How and Why Developers Migrate Python Tests

Lívia Barbosa, Andre Hora
Department of Computer Science
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte, Brazil
{liviaab,andrehora}@dcc.ufmg.br

***Abstract*—Nowadays, Python developers can rely on two major testing frameworks: unittest and pytest. Due to the benefits of pytest (*e.g.,* fixture reuse), several relevant projects in the Python ecosystem have migrated from unittest to pytest. Despite being performed by the Python community, we are not yet aware of how systems are migrated from unittest to pytest nor the major reasons behind the migration. In this paper, we provide the first empirical study to assess testing framework migration. We analyze *how* and *why* developers migrate from unittest to pytest. We mine 100 popular Python systems and assess their migration status. We find that 34% of the systems rely on both testing frameworks and that Python projects are moving to pytest. While some systems have fully migrated, others are still migrating after a long period, suggesting that the migration is not always straightforward. Overall, the migrated test code is smaller than the original one. Furthermore, developers migrate to pytest due to several reasons, such as the easier syntax, interoperability, easier maintenance, and fixture flexibility/reuse, however, the implicit mechanics of pytest is a concern. We conclude by discussing practical implications for practitioners and researchers.**

***Index Terms*—Software Testing, Software Repository Mining, Software Maintenance, unittest, pytest**

I. Introduction

Software testing is a key practice in modern software development. Nowadays, developers rely on a variety of testing frameworks to write test cases, catch regressions and bugs, and support healthy software evolution. In Python, which is among the most important programming languages in current days, developers can rely on two major testing frameworks: unittest [1] and pytest [2].

Unittest belongs to the Python standard library, whereas pytest is a third-party testing framework. While unittest is the default solution for Python developers since it is a native package, pytest has gained attention in the Python ecosystem. Indeed, pytest provides features not available in unittest, such as flexibility to create and reuse testing fixtures and simpler assertions [2]. Due to such benefits, several projects in the Python ecosystem have migrated from unittest to pytest. For example, in project Opsdroid[1] (a popular chatbot framework), there is an explicit call for this migration:[2]

> **Migrate tests from unittest to pytest #1502**
>
> Our existing test suite has been written with the Python unittest framework. However, as the test suite has grown and Opsdroid has become more complex we are running into issues with the tests. Mainly around setting up and tearing down tests. The team has decided that we want to migrate all tests to be written with the pytest framework instead so that we can make better use of fixtures. Fixtures are more reusable and portable and should help reduce complexity all over.

In fact, this migration movement is somehow common: we assessed the top-100 most popular Python systems in GitHub and we find that over 1/4 migrated or is migrating from unittest to pytest. To facilitate the migration, pytest can also run unittest tests, that is, Python projects can have both testing frameworks at the same time. Thus, the migration can happen incrementally over time, without the need for a single-shot migration. However, there is also a downside: the migration process may be slow, taking a long time to be concluded due to the complexity of the test suites. During this period in which the migration is not complete, the test suite may become even more complex as two testing frameworks are used interchangeably. For instance, projects like NumPy (a popular scientific computing tool) and scikit-learn (a popular machine learning library) have started their migration but did fully not conclude yet.

Despite being largely performed by the Python community, we are not yet aware of *how software projects are migrated* from unittest to pytest nor *the reasons behind the migrations*. This knowledge can be used to understand migration practices, supporting the production of a more efficient migration process. For example, this can be used to create migration guidelines, decrease the migration duration, and foment the creation of novel migration tools. Moreover, Python has a rich software ecosystem, which is the basis for a variety of — sometimes critical[3] — applications. Thus, having proper automated testing is fundamental to ensure quality and sustainable evolution [4], [5]. While API migration is a research topic broadly studied by prior literature (*e.g.,* [6]–[10]), to the best of our knowledge, the migration of *testing* frameworks

[1] https://opsdroid.dev
[2] https://github.com/opsdroid/opsdroid/issues/1502

[3] For example, NumPy was used to support Black Hole imaging and the detection of gravitational waves [3].

has never been deeply explored by the research community.

In this paper, we provide the first empirical study to assess testing framework migration. We analyze *how* and *why* developers migrate Python tests from unittest to pytest. First, we mine 100 popular Python projects and analyze their migration status. Then, we assess migration explanations in issues and pull requests to understand migration rationales. Our research questions address the migration extension, frequency, duration, transformations, and reasons:

- *RQ0 (extension): To what extent are unittest and pytest adopted in the Python ecosystem over time?* Most systems (77 out of 100) rely on unittest (20%), pytest (23%), or both (34%). Projects with unittest are moving to pytest: 66% of the ones initially with unittest now rely on pytest.
- *RQ1 (frequency): How frequent is code migrated from unittest to pytest?* From the 39 systems that started with unittest and adopted pytest over time, 28% (11) have fully migrated to pytest and 41% (16) are still migrating.
- *RQ2 (duration): How long does it take to migrate from unittest to pytest?* The migration may be fast or take a long period to be concluded, from months to years. Overall, projects that migrated successively kick-started it in different development phases, but the majority tends to concentrate the migration commits.
- *RQ3 (transformations): What code is migrated from unittest to pytest?* Most migration commits (90%) include assert migrations. Developers also tend to migrate fixtures (18%) and imports (13%). Overall, the migrated test code is 34% smaller than the original one.
- *RQ4 (reasons): Why is code migrated from unittest to pytest?* Developers migrate to pytest mostly due to the easier syntax, interoperability, easier maintenance, and fixture flexibility/reuse. In contrast, the implicit mechanics of pytest is the main concern.

We provide empirical evidence that test migration is common in the Python ecosystem. The migration delay suggests that performing it is not always trivial. Overall, the migrated test code is smaller than the original one, meaning that there is less test code to be maintained. Moreover, developers focus on discussing the advantages of the migration, however, some disadvantages are also highlighted. Based on our findings, we discuss implications for both practitioners and researchers. First, (i) we reveal a set of practices and reasons in favor and against migration. We shed light on (ii) the challenge of keeping track of migration and (iii) how migration issues can be improved with migration guidelines. Finally, we discuss novel research directions, including (iv) the investigation of the reasons why some projects start the migration but do not conclude it and (v) the proposal of novel techniques to document and automate the migration.

Contributions. The contributions of this paper are threefold: (i) we provide the first empirical assess the migration of *testing* frameworks; (ii) we present how code is migrated and the reasons for the migration; and (iii) we propose practical implications for practitioners and researchers.

II. Background and Motivation

A. Unittest and Pytest in a Nutshell

Unittest [1] and pytest [2] are the most popular testing frameworks in Python. Unittest belongs to the Python standard library, while pytest is a third-party testing framework. Unittest was originally inspired by JUnit [11] and has a similar flavor as major unit testing frameworks in other programming languages [1]. Pytest makes it easy to write small tests and scales to support complex functional testing [2].

Figure 1 presents an example of a test method written with (a) unittest and (b) pytest. In short, unittest relies on inheritance to create tests (*i.e.,* the test need to extend the unittest class `TestCase`), whereas pytest tests can be regular functions, with the *test* prefix. As a consequence, pytest tests tend to be less verbose than unittest ones. Another difference is the assertions: unittest provides `self.assert*` methods (*e.g.,* `assertTrue`, `assertEqual`, etc.), while pytest allows developers to use the regular Python `assert` for verifying expectations and values. There are many other differences,[4] for example, pytest facilitates the creation of parameterized tests and the reuse of fixtures.

```
1. from unittest import TestCase
2. class MyTestClass(TestCase):
3.     def test_foo(self):
4.         self.assertTrue(True)
```

(a) unittest

```
1. def test_foo():
2.     assert True
```

(b) pytest

Fig. 1: Example of a test written with unittest and pytest.

Overall, the community acknowledges some advantages of pytest. In project Opsdroid, a core developer states: "*Fixtures are more reusable and portable and should help reduce complexity all over*". The pytest documentation [2] mentions that there is "*No need to remember self.assert* names*". Reddit has multiple posts comparing both testing frameworks,[5] *e.g.,* "*Tests are shorter, easier to read, with more reusability and extensibility, and have better output*" and "*Parameterising allows you to run the same test in multiple configurations*".

B. Migrating from Unittest to Pytest

Despite unittest being a builtin Python package, the Python community seems to be moving to pytest. The fact that pytest tests can run unittest tests may facilitate the migration. Migrating from unittest to pytest would involve at least the following steps: (1) removing test from class and moving to regular functions; (2) replacing assertions by regular Python asserts; and (3) moving setup operations to fixtures. Figure 2 presents a migration in project TermGraph,[6] which performs steps (1) and (2). Notice that the class `CandleTests` is removed, the test methods are moved to regular test functions, and the assertions are replaced by regular asserts.

[4]Summary of the major differences: https://git.io/Jnc1m
[5]https://bit.ly/3cMPit7 and https://bit.ly/3zyEEzB
[6]TermGraph migration commit: https://git.io/Jncy5

unittest

```
1  - from unittest import TestCase
2    from termgraph import CandleStickGraph, Candle
3
4
5  - class CandleTests(TestCase):
6  -     def test_candle_int_4(self):
7  -         c = Candle(1, 4, 0, 3)
8  -         g = CandleStickGraph([c], 4)
9  -         drawn_string = g.draw(False)
10
11 -         lines = drawn_string.split("\n")
12 -         first_candle = [line[9] for line in lines[1:-1]]
13
14 -         self.assertEqual(CandleStickGraph.SYMBOL_STICK, first_candle[0])
15 -         self.assertEqual(CandleStickGraph.SYMBOL_CANDLE, first_candle[1])
16 -         self.assertEqual(CandleStickGraph.SYMBOL_CANDLE, first_candle[2])
17 -         self.assertEqual(CandleStickGraph.SYMBOL_STICK, first_candle[3])
```

pytest

```
1    from termgraph import CandleStickGraph, Candle
2
3
4  + def test_candle_int_4():
5  +     c = Candle(1, 4, 0, 3)
6  +     g = CandleStickGraph([c], 4)
7  +     drawn_string = g.draw(False)
8
9  +     lines = drawn_string.split("\n")
10 +     first_candle = [line[9] for line in lines[1:-1]]
11
12 +     assert CandleStickGraph.SYMBOL_STICK == first_candle[0]
13 +     assert CandleStickGraph.SYMBOL_CANDLE == first_candle[1]
14 +     assert CandleStickGraph.SYMBOL_CANDLE == first_candle[2]
15 +     assert CandleStickGraph.SYMBOL_STICK == first_candle[3]
```

Fig. 2: Class and assertion migration (TermGraph, d56652).

However, the migration is not always that straightforward. For example, Figure 3 presents a migration in Pyvim,[7] which performs steps (1), (2), and (3). In this case, the `setUp` method in unittest (*i.e.,* the fixture) is split into four fixture functions in pytest, which are annotated with `@pytest.fixture`. The test function `test_initial` is then adapted to receive the fixtures via parameters (*i.e.,* `window` and `tab_page`). In practice, when pytest runs a test, it looks at the parameters in that test function's signature and then searches for fixtures that have the same names as those parameters [2]. Once pytest finds them, it runs those fixtures, captures what they returned, and passes those objects into the test function as arguments [2]. Feature reuse is considered one advantage of pytest. However, it is worth noticing that, the larger and the more complex are the fixtures in unittest, the more challenging is the migration to pytest.

unittest

```
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])
```

pytest

```
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]
```

Fig. 3: Fixture migration (pyvim, 7e1c7b).

C. The Migration Movement

To better understand this phenomenon, we have inspected the migration status of the top-100 most popular Python software systems hosted on GitHub. We find that 27 out of 100 systems have fully migrated or are currently migrating to pytest (this data is further explored in RQ0). Among those projects, we find worldwide ones, like Flask (55K stars, one of the most popular web frameworks nowadays), scikit-learn (machine learning module), pandas (data analysis library), and NumPy (scientific computing tool), to name a few.

In addition to popular Python projects, we also inspected the overall GitHub ecosystem. By using the GitHub Search API, we searched for issues with the terms "*unittest to pytest*",[8] looking for issues that explicitly mention the migration. We find around 12K issues (700 open and 11,300 closed), suggesting that the migration is somehow common in Python. As an illustration, we present examples of recent issues:

- Shift from unittest to pytest: "*Unittest is cool, but pytest is cooler. It packs more features and as this continues to grow I can see myself leveraging them further for automated testing*" (DPY-Anti-Spam, issue #64).
- Switch from unittest to pytest: *"I like this because of the simple usage of assert with a condition following for testing rather than using assert**" (gcam_reader, issue #19).
- Move completely from unittest to pytest: *"The testing uses unittests in some tests and pytest in others. Move completely to pytest*" (df-wizard-chess, issue #5).

While the first two issues are initial calls for the migration, the last one is a recall to conclude the migration.

> The migration from unittest to pytest is widespread in the Python ecosystem. Better understating this migration can reveal novel practices, advantages, and disadvantages. This can support, for example, the production of novel migration guidelines and warn about the existence of migration bottlenecks.

III. Study Design

A. The Python Ecosystem

In this study, we assess test migration in the Python ecosystem. We select Python due to several reasons. *First*, Python is among the most important programming languages nowadays according to both GitHub[9] and TIOBE[10] rankings. *Second*, Python has a rich software ecosystem with worldwide adopted projects, including web frameworks, machine learning libraries, data analysis libraries, and scientific computing tools, to name a few, which are highly well-tested. *Third*, the testing landscape in Python is dominated by unittest [1] and pytest [2]; this does not happen in other popular programming languages like Java and JavaScript, in which a single or multiple testing frameworks are available. *Lastly*, the migration movement from unittest to pytest makes a relevant case to be investigated: not only the Python community itself can benefit from this assessment, but also any other software community experiencing similar migration issues.

B. Case Studies

We aim to study real-world and relevant software systems. Therefore, we collect the top-100 most popular Python software systems hosted on GitHub according to the number of

[7]pyvim migration commit: https://git.io/Jn8xh

[8]https://git.io/Jn8AL

[9]GitHub Languages ranking: https://git.io/JnOlr

[10]TIOBE ranking: https://www.tiobe.com/tiobe-index

stars, which is a metric largely adopted in the software mining literature as a proxy of popularity [12], [13]. In this process, we took special care to filter out non-software projects, such as tutorials, examples, samples, among others.

The 100 selected software systems are presented in our publicly available dataset. It includes systems that are broadly adopted worldwide, such as Django, Pandas, and Scikit-learn, to name a few. On the median, they have 1,800 commits, 165 authors, and 2,355 days since the first commit, showing that they are active and relevant projects. Our dataset is publicly available at https://doi.org/10.5281/zenodo.5847361.

C. Detecting Testing Frameworks Over Time

To explore how the software systems migrate from unittest to pytest, we first need to discover the testing frameworks the projects have used over time. For this purpose, we analyze both the *present* and the *past* versions of the software system. Specifically, we look for references to the APIs provided by the testing frameworks unittest [1] and pytest [2] and classify the system as follows (the notation "*past* → *present*" represents the testing frameworks used in past and present versions):

1) *unittest* → *pytest*: the present version of the system references pytest only, but its past versions referenced unittest only.
2) *unittest* → *unittest & pytest*: the present version of the system references both unittest and pytest, but its past versions referenced unittest only.
3) *unittest* → *unittest*: the present and past versions of the system reference unittest only.

We start by assessing the *present* version and then we assess its *past* versions in descending order until we find the proper classification. It is worth mentioning that the opposite change may happen: a system may start with pytest and change to unittest; we also keep track of those cases.

In unittest, we search for occurrences of the term *unittest* in source code, which may happen, for example, in test cases (*i.e.,* `unittest.TestCase`), importing (*e.g.,* `import unittest`), test skip (*e.g.,* `@unittest.skip`), and expected failure (*e.g.,* `@unittest.expectedFailure`).

In pytest, we search for occurrences of the term *pytest* in source code, which may happen in pytest fixtures (*e.g.,* `@pytest.fixture`), test skip (*e.g.,* `@pytest.mark.skip`), and expected failure (*e.g.,* `@pytest.mark.xfail`). In addition, a pytest test suite may have no direct reference to the term *pytest*. For example, the migration commit presented in Figure 2 changes the test from unittest to pytest, but the new source code (right-side) has no explicit reference to the term *pytest*.[11] Indeed, this is one of the advantages of pytest in which developers can build test suites by simply creating test functions with the `assert` keyword.

To overcome this issue in which the source code does not clearly indicate that the system relies on pytest, we also mine the files provided by CI/CD tools. Specifically, we consider the configuration file formats of the most popular CI/CD tools (*i.e.,* Travis, CircleCI, and GitHub Actions) and look for references to *pytest* in their configuration files. For example, considering the aforementioned case of Figure 2, we can detect that pytest is being adopted by verifying its *.travis.yml* file as it has pytest in the pipeline, *i.e.,* `poetry run pytest`.[12]

Evaluation 1. We evaluate the precision of the proposed method to correctly detect the testing frameworks. First, we selected 10 Python systems[13] and manually classified them according to the usage of the testing frameworks. For this purpose, the authors of the paper manually inspected their source code, commits, issues, and documentation. Next, we run our method on those 10 systems and contrast the automated classification with the manual one. As summarized in Table I, our method correctly detected the testing frameworks in past and presents versions, achieving a precision of 100%.

TABLE I: Evaluation: method to detect framework usage.

Migration Category	Classification		Precision
	Manual	Automated	
unittest → *unittest*	4	4	100%
unittest → *unittest & pytest*	3	3	100%
unittest → *pytest*	3	3	100%
All	10	10	100%

D. Assessing Testing Framework Migration

In the previous section, we presented a method to detect the testing frameworks used by a software system over time. It is the first step towards better understanding the usage of the testing frameworks, however, it is not enough to infer that the migration is happening. That is, the fact a system relies on both unittest and pytest in the present version does not necessarily mean that the system is migrating. It may represent, for example, a system that purposely relies on both testing frameworks (since this is a pytest feature [2]). Thus, for more precise migration analysis, we propose a method to detect systems that are migrating or are migrated from unittest to pytest. For this purpose, we need to analyze their commits and look for explicit migration evidence.

We consider that a system migrated (or is migrating) when it has at least one migration commit. We define a *migration commit* as a commit that explicitly migrates code from unittest to pytest. Examples of migration commits are presented in Figures 2 and 3. In those commits, the developers are deliberately changing code from unittest to pytest. Thus, migration commits allow us to discover the current *migration stage* of the system: not migrated, ongoing migration, or migrated.

Figure 4 illustrates the three migration stages and how they relate to the migration commits. The orange circles represent *migration commits*, whereas the black ones represent *normal commits* (*i.e.,* not migration commits). A migration starts when

[11]Code just after the migration: https://git.io/JcSQn

[12]CI/CD file with pytest: https://git.io/JcS7E

[13]These 10 systems are not part of the top-100 systems mined in this paper. We selected smaller real-world systems, considering commit count, so it could be possible to calculate precision and recall manually. The 10 systems are also presented in our dataset.

we detect the first migration commit (commit *m1*) and it ends when we detect the last migration commit (commit *mn*) and the system does not rely anymore on unittest. As pytest allows incremental migration [2], several migration commits may exist in a software system between the first and the last migration commits (*e.g.,* commits *m2* and *m3*), but not all commits in the *ongoing* stage are necessarily migration commits, *i.e.,* they can be normal commits (*e.g.,* commit *x*).

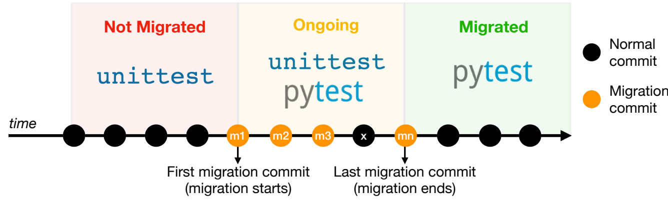

Fig. 4: Overview of the migration stages.

To detect migration commits from unittest to pytest, we assess the version history of the software systems. We rely on the PyDriller [14] mining tool to perform the historical analysis. Specifically, for each system, we iterate on its commits and verify the *removed* and *added* lines of code per commit. Based on the unittest and pytest API references [1], [2], we derive the following transformations to detect a migration commit. A commit is a *migration commit* if at least one of the following transformation rules hold and a *normal commit* otherwise:

1) **Assert migration**: the commit removes unittest `self.assert*` and adds `assert` keyword.
2) **Fixture migration**: the commit removes unittest fixtures (*i.e.,* `setUp`, `setUpClass`, `setUpModule`, `tearDown`, `tearDownClass`, and `tearDownModule`) and adds pytest fixtures (*i.e.,* `@pytest.fixture` and `@pytest.mark.usefixtures`).
3) **Import migration**: the commit removes `import unittest` and adds `import pytest`.
4) **Skip migration**: the commit removes unittest test skips (*i.e.,* `@unittest.skip`, `@unittest.skipIf`, and `@unittest.skipUnless`) and adds pytest test skips (*i.e.,* `@pytest.mark.skip` and `@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`)

One or more transformations may happen in a single migration commit. Figure 2 presents an example with *assert* migrations, while Figure 3 has *assert* and *fixture* migrations. Project Gaphor, for example, has a migration commit[14] with three transformations: *assert*, *fixture*, and *import* migrations.

Evaluation 2. To evaluate the proposed method in detecting migration commits, we compute its precision and recall. We rely on the same 10 Python systems used in Evaluation 1. We then run the proposed method and manually analyze their migration commits. Specifically, for each detected migration commit, the authors of the paper carefully manually inspected all code changes and verified whether it was a real migration commit from unittest to pytest.

Next, we compute the *precision* of the method in correctly detecting migration commits, looking for true positives (TP) and false positives (FP); *precision* = TP/(TP+FP). Computing the *recall* is more challenging because we need to assess all commits of a system to verify whether the proposed method is possibly missing any real migration commit, *i.e.,* the false negatives (FN). Thus, to strengthen this evaluation, we performed two recall analyses. In the first recall analysis, we randomly selected and manually inspected a sample of 366 out of all 7,758 commits (95% confidence level, 5% confidence interval). In the second recall analysis, we selected three systems, one in each migration stage, and manually assessed their 728 commits. In both cases, we assessed true positives (TP) and false negatives (FN); *recall* = TP/(TP+FN).

Table II summarizes this evaluation. We inspected 19 migration commits, leading to a *precision* of 100%. For recall, we inspected 366 commits in the first analysis and 728 commits in the second one, both achieving a *recall* of 100%.

TABLE II: Evaluation: method to detect migration commits.

Evaluation	Commits	TP	FP	FN	Value
Precision	19	19	0	-	100%
Recall (Analysis 1)	366	1	-	0	100%
Recall (Analysis 2)	728	4	-	0	100%

E. Research Questions

1) RQ0 (extension): In this motivational research question, we assess the usage of unittest and pytest over time. For this purpose, we detect the testing frameworks used in past and present versions, as detailed in Section III-C. **Rationale:** We aim to provide the first empirical evidence that the investigated phenomenon is somehow frequent in the Python ecosystem. We explore whether systems are likely to mix and exchange unittest and pytest over time.

2) RQ1 (frequency): Next, we assess the migration frequency. For this purpose, we analyze all commits of the selected projects looking for *migration commits* and assessing the *migration stages*, as detailed in Section III-D. **Rationale:** We aim to explore whether the studied systems have fully migrated or are still migrating. This may shed light on the facility or difficulty to perform the migration.

3) RQ2 (duration): In this RQ, we analyze the migration duration. We only focus on the systems that have already completed the migration (*i.e.,* the migrated systems) and assess their migration starting and ending date. Specifically, we report the *migration duration* in number of days as well as the *migration density*, as described in Table III. We classify the migration duration according to its speed (slow and fast) and the migration density according to its scattering (dense and sparse). **Rationale:** It is not clear whether developers tend to

[14]Gaphor migration commit: https://git.io/JcMX6

perform the migration on a single shot or over a long period. A short duration may suggest that the migration is somehow manageable, thus, this can encourage other projects facing the migration dilemma. In contrast, a longer duration may warn that the community is struggling to migrate, thus, projects considering the migration should be aware of the effort.

TABLE III: Summary of the duration metrics.

Metric	Short Description
Migration duration	Number of days between the first and last migration commits (*i.e., m1* and *mn* in Fgure 4)
Migration density	Ratio of migration commits during the migration (*i.e., migration commits / total commits during the migration duration*)

4) RQ3 (transformations): We assess the five transformations that happen in the migration commits, *i.e.,* assert, fixture, import, skip, and failure (see Section III-D). We also capture the size of the transformations in the number of deleted and added lines. **Rationale:** Our goal is to discover what are the most common transformations and the changing size, so we can gauge the maintainability of the changed code.

5) RQ4 (reasons): Finally, we investigate *why* developers migrate. We first tried to find explanations in the migration commits of the studied projects, however, we only found a limited amount. To overcome this limitation, we rely on the GitHub Search API to find other candidate sources. Specifically, we queried for the term "*unittest to pytest*" on *issues* and *pull requests* and manually inspected the first 100 results. Next, to filter out false positive results, we only selected the issues/PRs whose titles had some indication of migration. Moreover, we only kept the issues/PRs that mentioned some advantages and/or disadvantages, for example, the Jinja issue #424 ("*Consider Switching to Pytest*"). Finally, after this filtering step, we selected 61 issues/PRs.

We adopted thematic analysis to classify the explanations of the issues/PRs, with the following steps [15]: (1) initial reading of the issues/PRs, (2) generating a first code for each explanation, (3) searching for themes among the proposed codes, (4) reviewing the themes to find opportunities for merging, and (5) defining and naming the final themes. The first three steps were performed by the first author of the paper, while steps 4 and 5 were done together by all authors of the paper until consensus was achieved. **Rationale:** We aim to understand the reasons developers take into account to perform the migration. We also aim to reveal possible points against the migration. Both advantages and disadvantages may support other developers who are planning to migrate their projects.

IV. Results

A. RQ0 (extension): To what extent are unittest and pytest adopted in the Python ecosystem over time?

Table IV summarizes the testing frameworks used by the 100 studied systems in present and past versions. In the present version, 20 systems rely on unittest, 23 on pytest, and 34 rely on both unittest and pytest. It is also possible to observe how systems exchanged the testing frameworks over time. For example, 59 systems relied on unittest in the past, but in the present there are only 20, which represents a large reduction in usage. Interestingly, considering the 59 systems that relied on unittest in the past, 27 have now both unittest and pytest and 12 rely on pytest. This shows that over 66% (39 out of 59) of the systems initially with unittest now rely on pytest. In contrast, from the 18 systems that relied on pytest in the past, 7 now rely on both frameworks and none on unittest.

TABLE IV: Summary of testing framework usage over time.

		Present			
		unittest	pytest	both	Total
Past	unittest	20	12	27	59
	pytest	0	11	7	18
	Total	20	23	34	77

Notice that 31 out of 100 systems have not changed the testing frameworks over time (20 kept unittest and 11 kept pytest). Finally, 23 (*i.e.,* 100 – 77) systems do not fall in any category because they do not rely on unittest nor pytest.

Next, Table V details the number of files in the present version of the 39 (12+27) systems that started with unittest and adopted pytest. Overall, those systems have 1,771 files with unittest references and 3,410 files with pytest references; among those, there are 493 files referencing both frameworks. This means that 27% (*i.e.,* 493/1,171) of the present unittest files mix both unittest and pytest code. When considering the category *unittest → pytest*, on the median, each system has 36 pytest files. In the category *unittest → unittest & pytest*, on the median, each system has 36 unittest files, 17 pytest files, and 1 file with both testing frameworks.

TABLE V: Number of files with framework references in the last version. Between parentheses: the median per system.

Testing Framework	#Files in the Present		
	unittest	**pytest**	**both**
unittest → pytest	0	1,530 (36)	0
unittest → unittest & pytest	1,771 (36)	1,880 (17)	493 (1)
Total	1,771	3,410	493

RQ0 Conclusion. Most systems (77 out of 100) rely on unittest (20%), pytest (23%), or both (34%). Moreover, projects with unittest are moving to pytest: 66% (39 out of 59) of the systems initially with unittest now rely on pytest.

B. RQ1 (frequency): How frequent is code migrated from unittest to pytest?

We further explore the 39 systems that started with unittest and adopted pytest over time. We recall that a system is migrated (or is migrating) only when it has at least one *migration commit*, that is, a commit that explicitly migrates code from unittest to pytest.

From those 39 systems, 27 have at least one migration commit, while 12 have no migration commit. Regarding the 12 systems without migration commit, despite having unittest references in the past and pytest references in the present, they are not performing a migration; they are simply using both.

Table VI details the 27 systems with migration commits. They have a total of 330 migration commits; on the median, each system has 4 migration commits. Also, 11 systems are migrated, while 16 are still migrating.

TABLE VI: Summary of the systems with migration commits.

Migration Stage	#Systems	Migration Commits				
		#	median	max	min	σ
Migrated	11	102	2	53	1	15
Ongoing	16	228	4	88	1	22
All	27	330	4	-	-	19

The top-5 systems with the most migration commits are presented in Table VII. Aiohttp is the top-1, with 88 migration commits; notice that its migration is ongoing. Next, we have the projects Cookiecutter (53 migration commits), Apache Airflow (34), Ansible (33), and Pandas (26).

TABLE VII: Top-5 systems with most migration commits.

Pos	System	Migration Commits	Migration Stage	Examples
1	Aiohttp	88	Ongoing	7698ee
2	Cookiecutter	53	Migrated	2e7ea5
3	Apache Airflow	36	Ongoing	58c354
4	Ansible	33	Ongoing	aa7bd8
5	Pandas	23	Migrated	e303e2

RQ1 Conclusion. From the 39 systems that started with unittest and adopted pytest over time, 28% (11) have fully migrated to pytest, 41% (16) are still migrating, and 31% (12) did not migrate.

C. RQ2 (duration): How long does it take to migrate from unittest to pytest?

In this RQ, we focus on understanding the migrated systems. First, we explore the migration *duration*, that is, the period between the first and last migration commit (Table VIII). For simplicity, we flag as *fast* a migration concluded in up to one week and as *slow* a migration longer than one week. We find 6 out of 11 migrated systems with a *fast* migration; 4 systems have only one migration commit. Also, 5 out of migrated 11 systems have a *slow* migration. In this case, Pandas holds the longest migration: 23 migration commits during 2,326 days. Cookiecutter, which has the largest number of migration commits (53), took 132 days to migrate.

In addition, Table VIII also presents the migration *density*, that is, the ratio between the number of migration commits and the total number of commits during migration duration. We flag as *dense* the ratios greater than 1/10 (10%) and as *sparse* otherwise. We find 8 *dense* and 3 *sparse* migrations. For example, Cookiecutter has 53 migration commits and a total of 258 commits during the migration, leading to a dense migration of 20.54%, *i.e.,* 1/5 of the commits during the migration are *migration commits*. In contrast, Pandas has 23 migration commits and a total of 15,108 commits during the migration, leading to sparse migration of 0.15%. Overall, the migration may be fast or take a long period to be concluded, but most of the systems tend to concentrate it.

TABLE VIII: Summary of the migration duration and density.

System	Migration Commits	Duration (in days)	Density (Mig. Commits / Commits in Mig. Duration)
Celery	1	fast (1)	dense (100%)
Python-tel.	1	fast (1)	dense (100%)
Pytorch Geo.	1	fast (1)	dense (100%)
Routersploit	1	fast (1)	dense (100%)
Redis-py	6	fast (3)	dense (50%)
Flask	3	fast (5)	dense (11.11%)
Cookiecutter	53	slow (132)	dense (20.54%)
Saleor	2	slow (186)	dense (16.67%)
Allennlp	2	slow (514)	sparse (0.24%)
Requests	9	slow (1,553)	sparse (0.36%)
Pandas	23	slow (2,326)	sparse (0.15%)

Next, we assess *when* the migration has started in each system (Table IX). For example, in Celery, the first migration commit happened in commit number 8,880 out of 10,737 (82.7%), that is, the migration has started late in this project. In contrast, in Saleor, the first migration commit happened in commit number 992 out of 15,370 (6.4%), meaning that the migration has started early in this project. Overall, we observe that the migration may start in distinct development stages, from early periods (when the project is in initial steps) to late ones (when the project is possibly more mature).

TABLE IX: When the migration has started.

System	1st Migration Commit	Total Commits	Ratio (1st Migration Commit / Total Commits)
Celery	8,880	10,737	82.7%
Routersploit	571	699	81.7%
Python-tel.	1,418	1,983	71.5%
Allennlp	1,793	2,577	69.6%
Flask	1,577	3,170	49.7%
Pandas	8,818	24,505	36.0%
Redis-py	455	1,324	34.4%
Cookiecutter	556	2,198	25.3%
Pytorch Geo.	923	4519	20.4%
Requests	381	4542	8.4%
Saleor	992	15,370	6.4%

RQ2 Conclusion. The migration may be fast (up to one week) or take a long period to be concluded, from months to years. Most migrated systems (8 out of 11) tend to concentrate the migration commits, while only 3 perform the migration more sparsely. Systems start the migration in distinct development stages, from early to late ones.

D. RQ3 (transformations): What code is migrated from unittest to pytest?

We now explore the five transformations (*i.e.,* assert, fixture, import, skip, and failure migration) that happen in the migrated systems. Table X summarizes the frequency of the transformations. We observe that *assert* migrations are the most common, happening in 92 out of 102 migration commits (90%). Other common transformations are *fixture* (19 commits) and *import* (14 commits) migrations. Lastly, *skip* and *failure* migrations seem to be rarely performed. Overall, the prevalence of assert migrations is somehow expected because test cases should have at least one assert statement but not necessarily it will be skipped or failed. Moreover, the flexibility of the pytest fixtures is one of its advantages when compared to unittest [2], thus it may explain its high frequency of fixture migrations.

TABLE X: Frequency of the transformations.

Migration	#	%	Examples
Assert	92	90	60a926, 6a69aa, 2edda2
Fixture	19	18	1c6507, 34c6bd, 6a69aa
Import	14	13	be42d5, 533bc4, 6a69aa
Skip	2	2	2cfa8d, 61a243
Failure	0	0	-

Next, we present the size of the transformations calculated from the API references encountered, as summarized in Table XI. Overall, the migrated test code tends to decrease after the migration: we find 7,965 unittest deletions and 5,252 pytest additions, a reduction of 34% after the migration. In the case of the *assert* migration, there are 7,344 unittest `self.assert*` deletions and 4,714 native `assert` additions (a delta of -36%). Interestingly, in this case, we do not see a one-to-one transformation. The negative difference might mean that the developers also performed some kind of refactoring along with the migration, leading to fewer assert commands. The same negative difference happens to the *fixture* (-40%) and *skip* (-26%) migrations. Regarding the fixture size reduction, one possible explanation is that fixtures can be reused in pytest, which avoids duplication and produces less code. Finally, the *import* migration has a positive delta. We inspected the commits that applied this migration and noticed they mostly included new test files that added `import pytest`. These new files are created due to refactoring or remodularization of existing tests.

TABLE XI: Size of the transformations.

Migration	Size			
	Unittest Deletions	Pytest Additions	Δ	%Δ
Assert	7,344	4,714	-2,630	-36
Fixture	341	204	-137	-40
Import	97	202	+105	+108
Skip	188	138	-50	-26
Failure	0	0	0	0
Total	7,970	5,258	-2,712	-34

RQ3 Conclusion. The majority (90%) of the migration commits include assert migrations. Developers also tend to migrate fixtures (18%) and imports (13%). Overall, the migrated test code is 34% smaller than the original one, meaning fewer test code to be maintained.

E. RQ4 (reasons): Why is code migrated from unittest to pytest?

In the previous RQs, we explored several quantitative aspects of the migration from unittest to pytest. To better understand the rationales behind the migration, we now assess the explanations provided by the developers themselves. Table XII presents the reasons in favor and against the migration according to the 61 analyzed issues and pull requests. In summary, we find 9 advantages and 4 disadvantages.

TABLE XII: Migration advantages and disadvantages.

Advantages	#	Disadvantages	#
Easier syntax	12	Implicit mechanics	3
Interoperability	11	New tool to learn	2
Easier maintenance	8	Multiple test styles	2
Fixture flexibility/reuse	8	Migration duration	2
Built-in features	7	Other	5
Popularity	7		
Cleaner reports	6		
Parametrized tests	6		
Plugin integration	6		
Other	16		

Advantages. The most common advantage is *easier syntax* (12), meaning that the pytest syntax is easier than the unittest one. In PyCap, the developer states: "*[...] the pytest syntax is nicer, and allows us to take advantage of things like fixtures*".[15] In project Treon, the developer comments: "*[...] this would be a nice addition and removes a lot of boilerplate one needs for the unittest framework; syntax is easier to learn*".[16]

The second most frequent advantage is *interoperability* (11). Here, developers mention the fact that pytest also runs unittest as a strength. In Compliance Checker, the developer declares as a positive aspect: "*we can run the legacy unittests*".[17]

Next, we have the categories *easier maintenance* and *fixture flexibility/reuse*, both with 8 occurrences. In project Metron, the developer mentions regarding maintainability: "*It would be nice to migrate the test over to pytest-django to get rid of most of the boilerplate code*".[18] In Cookiecutter, it is highlighted the fixture advantages: "*There is a powerful fixtures system to support cleaner setup/teardown code, which supports per-test, per-class, per-module and global fixtures*".[19]

Developers also present many other reasons to migrate to pytest, including *built-in features*, *popularity*, *cleaner reports*, *parametrized tests*, and *plugin integration*. For instance, in

[15] https://git.io/JKiPm
[16] https://git.io/JKiXV
[17] https://git.io/JiZt7
[18] https://git.io/JKid1
[19] https://git.io/JK6rv

Cookiecutter, the developer also highlights the better reporting system: "*The reporting of test results is (IMO) cleaner than unittest, with a better summary, colour-coded output, and more detailed reporting of failures*". In Jinja, the developer mentions the plugins: "*[pytest] has over 100 plugins for easy integration with many frameworks, editors and CI servers*".[20]

Disadvantages. When discussing the migration, developers sometimes list possible disadvantages. The most common disadvantage is *implicit mechanics* (3), that is, the fact that pytest may perform the functionalities implicitly. In this context, the term "magic" is used twice in the analyzed issues. For example, in Cookiecutter, the developer says: "*There's a lot of 'magic' involved in the internals, which can be confusing*". Similarly, in project FermiLib, the developer mentions: "*Be careful with the 'magic': in particular, fixtures can sometimes be overused in ways that make test code hard to follow because too much is happening behind the scenes*".[21]

Finally, other disadvantages are *new tool to learn*, *multiple test styles*, and *migration duration*. For instance, one developer lists some negative aspects: "*Another tool to learn for contributors [...] Either we end up with multiple styles of test or there's a lot of work in rewriting existing tests*".

RQ4 Conclusion. Developers migrate from unittest to pytest mostly due to the easier syntax, interoperability, easier maintenance, and fixture flexibility/reuse. Disadvantages are less discussed, but the implicit mechanics of pytest is the main concern.

V. Discussion and Implications

A. For Practitioners

Migration practices, advantages, and disadvantages. We provide empirical evidence that the migration from unittest to pytest is common in the Python ecosystem. First, we find that 34% of the studied projects rely on both testing frameworks (RQ0). While some systems have fully migrated (11), others are still migrating (16), suggesting that the migration is not always straightforward (RQ1). Most projects (8 out of 11) that fully migrated concentrate the migration commits and start the migration in distinct development phases (RQ2). Overall, the migrated test code is 34% smaller than the original one, suggesting that there is less test code to be maintained (RQ3). Moreover, developers focus on discussing the advantages of the migration, like *easier syntax* and *fixture flexibility/reuse*, however, some disadvantages are highlighted, like *implicit mechanics* and *multiple test styles* (RQ4). This way, we reveal a set of practices and reasons in favor and against migration in the wild. Practitioners in charge of the migration should be aware that: **(i)** the migration may be complex and take time to conclude; **(ii)** projects that migrated successively kick-started it in different development phases, but tended to concentrate the migration commits; **(iii)** the migration may lead to less code to maintain; and **(iv)** the community is more in favor of migration, however, the points against are not negligible.

Keep track of the migration. Our results show that 16% of the 100 studied systems have started the migration but not concluded it (RQ1). This may suggest that those projects are somehow stuck, for example, due to the migration complexity or the lack of contributors. Indeed, as presented in RQ2, the migration may be distributed in dozens of migration commits, taking months or years to be concluded. We hypothesize that due to the size and complexity of some projects, it is easy to get lost during the migration. For example, when performing RQ4, we found almost no information in commit messages regarding the migration. This raises the following question: *how are developers keeping track of the migration?* As a first step, we looked for issues related to migration commits, but we found only three linked issues. The possible explanation is that teams track migrations with external management tools and communication platforms, or they are simply not tracked. Thus, we shed light on this migration challenge that is likely to happen in open-source projects. One simple solution to overcome this problem is to keep track of the migration tasks, for example, managing the migration via issues (*e.g.,* tagged with migration-related labels) and linking them to commits.

Improve migration guidelines. The lack of contributors may justify the delay to complete the migration. A common way to attract newcomers is via *good first issues* [16]. In the dataset of RQ4, for example, 12 issues are marked with the label "*help wanted*" and 5 as "*good first issue*", suggesting they are ideal for novel contributors. However, out of the 61 studied issues in RQ4, only 22 mentioned concrete steps to perform the migration. In Pandas, for instance, the issue #15990 presents basic steps to migrate, including functions that should be removed and replaced. The same happens in the issue #1502 of Opsdroid, which depicts migration steps like "*change assertions to use regular assert or pytest assertions*". Thus, to attract contributors, we recommend that migration issues should be created with detailed migration guidelines, like the ones of Pandas and Opsdroid.

B. For Researchers

Why the migration is not concluded. We found that dozens of popular systems started but did not conclude the migration (RQ1). During the period in which the migration is ongoing, the test suite may become even more complex as two testing frameworks are used interchangeably. Indeed, we detected that even the same test file may mix both unittest and pytest (RQ0). This raises the following question that can be addressed by further research: *why are some migrations started but not completed?* In RQ4, we provide some initial insights on this direction, for example, developers mention that the migration requires a *new tool to learn* and they complain about the *migration duration* and the possibility to have *multiple test styles* in the test suite, which may discourage, delay, or even suspend the migration. However, further studies should be performed with practitioners to better understand *why* some migration are *not* concluded, even after years.

[20] https://git.io/JKi7L
[21] https://git.io/JKiEc

Tools and techniques to document and automate the migration. To alleviate migration delay, novel tools can be proposed to detect and document unittest code that is not migrated yet (in a similar way to project-specific lint rules [17]–[19]). In this case, the candidate code to be migrated can be, for example, automatically monitored and logged via CI/CD workflows. Moreover, in our analysis, we detected hundreds of migration commits from unittest to pytest (RQ1). That is, on the one side, there is a need to complete the migration, while, on the other side, there are migration examples (*i.e.,* the migration commits) to learn from. From this data, migration rules can be inferred in a similar way to migration updates [9], [20]–[25]. Therefore, we envision that novel tools and techniques can be proposed by researchers to automate the migration and reduce the migration delay.

VI. Threats to Validity

Framework usage and migration commits. The method to detect the testing framework usage in past and present versions were manually evaluated, leading to a precision of 100% (see Section III-C). Similarly, the method to detect migration commits was also manually evaluated (see Section III-D), achieving precision and recall of 100%. Moreover, to avoid being subjected to refactoring operations and noisy changes, we only considered a migration commit if it strictly has a unittest removal *and* a pytest addition. Thus, the high precision and high recall reduce the chance of false positives and false negatives in our results.

Mocking libraries. In our experiments, we do not take into account the mocking libraries of unittest (unittest.mock) and pytest (pytest-mock) because they can be seen as "external" projects and are not the core of a testing framework. Thus, the presence or absence of mocking libraries do not affect the detection of testing frameworks nor migration commits.

Manual classification of issues/PRs. In RQ4, we manually classify the explanations found in issues/PRs about the migration advantages and disadvantages. In this case, we rely on thematic analysis [15] to reduce the subjectiveness.

Generalization of the results. In this study, we mine 100 real-world Python systems. Those systems are among the most popular in Python, thus, they are relevant projects. Despite these observations, our findings—as usual in empirical software engineering studies—cannot be directly generalized to other Python systems, projects implemented in other programming languages, or closed-source systems. Further studies should be performed on other software ecosystems.

VII. Related Work

There is vast literature covering library and framework migration and evolution. Prior research investigates API migration/evolution in Java [8], [10], [26]–[32], Android [23]–[25], [33]–[36], JavaScript [32], [37]–[41], and Python [42], [43], to name a few. For example, McDonnell *et al.* [33] detected that Android APIs are evolving fast and that clients do not follow the pace of API evolution. Nascimento *et al.* [37] found that there is no standard solution to deprecate JavaScript APIs, while Wang *et al.* [42] detected that Python API deprecation is poorly handled by library contributors. Migration is also studied at higher levels, for example, the migration between programming languages (*e.g.,* Java to Kotlin [7] and Java to C# [9]), and between programming language versions (*e.g.,* from Python 2 to Python 3 [6]).

Zerouali and Mens [44] analyzed the usage of eight testing-related libraries in 4,532 open-source Java projects hosted. The authors observed that many projects tend to use multiple libraries together and there exist permanent and temporary migrations between competing libraries. Malloy and Power [6] investigated to which degree Python 2 systems had migrated to Python 3, as there is no native backward compatibility. The results indicated that developers are not exploiting the new features and advantages of Python 3 and confining themselves to a language subset to preserve backward compatibility. In a related research line, Martinez and Mateus [7] studied the migration phenomenon from Java to Kotlin in Android applications. Kotlin is interoperable with Java, thus, developers can choose to migrate gradually. The authors performed a quantitative approach, using a history-based tool to detect commits that migrated code, and a qualitative approach by interviewing developers to assess why they migrated. Overall, the authors found that the migration occurred to access features only available in Kotlin and to obtain safer code.

Despite the various studies on different types of migrations and ecosystems, there is a lack of research addressing the migration of testing frameworks. Therefore, we contribute to the literature on library and framework migration with a novel study on testing framework migration, particularly in the Python ecosystem.

VIII. Conclusion

We presented an empirical study to assess *how* and *why* developers migrate from unittest to pytest. We detected that 34% of the systems rely on both testing frameworks and that Python projects are moving to pytest. The migration may be fast or take a long period to be concluded and the migrated test code is smaller. Developers migrate to pytest due to several reasons, however, the implicit mechanics of pytest is a concern. Based on our findings, we discuss implications for both practitioners and researchers. For example, we shed light on migration practices, advantages, and disadvantages and on how to keep track of the migration. Finally, we discussed novel research directions to document and automate the migration.

In future work, we plan to better understand why some popular projects start but do not conclude the migration. We also plan to better understand who are the developers in charge of performing the migration, that is, more experienced or novel contributors. Finally, based on the migration commits, we also plan to propose tools and techniques to aid the migration.

Acknowledgment

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

References

[1] Unittest, https://docs.python.org/3/library/unittest.html, September, 2021.

[2] Pytest, https://docs.pytest.org, September, 2021.

[3] NumPy, https://numpy.org, September, 2021.

[4] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[5] M. Feathers, *Working Effectively with Legacy Code*. Prentice Hall Professional, 2004.

[6] B. Malloy and J. Power, "An empirical analysis of the transition from Python 2 to Python 3," in *Empirical Software Engineering*. Springer International Publishing, 2019, p. 751–778.

[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. Hora, R. Robbes, M. T. Valente, N. Anquetil, A. Etien, and S. Ducasse, "How do developers react to API evolution? a large-scale empirical study," *Software Quality Journal*, vol. 26, no. 1, pp. 161–191, 2018.

[9] 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.

[10] 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, p. 1458–1492, 2020.

[11] JUunit, https://junit.org/junit5, September, 2021.

[12] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of GitHub repositories," in *International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 334–344.

[13] H. Silva and M. T. Valente, "What's in a GitHub star? understanding repository starring practices in a social coding platform," *Journal of Systems and Software*, vol. 146, pp. 112–129, 2018.

[14] 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.

[15] D. S. Cruzes and T. Dyba, "Recommended steps for thematic synthesis in software engineering," in *International Symposium on Empirical Software Engineering and Measurement*, 2011, pp. 275–284.

[16] X. Tan, M. Zhou, and Z. Sun, "A first look at good first issues on GitHub," in *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 398–409.

[17] L. Renggli, S. Ducasse, T. Gîrba, and O. Nierstrasz, "Domain-specific program checking," in *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer, 2010, pp. 213–232.

[18] A. Hora, N. Anquetil, S. Ducasse, and S. Allier, "Domain specific warnings: Are they any better?" in *International Conference on Software Maintenance (ICSM)*, 2012, pp. 441–450.

[19] K. F. Tómasdóttir, M. Aniche, and A. Van Deursen, "The adoption of JavaScript linters in practice: A case study on ESLint," *IEEE Transactions on Software Engineering*, vol. 46, no. 8, pp. 863–891, 2018.

[20] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in *International Conference on Software Engineering (ICSE)*, 2010, pp. 325–334.

[21] S. Meng, X. Wang, L. Zhang, and H. Mei, "A history-based matching approach to identification of framework evolution," in *International Conference on Software Engineering (ICSE)*, 2012, pp. 353–363.

[22] M. Lamothe and W. Shang, "Exploring the use of automated api migrating techniques in practice: an experience report on android," in *International Conference on Mining Software Repositories (MSR)*, 2018, pp. 503–514.

[23] F. Thung, S. A. Haryono, L. Serrano, G. Muller, J. Lawall, D. Lo, and L. Jiang, "Automated Deprecated-API Usage Update for Android Apps: How Far Are We?" in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 602–611.

[24] S. A. Haryono, F. Thung, H. J. Kang, L. Serrano, G. Muller, J. Lawall, D. Lo, and L. Jiang, "Automatic Android deprecated-API usage update by learning from single updated example," in *International Conference on Program Comprehension*, 2020, pp. 401–405.

[25] M. Fazzini, Q. Xin, and A. Orso, "Automated API-usage update for Android apps," in *International Symposium on Software Testing and Analysis*, 2019, pp. 204–215.

[26] H. Alrubaye, M. W. Mkaouer, and A. Ouni, "Migrationminer: An automated detection tool of third-party java library migration at the method level," in *International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 414–417.

[27] S. Xu, Z. Dong, and N. Meng, "Meditor: Inference and application of api migration edits," in *International Conference on Program Comprehension (ICPC)*, 2019, pp. 335–346.

[28] H. Alrubaye, D. Alshoaibi, E. Alomar, M. W. Mkaouer, and A. Ouni, "How does library migration impact software quality and comprehension? an empirical study," in *Reuse in Emerging Software Engineering Practices*, S. Ben Sassi, S. Ducasse, and H. Mili, Eds. Cham: Springer International Publishing, 2020, pp. 245–260.

[29] A. Brito, L. Xavier, A. Hora, and M. T. Valente, "Why and how Java developers break APIs," in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 255–265.

[30] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018.

[31] A. A. Sawant, R. Robbes, and A. Bacchelli, "On the reaction to deprecation of clients of 4+ 1 popular java apis and the jdk," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2158–2197, 2018.

[32] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an api: cost negotiation and community values in three software ecosystems," in *International Symposium on Foundations of Software Engineering (FSE)*, 2016, pp. 109–120.

[33] T. McDonnell, B. Ray, and M. Kim, "An empirical study of API stability and adoption in the Android ecosystem," in *International Conference on Software Maintenance*, 2013, pp. 70–79.

[34] 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.

[35] G. Bavota, M. Linares-Vasquez, C. E. Bernal-Cardenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "The impact of API change-and fault-proneness on the user ratings of Android apps," *IEEE Transactions on Software Engineering*, vol. 41, no. 4, pp. 384–407, 2014.

[36] L. Li, T. F. Bissyandé, H. Wang, and J. Klein, "Cid: Automating the detection of api-related compatibility issues in android apps," in *International Symposium on Software Testing and Analysis*, 2018, pp. 153–163.

[37] R. Nascimento, E. Figueiredo, and A. Hora, "JavaScript API Deprecation Landscape: A Survey and Mining Study," *IEEE Software*, 2021.

[38] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *International Conference on Mining Software Repositories (MSR)*, 2018, pp. 181–191.

[39] A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. González-Barahona, "An empirical analysis of technical lag in npm package dependencies," in *International Conference on Software Reuse*. Springer, 2018, pp. 95–110.

[40] A. Decan, T. Mens, and E. Constantinou, "On the evolution of technical lag in the npm package dependency network," in *International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 404–414.

[41] R. E. Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto, and A. Ihara, "Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm JavaScript packages," in *International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 559–563.

[42] 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.

[43] Z. Zhang, H. Zhu, M. Wen, Y. Tao, Y. Liu, and Y. Xiong, "How do python framework apis evolve? an exploratory study," in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 81–92.

[44] A. Zerouali and T. Mens, "Analyzing the evolution of testing library usage in open source Java projects," in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 417–421.