TestMigrationsInPy: A Dataset of Test Migrations from Unittest to Pytest

Altino Alves, Andre Hora
Department of Computer Science, UFMG
Belo Horizonte, Brazil
{altinojunior, andrehora}@dcc.ufmg.br

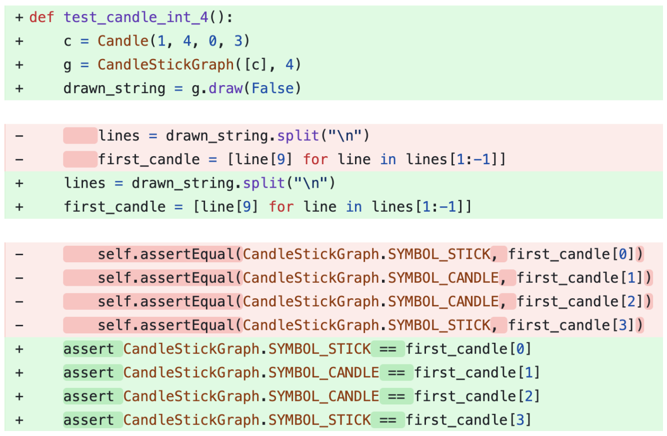
***Abstract*—Unittest and pytest are the most popular testing frameworks in Python. Overall, pytest provides some advantages, including simpler assertion, reuse of fixtures, and interoperability. Due to such benefits, multiple projects in the Python ecosystem have migrated from unittest to pytest. To facilitate the migration, pytest can also run unittest tests, thus, the migration can happen gradually over time. However, the migration can be time-consuming and take a long time to conclude. In this context, projects would benefit from automated solutions to support the migration process. In this paper, we propose TestMigrationsInPy, a dataset of test migrations from unittest to pytest. TestMigrationsInPy contains 923 real-world migrations performed by developers. Future research proposing novel solutions to migrate frameworks in Python can rely on TestMigrationsInPy as a ground truth. Moreover, as TestMigrationsInPy includes information about the migration type (*e.g.,* changes in assertions or fixtures), our dataset enables novel solutions to be verified effectively, for instance, from simpler assertion migrations to more complex fixture migrations. TestMigrationsInPy is publicly available at: https://github.com/altinoalvesjunior/TestMigrationsInPy.**

***Index Terms*—Software Testing, Framework Migration, LLMs, Software Repository Mining, unittest, pytest**

I. Introduction

Unittest [1] and pytest [2] are the most popular testing frameworks in Python [3], [4]. Unittest belongs to the Python standard library and pytest is a third-party testing framework. In pytest, tests can be regular functions, while unittest tests are contained in classes that inherit from `TestCase`. Consequently, pytest tests tend to be less verbose than unittest ones. Another difference is the assertions: unittest provides `self.assert*` methods, while pytest allows developers to use the regular `assert` statement for verifying expectations and values. Overall, pytest provides some advantages compared to unittest, including simpler assertion, reuse of fixtures, and interoperability [2], [4].

Due to such benefits, multiple projects in the Python ecosystem have migrated from unittest to pytest [4]. A prior study found that 27% of top-100 most popular Python projects migrated or were migrating to pytest [4]. Figure 1 presents a migration from unittest to pytest in project Termgraph.[1] The unittest `self.assertEqual` methods (red) are replaced by `assert` statements in pytest (green). As the pytest test becomes a function, the inheritance is not needed anymore.



```
+ def test_candle_int_4():
+     c = Candle(1, 4, 0, 3)
+     g = CandleStickGraph([c], 4)
+     drawn_string = g.draw(False)

-         lines = drawn_string.split("\n")
-         first_candle = [line[9] for line in lines[1:-1]]
+     lines = drawn_string.split("\n")
+     first_candle = [line[9] for line in lines[1:-1]]

-         self.assertEqual(CandleStickGraph.SYMBOL_STICK, first_candle[0])
-         self.assertEqual(CandleStickGraph.SYMBOL_CANDLE, first_candle[1])
-         self.assertEqual(CandleStickGraph.SYMBOL_CANDLE, first_candle[2])
-         self.assertEqual(CandleStickGraph.SYMBOL_STICK, first_candle[3])
+     assert CandleStickGraph.SYMBOL_STICK == first_candle[0]
+     assert CandleStickGraph.SYMBOL_CANDLE == first_candle[1]
+     assert CandleStickGraph.SYMBOL_CANDLE == first_candle[2]
+     assert CandleStickGraph.SYMBOL_STICK == first_candle[3]
```

Fig. 1: Migration from unittest to pytest (Termgraph).

A migration that involves setups and teardowns may be harder to achieve because there is no direct mapping between unittest and pytest. For example, in project pyvim, the unittest `setUp` method is split into four pytest fixture functions.[2] To facilitate the migration process, pytest can also run unittest tests, meaning that Python test suites can have both testing frameworks simultaneously. Consequently, the migration can happen gradually over time. However, there are also drawbacks: the migration process can be time-consuming and take a long time [4]. In this context, projects would benefit from automated solutions to support the migration process.

In this paper, we propose TestMigrationsInPy, a dataset of test migrations from unittest to pytest. TestMigrationsInPy contains 923 real-world migrations performed by developers. Our dataset construction has two major steps: (1) we automatically detect commits with migrations from unittest to pytest, and (2) we manually inspect the migration commits to identify isolated migrations. We envision the following usages for TestMigrationsInPy. First, future research proposing novel solutions to migrate frameworks in Python can rely on TestMigrationsInPy as a ground truth. Second, as TestMigrationsInPy also includes information about the migration type (*e.g.,* changes in assertions or fixtures), our dataset enables novel solutions to be verified effectively, from simpler assertion migrations to more complex fixture migrations.

[1] https://github.com/sgeisler/termgraph/commit/d5665248b7d596cabe0a5

[2] https://github.com/prompt-toolkit/pyvim/commit/7e1c7bfb505cefba468

Originality: To our knowledge, this is the first dataset in the context of testing framework migration. Particularly, we focus on a highly relevant migration in the Python ecosystem: unittest to pytest [4].

Data Availability: TestMigrationsInPy is publicly available at: https://github.com/altinoalvesjunior/TestMigrationsInPy.

II. Dataset Construction

Our dataset construction has two major steps. First, we automatically detect commits with migrations from unittest to pytest (Section II-A). Second, we manually inspected the migration commits to identify isolated migrations, that is, migrations that simply replace unittest by pytest (Section II-B).

A. Detecting Migrations from Unittest to Pytest

We rely on the migration detection tool proposed by Barbosa and Hora [4] to detect systems that migrated from unittest to pytest. We adopt this tool because it has a precision and recall of 100% in detecting migrations. The tool classifies the system as *migrated* (or as *is migrating*) when it has at least one migration commit. A *migration commit* is a commit that explicitly migrates code from unittest to pytest. To detect migration commits, it assesses the version history of the system with the support of PyDriller [5]. For a given system, it iterates on its commits and analyzes the *removed* and *added* lines of code per commit. A commit is a *migration commit* if at least one of the following migration types is true [4]:

1) **Assert migration**: the commit removes unittest `self.assert*` and adds pytest `assert` statements.
2) **Fixture migration**: the commit removes unittest setups/teardowns, and adds pytest fixtures.
3) **Import migration**: the commit removes `import unittest` and adds `import pytest`.
4) **Skip migration**: the commit removes unittest test skips and adds pytest test skips.
5) **Expected failure migration**: the commit removes unittest expected failure and adds pytest expected failure.

B. Detecting Isolated Migrations

The dataset is built based on the manual analysis of 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 commis may include unrelated (*i.e.,* tangled) changes [6], *e.g.,* it may perform migration and add/remove/update assertions. To avoid this problem, we manually detect *isolated migrations*, that is, migrations that simply replace unittest by pytest, and no other unrelated changes are involved. Moreover, to avoid noise caused by large commits, we filter commits that modified more than 5 file tests.

III. Dataset Description

Our dataset has been curated from the test suites of 100 highly popular Python software systems. The 100 selected systems come from our study that empirically analyzed the migration from unittest to pytest [4]. It includes systems that are broadly adopted worldwide, such as Pandas, Flask, Requests, Cookiecutter, Aiohttp, Ansible, to name a few.[3]

First, we executed the migration detection tool (as described in Section II-A) in the 100 selected projects and detected 690 migration commits in 37 projects. Of the 690 migration commits, we manually detected 923 isolated migrations (described in Section II-B), which were used to create our dataset. Thus, TestMigrationsInPy contains 923 real-world migrations from unittest to pytest.

Dataset Structure: To facilitate navigation in our dataset, we organized it as a repository in GitHub. Each project has a list of migration commits. Each migration commit has a list of isolated migrations and their respective code (located in folder `diff`) and a migration summary (located in file `output.info`):

- Migration code (`diff`): contains the migrations of a commit; each migration includes the test code before (with unittest) and after (with pytest) the migration.
- Migration summary (`output.info`): contains the commit hash, the number of changed files, the number of migrations in the commit, and type. The "type" attribute includes information about the migration type, for example, whether it changes assertions or fixtures (see an example in the next section).

IV. Dataset Examples

Migrations from unittest to pytest do not have the same level of difficulty. For example, migrating assertions may be simple because one only needs to replace the unittest assertions with pytest ones and adapt the data being compared. In contrast, a migration that replaces unittest setups/teardowns by pytest fixtures may be harder to achieve because there is no direct mapping. Next, we present two examples to illustrate such scenarios.

A. Example 1: Simple Migration of Assertions

Project Saleor has 1 migration commit.[4] Such migration commit has 4 migrations, as summarized in its `diff`[5] and detailed in `output.info`[6] (see Figure 2). Note that the "type" attribute informs us that these migrations only involve assertion changes. For instance, migration #1 migrates test `test_facebook_login_url` from unittest[7] to pytest,[8] as detailed in Figure 3. In this case, the unittest `self.assert*` statements are replaced by the pytest `assert` ones. Notice that it is needed to adapt the data being compared, for instance, `self.assertEquals(func,`

[3]The complete list of systems can be found in the original dataset: https://doi.org/10.5281/zenodo.5594254.

[4]List of migration commits in Saleor: https://github.com/altinoalvesjunior/TestMigrationsInPy/tree/main/projects/saleor

[5]Migration code (Saleor, commit #1): https://github.com/altinoalvesjunior/TestMigrationsInPy/tree/main/projects/saleor/1/diff

[6]Migration summary (Saleor, commit #1): https://github.com/altinoalvesjunior/TestMigrationsInPy/blob/main/projects/saleor/1/output.info

[7]Test before: https://github.com/altinoalvesjunior/TestMigrationsInPy/blob/main/projects/saleor/1/diff/mig1-before-test_registration.py

[8]Test after: https://github.com/altinoalvesjunior/TestMigrationsInPy/blob/main/projects/saleor/1/diff/mig1-after-test_registration.py

`oauth_callback)` is replaced by `assert func is oauth_callback`. Other changes would be required depending on the assertion being used in unittest. For instance, `self.assertIn(a,b)` should be replaced by `assert a in b`, and `self.assertIsInstance(a,b)` should be replaced by `assert isinstance(a, b)`, to name a few.

```
"commit_hash": "d629135f843de17ba96dee176816ed6d76fd54ce",
"commit_link": "https://github.com/mirumee/saleor/commit/d629135f843de17ba96dee176816ed6d76fd54ce",
"testFilesChanged": 4,
"testFilesChangedWithMigrations": 1,
"filesWithMigration": ["saleor/registration/test_registration.py"],
"type": ["assert", "testcase"],
"numberOfMigration": 4
```

Fig. 2: Example of `output.info` file (Saleor, commit #1).

```
class LoginUrlsTestCase(TestCase):
    """Tests login url generation."""

    def test_facebook_login_url(self):
        """Facebook login url is properly generated"""
        facebook_client = FacebookClient(local_host='localhost')
        facebook_login_url = URL(facebook_client.get_login_uri())
        query = facebook_login_url.query_params()
        callback_url = URL(query['redirect_uri'][0])
        func, _args, kwargs = resolve(callback_url.path())
        self.assertEquals(func, oauth_callback)
        self.assertEquals(kwargs['service'], FACEBOOK)
        self.assertEqual(query['scope'][0], FacebookClient.scope)
        self.assertEqual(query['client_id'][0], str(FacebookClient.client_id))
```

(a) Test code before the migration (with unittest).

```
def test_facebook_login_url():
    facebook_client = FacebookClient(local_host='localhost')
    facebook_login_url = URL(facebook_client.get_login_uri())
    query = facebook_login_url.query_params()
    callback_url = URL(query['redirect_uri'][0])
    func, _args, kwargs = resolve(callback_url.path())
    assert func is oauth_callback
    assert kwargs['service'] == FACEBOOK
    assert query['scope'][0] == FacebookClient.scope
    assert query['client_id'][0] == str(FacebookClient.client_id)
```

(b) Test code after the migration (with pytest).

Fig. 3: Example of migration of asserts (Saleor, commit #1, migration #1)

B. *Example 2: Complex Migration of Fixtures and Assertions*

Project Dash has 4 migration commits.[9] Migration commit #1 has 9 migrations, as presented in its `diff`[10] and summarized in its `output.info`[11] (see Figure 4). Notice that the "type" attribute shows that the migration involves both assertion and fixture changes. As an example, migration #4 migrates a setup and a test from unittest[12] to pytest,[13] as detailed in Figure 5. In this case, there are three major changes. First, the unittest test class `TestConfigs` (with inheritance to `TestCase`) is removed, as it is not needed in pytest. Second, the unittest `setUp` method is replaced by the pytest fixture `empty_environ`, which is annotated with `@pytest.fixture`. Third, the unittest test method `test_pathname_prefix_from_environ_app_name` is replaced by the pytest test function with the same name. The new pytest test function receives the fixture `empty_environ` as a parameter. When pytest runs a test, it looks at the parameters in that test function's signature and then searches for fixtures with the same names as those parameters [2], [4]. Once pytest finds them, it runs those fixtures, captures what they returned, and passes those values into the test function as arguments [2], [4].

As a more challenging migration of fixtures, in project pyvim, the unittest `setUp` method is split into four pytest fixture functions: `prompt_buffer`, `editor_buffer`, `window`, and `tab_page`.[14]

[9]List of migration commits in Dash: https://github.com/altinoalvesjunior/TestMigrationsInPy/tree/main/projects/dash

[10]Migration code (Dash, commit #1): https://github.com/altinoalvesjunior/TestMigrationsInPy/tree/main/projects/dash/1/diff

[11]Migration summary (Dash, commit #1): https://github.com/altinoalvesjunior/TestMigrationsInPy/blob/main/projects/dash/1/output.info

[12]Test before: https://github.com/altinoalvesjunior/TestMigrationsInPy/blob/main/projects/dash/1/diff/mig4-before-test_configs.py

[13]Test after: https://github.com/altinoalvesjunior/TestMigrationsInPy/blob/main/projects/dash/1/diff/mig4-after-test_configs.py

```
"commit_hash": "bfe4e9f6f16da07990fe31a6be5956162b3b0fae",
"commit_link": "https://github.com/plotly/dash/commit/bfe4e9f6f16da07990fe31a6be5956162b3b0fae",
"testFilesChanged": 1,
"testFilesChangedWithMigrations": 1,
"filesWithMigration": ["tests/unit/test_configs.py"],
"type": ["assert", "fixture", "testcase", "add Param"],
"numberOfMigration": 9
```

Fig. 4: Example of `output.info` file (Dash, commit #1).

```
class TestConfigs(unittest.TestCase):

    def setUp(self):
        for k in DASH_ENV_VARS.keys():
            if k in os.environ:
                os.environ.pop(k)

    def test_pathname_prefix_from_environ_app_name(self):
        os.environ['DASH_APP_NAME'] = 'my-dash-app'
        _, routes, req = pathname_configs()
        self.assertEqual('/my-dash-app/', req)
        self.assertEqual('/', routes)
```

(a) Test code before the migration (with unittest).

```
@pytest.fixture
def empty_environ():
    for k in DASH_ENV_VARS.keys():
        if k in os.environ:
            os.environ.pop(k)

def test_pathname_prefix_from_environ_app_name(empty_environ):
    os.environ["DASH_APP_NAME"] = "my-dash-app"
    _, routes, req = pathname_configs()
    assert req == "/my-dash-app/"
    assert routes == "/"
```

(b) Test code after the migration (with pytest).

Fig. 5: Example of migration fixtures and assertions (Dash, commit #1, migration #4)

[14]https://github.com/prompt-toolkit/pyvim/commit/7e1c7bfb505cefba468

V. Dataset Usage: Support the Development of Novel Framework Migration Solutions

Multiple studies explore migration, both empirically [4], [7] and by proposing automatic migration solutions [6], [8]–[11]. For example, Barbosa and Hora empirically explored how developers migrate Python tests from unittest to pytest [4]. In a related line, Martinez and Mateus studied the migration from Java to Kotlin [7], [12]. Recently, Large Language Models (LLMs) have been adopted in multiple software engineering tasks [8], [13]–[21], including code migration [8], [22]. Di Rocco *et al.* proposed DeepMig, a transformer-based approach to support coupled library and code migrations in Java [22]. Almeida *et al.* provided an initial study to explore automatic library migration using LLMs [8]. Both LLM-based migrations presented promising results [8], [22].

Our dataset contains real-world migrations performed by developers from the testing framework unittest to pytest. We manually verified the migrations to reduce the possible noise caused by unrelated, tangled changes [6]. Therefore, we foresee the following usages for TestMigrationsInPy.

Usage 1: Future research proposing novel solutions to migrate frameworks in Python can rely on TestMigrationsInPy as a ground truth. For example, to explore migration with LLMs, the test code before (with unittest) can be migrated with such models, and the test code after (with pytest) can be used as the ground truth for the LLM-migrated test.

Our dataset also includes information about the migration type, such as changes in assertions or fixtures. As migrations of fixtures are more complex than migrations of assertions, they can be classified according to their difficulty level.

Usage 2: TestMigrationsInPy enables novel migration solutions to be verified effectively, from simpler assertion migrations to more complex fixture migrations.

In this context, we have used GPT-4o and TestMigrationsInPy to migrate Python tests from unittest to pytest. Overall, our initial results show that GPT-4o can be used to accelerate the migration process from unittest to pytest. However, we observed that developers should pay attention to fixing minor wrong updates that the model can perform, particularly when migrating fixtures.

VI. Limitations

Our dataset has been curated from the test suites of 100 highly popular Python software systems [4]. Despite these systems being relevant and real-world, they do not represent the entire Python ecosystem. Particularly, less popular projects may adopt migration practices that are uncommon in more widely-used projects. Further versions of the dataset can include migrations from more projects, allowing a more comprehensive understanding of the migration landscape.

VII. Related Work

A. Code Migration

Framework evolution and migration are research topics largely explored by the literature in multiple ecosystems [4], [23]–[35]. In the context of testing framework migration, Barbosa and Hora empirically explored how developers migrate Python tests from unittest to pytest [4]. In many cases, the migration was not simple, taking a long period to conclude or never concluded at all. In a related research line, Martinez and Mateus studied the migration from Java to Kotlin [7], [12]. Kotlin is interoperable with Java, thus, developers can migrate gradually. Overall, the migration occurred to access features only available in Kotlin and to obtain safer code.

Recently, LLMs have been adopted in multiple software engineering tasks, including generating tests, refactoring, fixing bugs, and supporting code review [8], [13]–[22]. Notably, it has demonstrated significant results in code generation [13], [36]. In this context, Di Rocco *et al.* proposed DeepMig, a transformer-based approach to support coupled library and code migrations in Java [22]. The research presents promising results, showing that DeepMig can 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 [8]. With GPT-4o, the authors migrated a client application to a newer version of SQLAlchemy, a Python ORM library. TestMigrationsInPy can support the development of novel solutions to migrate frameworks in Python.

B. Datasets to Support Testing Research

Multiple datasets have been proposed to support software testing research [37]–[41]. For example, TestDossier is a dataset of tested values automatically extracted from the execution of Python tests [37]. Methods2Test is a dataset of focal methods mapped to test cases extracted from Java projects [38]. TestRoutes is a test-to-code dataset containing traceability information for Java test cases [39]. Jbench is a dataset of data races for concurrency testing [40]. TestMigrationsInPy contributes to the software testing literature with a novel dataset to support test migration research.

VIII. Conclusion

We proposed TestMigrationsInPy, a dataset of test migrations from unittest to pytest. TestMigrationsInPy contains real-world migrations performed by developers.

Further Improvements: First, our dataset can include less popular projects to have a better overview of the migration landscape. Second, the migrations can have a more specific classification for the unittest test code, for instance, with specific used assertions and fixtures (*e.g.,* `setUp`, `setUpClass`, `tearDown`, etc.). Third, the pytest test code can have a more specific classification of the pytest features used in the migrated test (*e.g.,* parametrized tests).

Acknowledgments

This research is supported by CAPES, CNPq, and FAPEMIG.

References

[1] Unittest, https://docs.python.org/3/library/unittest.html, October, 2024.

[2] Pytest, https://docs.pytest.org, October, 2024.

[3] Python Developers Survey 2023 Results, https://lp.jetbrains.com/python-developers-survey-2023/#frameworks-and-libraries, October, 2024.

[4] L. Barbosa and A. Hora, "How and why developers migrate python tests," in *International Conference on Software Analysis, Evolution and Reengineering*, 2022, pp. 538–548.

[5] D. Spadini, M. Aniche, and A. Bacchelli, "Pydriller: Python framework for mining software repositories," in *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 908–911.

[6] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," in *International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2015, pp. 341–350.

[7] M. Martinez and B. G. Mateus, "How and Why did developers migrate Android Applications from Java to Kotlin? A study based on code analysis and interviews with developers," *arXiv preprint arXiv:2003.12730*, 2020.

[8] A. Almeida, L. Xavier, and M. T. Valente, "Automatic library migration using large language models: First results," in *International Symposium on Empirical Software Engineering and Measurement*, 2024, pp. 1–7.

[9] A. Hora and M. T. Valente, "apiwave: Keeping track of API popularity and migration," in *International Conference on Software Mintenance and Evolution*, 2015, pp. 321–323.

[10] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining api mapping for language migration," in *International Conference on Software Engineering*, 2010, pp. 195–204.

[11] H. D. Phan, A. T. Nguyen, T. D. Nguyen, and T. N. Nguyen, "Statistical migration of api usages," in *International Conference on Software Engineering Companion*. IEEE, 2017, pp. 47–50.

[12] M. Martinez and B. G. Mateus, "Why did developers migrate Android applications from Java to Kotlin?" *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4521–4534, 2021.

[13] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," *arXiv preprint arXiv:2310.03533*, 2023.

[14] M. Monteiro, B. C. Branco, S. Silvestre, G. Avelino, and M. T. Valente, "End-to-end software construction using chatgpt: An experience report," *arXiv preprint arXiv:2310.14843*, 2023.

[15] J. T. Liang, C. Badea, C. Bird, R. DeLine, D. Ford, N. Forsgren, and T. Zimmermann, "Can gpt-4 replicate empirical software engineering research?" *arXiv preprint arXiv:2310.01727*, 2023.

[16] M. Tufano, S. Chandel, A. Agarwal, N. Sundaresan, and C. Clement, "Predicting code coverage without execution," *arXiv preprint arXiv:2307.13383*, 2023.

[17] R. E. Georgsen, "Beyond code assistance with gpt-4: Leveraging github copilot and chatgpt for peer review in vse engineering," EasyChair, Tech. Rep., 2023.

[18] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, 2023.

[19] A. Hora, "Predicting test results without execution," in *International Conference on the Foundations of Software Engineering*, 2024, pp. 542–546.

[20] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, 2023.

[21] N. Alshahwan, J. Chheda, A. Finogenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, "Automated unit test improvement using large language models at meta," in *International Conference on the Foundations of Software Engineering*, 2024, pp. 185–196.

[22] J. Di Rocco, P. T. Nguyen, C. Di Sipio, R. Rubei, D. Di Ruscio, and M. Di Penta, "Deepmig: A transformer-based approach to support coupled library and code migrations," *Information and Software Technology*, vol. 177, p. 107588, 2025.

[23] M. Lamothe, Y.-G. Guéhéneuc, and W. Shang, "A systematic review of api evolution literature," *ACM Computing Surveys (CSUR)*, vol. 54, no. 8, pp. 1–36, 2021.

[24] J. Wang, L. Li, K. Liu, and H. Cai, "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*, 2020, pp. 233–244.

[25] L. Xavier, A. Brito, A. Hora, and M. T. Valente, "Historical and impact analysis of api breaking changes: A large-scale study," in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 138–147.

[26] L. Li, J. Gao, T. F. Bissyandé, L. Ma, X. Xia, and J. Klein, "Characterising deprecated android apis," in *International Conference on Mining Software Repositories (MSR)*, 2018, pp. 254–264.

[27] A. Brito, M. T. Valente, L. Xavier, and A. Hora, "You broke my code: understanding the motivations for breaking changes in apis," *Empirical Software Engineering*, vol. 25, pp. 1458–1492, 2020.

[28] A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente, "How do developers react to api evolution? the pharo ecosystem case," in *International Conference on Software Maintenance and Evolution*. IEEE, 2015, pp. 251–260.

[29] G. Brito, A. Hora, M. T. Valente, and R. Robbes, "On the use of replacement messages in api deprecation: An empirical study," *Journal of Systems and Software*, vol. 137, pp. 306–321, 2018.

[30] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to api deprecation? the case of a smalltalk ecosystem," in *International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.

[31] J. Li, Y. Xiong, X. Liu, and L. Zhang, "How does web service api evolution affect clients?" in *International Conference on Web Services*. IEEE, 2013, pp. 300–307.

[32] A. A. Sawant, R. Robbes, and A. Bacchelli, "To react, or not to react: Patterns of reaction to api deprecation," *Empirical Software Engineering*, vol. 24, pp. 3824–3870, 2019.

[33] A. A. Sawant, G. Huang, G. Vilen, S. Stojkovski, and A. Bacchelli, "Why are features deprecated? an investigation into the motivation behind deprecation," in *International Conference on Software Maintenance and Evolution*. IEEE, 2018, pp. 13–24.

[34] R. Nascimento, E. Figueiredo, and A. Hora, "JavaScript API deprecation landscape: A survey and mining study," *IEEE Software*, vol. 39, no. 3, pp. 96–105, 2021.

[35] B. A. Malloy and J. F. Power, "An empirical analysis of the transition from python 2 to python 3," *Empirical Software Engineering*, vol. 24, pp. 751–778, 2019.

[36] OpenAI, "Gpt-4 technical report," 2023.

[37] A. Hora, "Testdossier: A dataset of tested values automatically extracted from test execution," in *International Conference on Mining Software Repositories*. IEEE, 2024, pp. 299–303.

[38] M. Tufano, S. K. Deng, N. Sundaresan, and A. Svyatkovskiy, "Methods2test: A dataset of focal methods mapped to test cases," in *International Conference on Mining Software Repositories*, 2022, pp. 299–303.

[39] A. Kicsi, L. Vidács, and T. Gyimóthy, "Testroutes: A manually curated method level dataset for test-to-code traceability," in *International Conference on Mining Software Repositories*, 2020, pp. 593–597.

[40] J. Gao, X. Yang, Y. Jiang, H. Liu, W. Ying, and X. Zhang, "Jbench: a dataset of data races for concurrency testing," in *International Conference on Mining Software Repositories*, 2018, pp. 6–9.

[41] E. Bui and H. Rocha, "Snapshot testing dataset," in *International Conference on Mining Software Repositories*. IEEE, 2023, pp. 558–562.