude Sonnet 4 achieved slightly higher correctness than GPT-4o. *Simple* migrations presented higher success rates in both models, but Claude Sonnet performed better in *complex* ones.

4.2 RQ2: Coverage in Test Migrations

In this RQ, we explore whether LLM-generated tests change the test coverage as compared to the original migration. Changes in test coverage may indicate that a correct but different test was generated by the models.

We find that the coverage remained exactly the same for all successful cases. For instance, *ansible/ansible* maintained 39%, *cookiecutter/cookiecutter* 33%, *httpie/httpie* 94%, and *beetbox/beets* 62.96%. These results suggest that LLM-generated migrations effectively reproduced the test behavior.

RQ2. Test coverage remained unchanged in successful migrations, i.e., the LLM-generated migrations preserved the test behavior.

4.3 RQ3: Errors in Test Migrations

This RQ explores the 247 cases where LLM-generated migrations failed to execute. We identified six major categories of errors across such LLM-generated test migrations that failed to execute. These categories are described below and represent distinct failure patterns observed during the test execution:

- **AssertionError:** subtle changes that led to failed assertions, such as rounding errors, string formatting changes, or different expected values.

Table 4: Unsuccessful migrations by model, temperature, strategy, and difficulty.

Error	owner/project	GPT 4o						Claude Sonnet 4						Total	
		Zero-shot		One-shot		Chain-of-Thought		Zero-shot		One-shot		Chain-of-Thought			
		0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0		
AssertionError	<i>ray-project/ray</i>	1	1	1	1	1	1	1	1	1	1	1	1	12	
Missing Fixtures	<i>httpie/httpie</i>	0	0	1	1	0	0	0	0	1	1	0	0	4	
Signature Drift	<i>beetbox/beets</i>	3	3	2	3	3	2	0	0	1	1	0	0	18	
Structural Mismatch	<i>cookiecutter/cookiecutter</i>	3	3	1	1	1	1	1	1	2	2	2	2	20	
SyntaxError	<i>ansible/ansible</i>	0	0	1	0	0	0	0	0	0	0	0	0	1	
	<i>apache/airflow</i>	3	3	3	3	3	3	3	3	3	3	3	3	36	
TypeError	<i>redis/redis-py</i>	13	13	13	13	13	13	13	13	13	13	13	13	156	
Total		23	23	22	22	21	20	18	18	21	21	19	19	247	

- **Structural Mismatch:** inconsistencies in the structural style or test organization that caused failures in test discovery or disrupted class-based dependencies.
- **Missing Fixtures:** missing references to fixtures, clients, or variables used in the tests.
- **TypeError:** misuse of data types or interactions with uninitialized objects, frequently linked to absent mocks or clients.
- **Signature Drift:** undesired modifications in function signatures, such as added, removed, or renamed parameters that cause incompatibility with other test components.
- **SyntaxError (Parse):** malformed instructions in the code, such as unbalanced parentheses, malformed try/except blocks, or indentation issues.

Table 4 details the results per category. Across all executions, a total of 247 failures were recorded out of 480 migrations, encompassing all combinations of models, prompting strategies, and temperature settings. Among the 40 unique migrations, 25 experienced at least one failure. When analyzing failures across configurations, we observed that GPT-4o produced the highest number of failing migrations under the Zero-shot strategy, while Claude Sonnet 4 did so under the One-shot strategy. Despite these variations, temperature had virtually no influence on the outcome—error counts remained identical between *Deterministic* (0.0) and *Neutral* (1.0) temperatures for all strategies, except for a marginal difference of one case in the Chain-of-Thought configuration for GPT-4o.

Interestingly, the number of errors reported by some types remained the same regardless of the LLM setup. For example, the number of *TypeError* remained 13 between all configurations; similar behavior happens with *SyntaxError* detected in *apache/airflow* and *AssertionError*. In *redis/redis-py*, most errors were caused by missing dependencies and incorrect fixture substitutions, which led to *TypeError* exceptions. Similarly, *apache/airflow* frequently failed due to malformed control blocks (e.g., unbalanced try/except statements), resulting in *SyntaxError* during module imports.

In contrast, *ansible/ansible* and *httpie/httpie* have more localized failures, mainly caused by broken test discovery when migrated functions were moved out of their original classes or lost their fixture associations. Finally, in *beetbox/beets* and *cookiecutter/cookiecutter*, signature drift and inheritance modifications disrupted shared setup routines, producing integration inconsistencies that propagated across dependent tests.

RQ3. Overall, 51.5% (247 out of 480) of all migrations failed the execution. They represent 25 out of the 40 unique migrations. Most issues involved problems in handling dependencies, incorrect fixture usage, or inconsistencies in structural elements, such as test setup and inheritance.

4.4 RQ4: Changes in Test Migrations

To guide this analysis, we organized the discussion into two perspectives: (i) changes in code structure and (ii) duplicated migrations.

4.4.1 Changes in Code Structure. LLMs exhibited distinct structural changes when migrating tests from *unittest* to *Pytest*, as detailed in Table 5. Overall, Claude Sonnet 4 presented a conservative migration style, preserving most of the original test code across the multiple configurations. For example, Claude preserved class-oriented organization in 70.83% of migrations and maintained references to *unittest* in 33.75%. In contrast, GPT-4o performed more substantial changes. Classes were preserved in 45% of the cases, the *unittest* package is mentioned in 9.17% of the tests. Despite these differences, both models fully removed *unittest self.assert**, mostly adopting native *Pytest assert*.

Related to prompt strategies, we observe that the number of classes declined substantially (Claude 49%, GPT 2.5%) when using One-shot; the number of *import pytest* statements also increased in this scenario. The Zero-shot, on the other hand, generated outputs structured and aligned with the original tests. Claude preserved class structures in 100% of cases and GPT in 88%. Pytest's advanced commands are more present in migrations generated by the *Chain-of-Thoughts* strategy. Both GPT-4o and Claude achieved the highest results for *@pytest.fixture* and *pytest.raises()* in migrations performed using this prompt.

Temperature variation had a negligible impact on these trends. Across all prompting settings, the difference between deterministic and neutral configurations remained below 2.5 p.p. for most metrics. Nonetheless, the deterministic approach produced more consistent code—presented lower variations in fixture usage and import organization—whereas neutral generations introduced only minor stylistic differences without semantic impact.

4.4.2 Duplicated Migrations. Overall, we identified 117 duplicated migration groups, encompassing 348 variants (72.5% of all

Table 5: Code metrics before and after migration across models, prompts, and temperatures.

Model	Prompt	Temp.	Total	class	test_*	import Pytest	import unittest	@pytest.fixture	pytest.raises()	self.assert*	assert (Pytest)	LOC (mean)
Code (before)	–		40	100%	87.50%	7.50%	100%	0%	0%	85%	10%	25
GPT 4o	Zero-shot	1.0	40	87.50%	90%	40%	10%	5%	15%	0%	77.50%	19
		0.0	40	87.50%	90%	40%	7.50%	5%	15%	0%	85%	19
	One-shot	1.0	40	2.50%	92.50%	100%	7.50%	22.50%	15%	0%	85%	19
		0.0	40	2.50%	90%	97.50%	7.50%	22.50%	15%	0%	85%	19
	CoT	1.0	40	42.50%	92.50%	87.50%	12.50%	32.50%	17.50%	0%	87.50%	20
		0.0	40	47.50%	90%	90%	10%	30%	17.50%	0%	85%	20
	<i>Total</i>		240	45%	90.83%	75.83%	9.17%	19.58%	15.83%	0%	84.17%	19
Claude Sonnet 4	Zero-shot	1.0	40	100%	90%	40%	20%	0%	15%	0%	85%	19
		0.0	40	100%	90%	42.50%	17.50%	0%	15%	0%	85%	19
	One-shot	1.0	40	50%	90%	95%	45%	7.50%	15%	0%	85%	20
		0.0	40	47.50%	90%	92.50%	35%	7.50%	15%	0%	85%	20
	CoT	1.0	40	65%	90%	80%	42.50%	7.50%	15%	0%	85%	19
		0.0	40	62.50%	90%	82.50%	42.50%	12.50%	17.50%	0%	85%	19
	<i>Total</i>		240	70.83%	90%	72.08%	33.75%	5.83%	15.42%	0%	85%	19

generated migrations). Each group represents a set of identical migration outputs produced under different model or configuration settings, while variants refer to the individual instances within those groups. This highlights a marked convergence in model behavior, where different configurations frequently result in identical code generations. As shown in Table 6, Claude Sonnet 4 exhibited the highest degree of redundancy, with 89.17% of its variants being identical across at least one configuration, compared to 55.83% for GPT-4o. At the prompt level, Claude maintained near-deterministic behavior, reaching duplication rates of 92.50% under Zero-shot, 90% under One-shot, and 85% under Chain-of-Thought. In contrast, GPT produced more diverse and variable generations, with duplication ranging from 67.50% in Zero-shot to 45% in Chain-of-Thought.

Temperature variation had minimal impact on duplication rates. Claude’s outputs remained nearly identical across the *precise* (88.33%) and *neutral* (90%) configurations, while GPT consistently produced around 55.83% duplicates in both. These observations suggest that prompt design—rather than temperature—plays the dominant role in driving convergence among generated code variants.

RQ4. Claude Sonnet 4 preserved class-based structures and *unittest* elements, while GPT-4o favored function-based rewrites with Pytest elements. Duplicated migrations were frequent, revealing convergence toward stable migration templates, especially for Claude (up to 92.50%).

5 Discussion and Implications

LLMs underperformed the migration of complex cases. Both models achieved consistent results in simple migrations, but their performance declined in complex scenarios. While 51% of simple migrations were executed successfully, this performance drops to 40% when considering only complex ones (RQ1). We believe that these failures are primarily due to the models’ limited capacity to analyze the architectural aspects of the tests being migrated. For example, most execution failures involved incorrect fixtures and test setup. This behavior was also noticed in other works, where LLMs struggle to find the correct outcome in larger scenarios [52].

Temperature setup did not impact the migrations. Despite being theoretically designed to influence creativity and exploration, temperature had minimal impact on the migration outcomes. As observed in RQ1, both models produced almost identical structural outcomes across all configurations, i.e., they produced the same set of successful tests independent of migration difficulty or prompt strategy. Similar behavior is also noted for failed tests (RQ3); a higher temperature value did not help the LLMs to overcome with an alternative solution that was correct. These findings emphasize

Table 6: Duplicated migrations.

Model	Prompt	Temp.	Total	Duplications
GPT 4o	Zero-Shot	0.0	40	26
		1.0	40	27
	One-Shot	0.0	40	23
		1.0	40	22
	Chain-of-Thought	0.0	40	18
		1.0	40	18
Claude Sonnet 4	Zero-Shot	0.0	40	37
		1.0	40	37
	One-Shot	0.0	40	36
		1.0	40	36
	Chain-of-Thought	0.0	40	34
		1.0	40	34
Total			480	328

that, for local code transformation tasks, prompt design and access to relevant project context have more influence on outcome quality than random variation in model outputs [18, 19].

No prompt configuration outperformed the others. Across all prompt strategies, no configuration consistently outperformed the others. As observed in RQ1, while Chain-of-Thought occasionally produced slightly more structured migrations, the differences were minimal. Despite the variations in prompting strategy, both models frequently generated correct migrations for similar scenarios, indicating a strong dependency on common transformation patterns. Once a migration pattern is explicitly mentioned—such as converting `setUp()` to `@pytest.fixture`—the LLMs started using it with more frequency [13, 18]. By contrast, adding a source code example can occasionally introduce some bias to the migration process. This happens with the One-shot prompt configuration, where it consistently added the same fixture-based setup pattern from the provided example—particularly inserting unnecessary `@pytest.fixture` functions even in tests that did not originally rely on shared fixtures. This suggests that source code examples can constrain the LLMs capacity to provide a more diverse output.

Migration executions failed mostly due to context-sensitive instructions, such as fixtures and test setup. Most of the execution failures observed in RQ3 originated from missing dependencies that were not explicitly represented in the test files. While the LLMs successfully translated the syntax from `unittest` to `Pytest`, they frequently ignored the relationship between fixtures, setup routines, and external components to the tests under migration. For instance, in `redis/redis-py`, fixture-based clients were replaced with direct object instantiations (e.g., `redis.StrictRedis()`), breaking the controlled initialization of shared Redis connections and resulting in `TypeError` exceptions. Other failures, however, stemmed from structural and semantic inconsistencies rather than missing dependencies: in `apache/airflow`, malformed control structures—such as unbalanced `try/except` blocks—caused syntax errors during import, while in `ray-project/ray`, subtle rounding precision mismatches produced false `AssertionError` failures. In either case, we found no LLM setup able to fix these changes, suggesting additional information is needed to properly detect and fix these errors. These findings highlight the need for approaches that incorporate detailed, fine-grained information about the test environment, allowing LLMs to understand and leverage contextual relationships among test components.

Advanced commands are used only if explicitly mentioned in the prompt. As shown in RQ4, both models rarely applied advanced Pytest features unless explicitly mentioned in the prompt. Even though Pytest supports richer constructs, such as parameterization, for example, the models typically performed direct one-to-one transformations from `unittest`, avoiding higher-level abstractions. In other words, the models favor a conservative transformation over a more idiomatic one, even when these could lead to more maintainable test suites. This behavior is consistent with previous studies showing that LLMs focus on reproducing familiar syntax instead of fully understanding and applying the deeper semantics of the target framework [18, 19]. The effective automation of testing framework migrations passes by adopting techniques to

encourage the model to generate more idiomatic outcomes, such as guided prompts and finetuning strategies [28, 50].

6 Threats to Validity

Internal Validity. Although all projects were executed in a controlled environment, different dependencies' versions or project configurations could still influence the results. Also, the non deterministic behavior of LLMs may cause slight variations in outputs between executions. To mitigate this, we used fixed prompts, temperature settings, and the number of executions for each LLM.

Construct Validity. A threat to construct validity arises from how migration success was defined and measured. Execution and coverage preservation are practical indicators of correctness but do not capture all aspects of test quality. A migrated test may still pass while behaving differently when integrated into the full suite. To mitigate this, all migrations were executed in real project environments, using coverage as an additional quality measure to reduce false positives—cases where tests pass but fail to preserve their original intent.

External Validity. The external validity of our findings is constrained by two main factors. First, we isolated the migration from other changes performed on the system. Second, we analyzed a limited number of models and projects. To mitigate these, we selected widely adopted open-source projects and validated migrations in real execution environments, ensuring that the evaluated scenarios reflect realistic development settings.

Conclusion Validity. We relied on the execution of the migrated tests to analyze the performance of LLMs. Although all migrations were executed under controlled local environments, minor changes in dependencies or package versions could have influenced individual outcomes. We rebuild the projects from their verified releases and executed them multiple times to ensure stability and reproducibility of results.

7 Related Work

Library and framework evolution and migration are research topics largely explored by the literature in multiple ecosystems [4, 10, 11, 26, 31, 33, 34, 37, 39, 45, 48, 49, 56, 58]. In the context of testing framework migration, Barbosa and Hora [4] empirically explored how developers migrate Python tests from `unittest` to `Pytest`. The authors detect that multiple popular Python projects migrated to `Pytest`. In many cases, the migration was not simple, taking a long period to conclude or never concluded at all.

Recently, Large Language Models (LLMs) have been adopted in multiple software engineering tasks, including generating tests, refactoring, fixing bugs, and supporting code review [1, 2, 16, 20, 22, 24, 27, 36, 38, 50, 54]. Di Rocco *et al.* proposed DeepMig, a transformer-based approach to support coupled library and code migrations in Java [16]. The research presents promising results, showing that DeepMig is able to recommend both libraries and code; in several projects with a perfect match. Almeida *et al.* provided an initial study to explore automatic library migration using LLMs [1]. Specifically, with the support of GPT-4o, the authors migrated a client application to a newer version of SQLAlchemy, a Python Object-Relational Mapping (ORM) library. The study

presents promising results, concluding that LLMs can correctly migrate the project with only minor mistakes. Our research contributes to the literature with a novel solution based on LLMs to support testing framework migration.

8 Conclusion

Our study analyzed how Large Language Models perform automated test migrations from `unittest` to `Pytest`. Across 40 isolated migrations using GPT-4o and Claude Sonnet 4, with three prompting strategies (Zero-shot, One-shot, and Chain-of-Thought) and two temperature settings (0.0 – deterministic and 1.0 – neutral), LLMs achieved an overall 48.54% effectiveness rate. This effectiveness was validated through real executions on 7 of the top 100 Python open-source projects on GitHub, showing that even under isolated, context-free conditions, LLM-generated migrations can maintain partial functionality in real-world software environments.

Despite this potential, several challenges remain. Both models often struggled with structural coherence, fixture adaptation, and dependency management. Claude Sonnet 4 exhibited a conservative migration style—preserving class-based architectures and legacy `unittest` references, while GPT-4o favored more transformations toward function-oriented and fixture-driven designs. Prompt strategy emerged as a key factor: One-shot and Chain-of-Thoughts improved syntactic modernization but reduced architectural fidelity. Temperature variation, however, had a negligible impact on results.

Finally, future work may explore (i) test migrations performed by coding agents, which can perform software testing tasks autonomously [25, 32, 46], (ii) contribution studies [9, 15, 23] to evaluate developer acceptance by submitting LLM-migrated tests as pull requests to open-source projects, (iii) a deeper assessment of the quality of LLM-migrated tests in comparison to manually migrated ones, and (iv) analysis on large, industry-grade repositories that rely on advanced `Pytest` features.

Acknowledgments

This research was supported by CNPq (process 403304/2025-3), CAPES, and FAPEMIG. This work was partially supported by INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence), www.ines.org.br, CNPq grant 408817/2024-0.

References

- [1] Aylton Almeida, Laerte Xavier, and Marco Túlio Valente. 2024. Automatic Library Migration Using Large Language Models: First Results. In *International Symposium on Empirical Software Engineering and Measurement*. 1–7.
- [2] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandra Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated unit test improvement using large language models at meta. In *International Conference on the Foundations of Software Engineering*. 185–196.
- [3] Altino Alves and Andre Hora. 2025. TestMigrationsInPy: A Dataset of Test Migrations from `Unittest` to `Pytest`. In *International Conference on Mining Software Repositories*. IEEE, 841–845.
- [4] Livia Barbosa and Andre Hora. 2022. How and why developers migrate Python tests. In *International Conference on Software Analysis, Evolution and Reengineering*. 538–548.
- [5] Ali Bayram, Gonca Gökce Menekse Dalveren, and Mohammad Derawi. 2025. Comparative Analysis of AI Models for Python Code Generation: A HumanEval Benchmark Study. *Applied Sciences* 15, 18 (2025), 9907.
- [6] Emily M Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. 2021. On the dangers of stochastic parrots: Can language models be too big? In *ACM Conference on fairness, accountability, and transparency*. 610–623.
- [7] Hudson Borges, Andre Hora, and Marco Túlio Valente. 2016. Understanding the Factors that Impact the Popularity of GitHub Repositories. In *International Conference on Software Maintenance and Evolution (ICSM)*. 334–344.
- [8] Mohamed Boukhelif, Nassim Kharroum, and Mohamed Hanine. 2024. LLMs for intelligent software testing: a comparative study. In *International Conference on Networking, Intelligent Systems and Security*. 1–8.
- [9] Carolin Brandt, Ali Khatami, Maiquel Wessel, and Andy Zaidman. 2024. Shaken, Not Stirred: How Developers Like Their Amplified Tests. *IEEE Transactions on Software Engineering* (2024).
- [10] Aline Brito, Marco Túlio Valente, Laerte Xavier, and Andre Hora. 2020. You broke my code: understanding the motivations for breaking changes in APIs. *Empirical Software Engineering* 25 (2020), 1458–1492.
- [11] Gleison Brito, Andre Hora, Marco Túlio Valente, and Romain Robbes. 2018. On the use of replacement messages in API deprecation: An empirical study. *Journal of Systems and Software* 137 (2018), 306–321.
- [12] Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. 2025. Unleashing the potential of prompt engineering for large language models. *Patterns* (2025).
- [13] Wentao Chen, Lizhe Zhang, Li Zhong, Letian Peng, Zilong Wang, and Jingbo Shang. 2025. Memorize or generalize? evaluating LLM code generation with evolved questions. *arXiv preprint arXiv:2503.02296* (2025).
- [14] Coverage.py. January, 2026. <https://coverage.readthedocs.io>.
- [15] 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* 24 (2019), 2603–2635.
- [16] Juri Di Rocco, Phuong T Nguyen, Claudio Di Sipio, Riccardo Rubei, Davide Di Ruscio, and Massimiliano Di Penta. 2025. DeepMig: A transformer-based approach to support coupled library and code migrations. *Information and Software Technology* 177 (2025), 107588.
- [17] Martin Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. 2015. Untangling fine-grained code changes. In *International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 341–350.
- [18] Yihong Dong, Yuchen Liu, Xue Jiang, Zhi Jin, and Ge Li. 2025. Rethinking Repetition Problems of LLMs in Code Generation. *arXiv preprint arXiv:2505.10402* (2025).
- [19] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating large language models in class-level code generation. In *International Conference on Software Engineering*. 1–13.
- [20] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. *arXiv preprint arXiv:2310.03533* (2023).
- [21] Tianchen Gao, Jiashun Jin, Zheng Tracy Ke, and Gabriel Moryoussef. 2025. A comparison of deepseek and other LLMs. *arXiv preprint arXiv:2502.03688* (2025).
- [22] Roar Elias Georgsen. 2023. Beyond Code Assistance with GPT-4: Leveraging GitHub Copilot and ChatGPT for Peer Review in VSE Engineering. Technical Report. EasyChair.
- [23] Andre Hora. 2024. PathSpotter: Exploring Tested Paths to Discover Missing Tests. In *International Conference on the Foundations of Software Engineering*. 647–651.
- [24] Andre Hora. 2024. Predicting Test Results without Execution. In *International Conference on the Foundations of Software Engineering*. 542–546.
- [25] Andre Hora and Romain Robbes. 2026. Are Coding Agents Generating Over-Mocked Tests? An Empirical Study. In *International Conference on Mining Software Repositories (MSR 2026)*. In press.
- [26] Andre Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, and Marco Túlio Valente. 2015. How do developers react to api evolution? the pharo ecosystem case. In *International Conference on Software Maintenance and Evolution*. IEEE, 251–260.
- [27] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* (2023).
- [28] Cristina Improta, Rosalia Tufano, Pietro Ligouri, Domenico Cotroneo, and Gabriele Bavota. 2025. Quality In, Quality Out: Investigating Training Data's Role in AI Code Generation. In *33rd International Conference on Program Comprehension (ICPC)*. 454–465.
- [29] Md Mohayeminul Islam, Ajay Kumar Jha, May Mahmoud, Ildar Akhmetov, and Sarah Nadi. 2025. An Empirical Study of Python Library Migration Using Large Language Models. *arXiv:2504.13272 [cs.SE]* <https://arxiv.org/abs/2504.13272>
- [30] JetBrains. Python Developers Survey 2024 Results. January, 2026. <https://lp.jetbrains.com/python-developers-survey-2024>.
- [31] Maxime Lamothé, Yann-Gaël Guéhéneuc, and Weiyi Shang. 2021. A systematic review of API evolution literature. *ACM Computing Surveys (CSUR)* 54, 8 (2021), 1–36.
- [32] Hao Li, Haoxiang Zhang, and Ahmed E Hassan. 2025. 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).

- [33] Jun Li, Yingfei Xiong, Xuanzhe Liu, and Lu Zhang. 2013. How does web service API evolution affect clients?. In *International Conference on Web Services*. IEEE, 300–307.
- [34] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. 2018. Characterising deprecated android apis. In *International Conference on Mining Software Repositories (MSR)*. 254–264.
- [35] Lujun Li, Lama Sleem, Niccolò Gentile, Geoffrey Nichil, and Radu State. 2025. Exploring the Impact of Temperature on Large Language Models: Hot or Cold? *arXiv preprint arXiv:2506.07295* (2025).
- [36] Jenny T Liang, Carmen Badea, Christian Bird, Robert DeLine, Denae Ford, Nicole Forsgren, and Thomas Zimmermann. 2023. Can GPT-4 Replicate Empirical Software Engineering Research? *arXiv preprint arXiv:2310.01727* (2023).
- [37] Brian A Malloy and James F Power. 2019. An empirical analysis of the transition from python 2 to python 3. *Empirical Software Engineering* 24 (2019), 751–778.
- [38] Mauricio Monteiro, Bruno Castelo Branco, Samuel Silvestre, Guilherme Avelino, and Marco Tulio Valente. 2023. End-to-End Software Construction using ChatGPT: An Experience Report. *arXiv preprint arXiv:2310.14843* (2023).
- [39] Romulo Nascimento, Eduardo Figueiredo, and Andre Hora. 2021. JavaScript API deprecation landscape: A survey and mining study. *IEEE Software* 39, 3 (2021), 96–105.
- [40] OpenAI. 2023. GPT-4 Technical Report. *arXiv:2303.08774 [cs.CL]*
- [41] Pytest. January, 2026. <https://docs.pytest.org>.
- [42] Rudolf Ramler, Philipp Straubinger, Reinhold Plösch, and Dietmar Winkler. 2025. Unit Testing Past vs. Present: Examining LLMs' Impact on Defect Detection and Efficiency. *arXiv preprint arXiv:2502.09801*.
- [43] Matthew Renze. 2024. The effect of sampling temperature on problem solving in large language models. In *Findings of the association for computational linguistics: EMNLP 2024*. 7346–7356.
- [44] Madelon Renze and Eren Guven. 2024. The Effect of Sampling Temperature on Problem Solving in Large Language Models. *Findings of the Association for Computational Linguistics: EMNLP 2024* (2024). <https://arxiv.org/abs/2402.05201>
- [45] Romain Robbes, Mircea Lungu, and David Röthlisberger. 2012. How do developers react to API depreciation? The case of a Smalltalk ecosystem. In *International Symposium on the Foundations of Software Engineering*. 1–11.
- [46] Romain Robbes, Théo Matricon, Thomas Degueule, Andre Hora, and Stefano Zacchiroli. 2026. Promises, Perils, and (Timely) Heuristics for Mining Coding Agent Activity. In *International Conference on Mining Software Repositories (MSR 2026)*. In press.
- [47] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. 2024. A systematic survey of prompt engineering in large language models: Techniques and applications. *arXiv preprint arXiv:2402.07927* (2024).
- [48] Anand Ashok Sawant, Guangzhe Huang, Gabriel Vilen, Stefan Stojkovski, and Alberto Bacchelli. 2018. Why are features deprecated? an investigation into the motivation behind deprecation. In *International Conference on Software Maintenance and Evolution*. IEEE, 13–24.
- [49] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. 2019. To react, or not to react: Patterns of reaction to API depreciation. *Empirical Software Engineering* 24 (2019), 3824–3870.
- [50] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* (2023).
- [51] Hudson Silva and Marco Tulio Valente. 2018. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software* 146 (2018), 112–129.
- [52] Luciana Lourdes Silva, Janis Rosa da Silva, João Eduardo Montandon, Marcus Andrade, and Marco Tulio Valente. 2024. Detecting code smells using chatgpt: Initial insights. In *International Symposium on Empirical Software Engineering and Measurement*. 400–406.
- [53] Yasuaki Sumita, Koh Takeuchi, and Hisashi Kashima. 2025. Cognitive Biases in Large Language Models: A Survey and Mitigation Experiments (SAC '25). Association for Computing Machinery, New York, NY, USA, 1009–1011. doi:10.1145/3672608.3707812
- [54] Michele Tufano, Shubham Chandel, Anisha Agarwal, Neel Sundaresan, and Colin Clement. 2023. Predicting Code Coverage without Execution. *arXiv preprint arXiv:2307.13383* (2023).
- [55] Unittest. January, 2026. <https://docs.python.org/3/library/unittest.html>.
- [56] Jiawei Wang, Li Li, Kui Liu, and Haipeng Cai. 2020. Exploring how deprecated Python library APIs are (not) handled. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 233–244.
- [57] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.
- [58] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. 2017. Historical and impact analysis of API breaking changes: A large-scale study. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 138–147.
- [59] Yuqi Zhu, Jia Li, Ge Li, Yunfei Zhao, Zhi Jin, and Hong Mei. 2024. Hot or cold? adaptive temperature sampling for code generation with large language models. In *AAAI Conference on Artificial Intelligence*, Vol. 38. 437–445.