UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Lívia Almeida Barbosa

Assessing the Migration of Testing Frameworks in the Python Ecosystem

Belo Horizonte
2022

Lívia Almeida Barbosa

Assessing the Migration of Testing Frameworks in the Python Ecosystem

Final Version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Andre Cavalcante Hora

Belo Horizonte
2022

Barbosa, Líivia Almeida.

B238a Assessing the migration of testing frameworks in the python ecosystem [manuscrito] / Líivia Almeida Barbosa. — 2022.
60 f. il.; 29 cm.

Orientador: André Cavalcante Hora.
Dissertação (mestrado) - Universidade Federal de Minas Gerais – Departamento de Ciência da Computação
Referências: f. 56-60.

1. Computação – Teses. 2. Software – Testes – Teses. 3. Mineração de dados (Computação) – Teses. 4. Engenharia de Software – Teses. 5. Python (Linguagem de programação de computador) – Teses. I. Hora, Andre Cavalcante. II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Computação. III. Título.

CDU 519.6*32 (043)

Ficha Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa CRB 6/1510 Universidade Federal de Minas Gerais - ICEx

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Assessing the Migration of Testing Frameworks in the Python Ecosystem

LÍVIA ALMEIDA BARBOSA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

André Cavalcante Hora

PROF. ANDRÉ CAVALCANTE HORA - Orientador
Departamento de Ciência da Computação - UFMG

PROF. MARCO TÚLIO DE OLIVEIRA VALENTE
Departamento de Ciência da Computação - UFMG

PROF. NICOLAS ANQUETIL
RMod - University of Lille

Belo Horizonte, 12 de julho de 2022.

Resumo

Atualmente, desenvolvedores Python podem utilizar dois frameworks de teste: unittest e pytest. Devido aos benefícios do pytest (tais como reuso de fixtures), muitos projetos relevantes no ecossistema Python migraram do unittest para o pytest. Apesar de ser realizada pela comunidade Python, ainda não temos informações suficientes sobre a migração de unittest para pytest, nem sobre as razões da migração. Neste estudo, analizamos *como* e *por qual motivo* desenvolvedores migram de unittest para pytest. Para isso, mineramos os 100 sistemas Python mais populares e analisamos seus status de migraçãos. Os resultados mostram que 34% dos sistemas dependem de ambos frameworks de teste e que projetos Python estão, de fato, realizando a migração para pytest. Enquanto alguns sistemas migraram completamente, outros ainda estão migrando após um longo período, sugerindo que a migração não é sempre direta. Em geral, o código de teste migrado é menor que o original. Além disso, desenvolvedores migram para pytest por diversas razões, como sintaxe simplificada, interoperabilidade, manutenção facilitada e flexibilidade/reuso de fixtures, entretanto, a mecânica implícita do pytest e o fato de ser um pacote não nativo são preocupações relevantes.

Palavras-chave: Teste de Software, Manutenção de Software, Mineração de Repositórios de Software, engenharia de Software Empírica, Python, Pytest, Unittest.

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. This study provides 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 and the fact that it is a separated package are concerns.

Keywords: Software Testing, Software Maintenance, Mining Software Repositories, Empirical Software Engineering, Python, Pytest, Unittest.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Motivation

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 [40] and pytest [32]. Many projects rely on one or even both frameworks. For example, F Prime,[1] an open-source software framework created by NASA for development and deployment of embedded systems and spaceflight applications, currently depends on pytest. Django,[2] a widely used web framework, relies on unittest. Numpy,[3] a popular scientific computing tool, uses both framework in its tests.

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, simpler assertions and built-in parameterized tests [32]. Due to such benefits, several projects have migrated from unittest to pytest. For example, in project Opsdroid[4] (a popular chatbot framework), there is an explicit call for this migration:[5]

[1] https://github.com/nasa/fprime
[2] https://github.com/django/django
[3] https://github.com/numpy/numpy
[4] https://opsdroid.dev
[5] https://github.com/opsdroid/opsdroid/issues/1502

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.

While some projects have this call to initiate migration, others have invitations to conclude it. For example, in projects owned by the Home Assistant organization,[6] used for home automation, there are some open issues asking to conclude the migration:[7]

Rewrite graphite unittest tests to pytest style test functions #40850

The Home Assistant core standard is to write tests as standalone pytest test functions. We still have some old tests that are based on unittest.TestCase. We want all these tests to be rewritten as pytest test functions.

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 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, results 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[8] — applications. Thus, having proper automated testing is fundamental to ensure quality and sustainable evolution [5, 14]. While API

[6] https://github.com/home-assistant

[7] https://github.com/home-assistant/core/issues/40850

[8] For example, NumPy was used to support Black Hole imaging and the detection of gravitational waves [31].

migration is a research topic broadly studied by prior literature (*e.g.,* [25, 27, 18, 48, 8]), to the best of our knowledge, the migration of *testing* frameworks have never been deeply explored by the research community.

1.2 Proposed Work

This master dissertation provides an empirical study to assess testing framework migration. We analyze *how* and *why* developers migrate Python tests from unittest to pytest. First, we propose a quantitative study to understand *how* developers migrate. For this purpose, we mine 100 popular Python projects and analyze their migration status. We provide research questions to assess the migration extension, frequency, duration, transformations, as follows:

- *RQ1 (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.
- *RQ2 (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.
- *RQ3 (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.
- *RQ4 (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.

Second, we propose a qualitative study to explore *why* developers migrate. In this analysis, we assess migration explanations in issues and pull requests from GitHub and web resources from a Grey Literature Review (*e.g.,* blog posts, Q&A forums, and documentation) to understand migration rationales. We also perform a short survey with developers to better understand their view of the migration process. We provide research questions to assess the migration reasons, guidelines, and process, as follows:

- *RQ5 (reasons): Why is code migrated from unittest to pytest?* Developers migrate to pytest mostly due to the easier syntax, interoperability, and fixture flexibility and reuse. In contrast, the implicit mechanics of pytest and being a separated package are the main concerns.
- *RQ6 (guidelines): How common are guidelines to support the migration and what is their content?* We find that 40% of resources have code examples and tips to help perform the migration.
- *RQ7 (process): What is the migrator's view on the process?* We find that the size of the codebase and manual effort are the main concerns.

In summary, we provide empirical evidence that test migration is common in the Python ecosystem. We find that over 1/3 of the studied systems rely on both testing frameworks. The migration delay suggests that performing it is not always trivial. Overall, the migrated test code is smaller than the original one, meaning there is less test code to be maintained. Furthermore, developers focus on discussing the advantages of the migration, but some disadvantages are also highlighted.

1.3 Contributions

The contributions of this master dissertation are threefold: (i) it provides the first empirical assess the migration of *testing* frameworks; (ii) it presents how code is migrated and the reasons for the migration; and (iii) it proposes practical implications for practitioners and researchers.

Based on findings, we propose 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, (v) the proposal of novel techniques to document and automate the migration, and (vi) the developer experience and the project follow up during and after the migration.

1.4 Publication

As a result this research, the following article has been published [3]. Therefore, this dissertation contains material from this article:

- Lívia Almeida, Andre Hora. How and Why Developers Migrate Python Tests. In *29th IEEE International Conference on Software Analysis, Evolution and Reengineering* (SANER), pages 1-11, 2022

1.5 Outline of the Dissertation

This master dissertation is organized as follows:

- **Chapter 2** introduces the unittest and pytest testing frameworks and describes the migration phenomenon in the Python ecosystem. It also presents the related work on API evolution and software migration.
- **Chapter 3** describes the study design to answer the proposed research questions on *how* and *why* developers migrate Python tests.
- **Chapter 4** presents the results of the quantitative study on how developers migrate Python tests(RQs 1-4). It also presents threats to validity related to these questions.
- **Chapter 5** presents the results of the qualitative study on on why developers migrate Python tests (RQs 5-7). Its also discusses threats to validity.
- **Chapter 6** discusses implications for practitioners and researchers.
- **Chapter 7** concludes this dissertation, presenting an overview, contributions, and suggestions for future work.

Chapter 2

Background

This chapter introduces both testing frameworks (Section 2.1), assesses how the migration is performed (Section 2.2), explores how it is relevant in the Python ecosystem (Section 2.3), and presents related work on software evolution and migration (Section 2.4).

2.1 Unittest and Pytest in a Nutshell

Unittest [40] and pytest [32] 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 [19] and has a similar flavor as major unit testing frameworks in other programming languages [40]. Pytest makes it easy to write small tests and scales to support complex functional testing [32].

Figures 2.1 and 2.2 present examples of tests methods written with unittest and pytest. Unittest examples in Figure 2.1 were taken from the official documentation and Figure 2.2 has the pytest version of the same test cases. Unittest relies on inheritance to create tests (*i.e.,* the test needs 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,[1] for example, pytest facilitates the creation of parameterized tests and the reuse of fixtures.

Overall, the community acknowledges some advantages of pytest. In project Ops-droid, a core developer states: "*Fixtures are more reusable and portable and should help reduce complexity all over*". The pytest documentation [32] mentions that there is "*No need to remember self.assert* names*". Reddit has multiple posts comparing both testing

[1]Summary of the major differences: https://git.io/Jnc1m

```python
import unittest

class TestStringMethods(unittest.TestCase):
    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        with self.assertRaises(TypeError):
            s.split(2)
```

(a) Simple unittest tests.

```python
import unittest

class MyWidgetTestCase(unittest.TestCase):
  def setUp(self):
    self.widget = MyWidget('The widget')

  def test_default_widget_size(self):
    self.assertEqual(self.widget.size(), (50,50))

  def test_widget_resize(self):
    self.widget.resize((100,150))
    self.assertEqual(self.widget.size(), (100,150))
```

(b) Unittest test with setup.

Figure 2.1: Unittest examples.

```python
import pytest

def test_upper():
    assert 'foo'.upper() == 'FOO'

def test_isupper():
    assert 'FOO'.isupper()
    assert not 'Foo'.isupper()

def test_split():
    s = 'hello world'
    assert s.split() == ['hello', 'world']
    with pytest.raises(TypeError):
        s.split(2)
```

(a) Simple pytest tests.

```python
import pytest

@pytest.fixture
def my_widget():
  widget = MyWidget('The widget')
  return widget

def test_default_widget_size(my_widget):
  assert my_widget.size() == (50,50)

def test_widget_resize(my_widget):
  my_widget.resize((100,150))
  assert my_widget.size() == (100,150)
```

(b) Pytest test with fixtures.

Figure 2.2: Pytest examples.

frameworks,[2] *e.g.*, "*Tests are shorter, easier to read, with more reusability and extensibility, and have better output*", "*Parameterising allows you to run the same test in multiple configurations*", and "*One of the greatest strengths of pytest is that it makes much better use of Python than unittest does: it very nicely leverages free functions, decorators, context managers*".

2.2 Migrating from Unittest to Pytest

Despite unittest being a built-in Python package, the Python community seems to be moving to pytest. The fact that pytest can run unittest tests may facilitate the

[2] https://bit.ly/3cMPit7, https://bit.ly/3zyEEzB, and https://bit.ly/2SIdYvL

migration. Migrating from unittest to pytest would involve at least the following steps: (1) removing tests from class and moving to regular functions; (2) replacing assertions with regular Python asserts; and (3) moving setup operations to fixtures. Figure 2.3a presents a migration in project TermGraph,[3] 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.

unittest

```
- from unittest import TestCase
  from termgraph import CandleStickGraph, Candle


- class CandleTests(TestCase):
-     def test_candle_int_4(self):
-         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]]

-         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])
```

pytest

```
  from termgraph import CandleStickGraph, Candle


+ 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]]

+     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]
```

(a) Class and assertion migration (TermGraph, hash: d56652).

unittest

```
- class BufferTest(unittest.TestCase):
-     def setUp(self):
-         b = Buffer()
-         eb = EditorBuffer('b1', b)
-         self.window = Window(eb)
-         self.tabpage = TabPage(self.window)
```

pytest

```
+ @pytest.fixture
+ def prompt_buffer():
+     return Buffer()
+
+
+ @pytest.fixture
+ def editor_buffer(prompt_buffer):
+     return EditorBuffer(prompt_buffer, 'b1')
+
+
+ @pytest.fixture
+ def window(editor_buffer):
+     return Window(editor_buffer)
+
+
+ @pytest.fixture
+ def tab_page(window):
+     return TabPage(window)
```

unittest

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

pytest

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

(b) Fixture migration (pyvim, hash: 7e1c7b).

Figure 2.3: Real migration examples.

However, the migration is not always that straightforward. For example, Figure 2.3b presents a migration in Pyvim,[4] 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

[3] TermGraph migration commit: https://git.io/Jncy5
[4] pyvim migration commit: https://git.io/Jn8xh

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 [32]. Once pytest finds them, it runs those fixtures, captures what they returned, and passes those objects into the test function as arguments [32]. 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.

Sometimes, to perform the fixture migration, one needs to have deep knowledge of the context to properly migrate without loss of information. When the fixture is composed of multiple environment states it is necessary to understand the semantics before migrating the entire fixture or dividing it into smaller and reusable fixtures. On the other hand, the other migrations (*e.g.,* assert, skip, and expected failure) are syntactic, therefore, more straightforward and easier to perform.

2.3 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), Ansible (48K stars, automation platform), scikit-learn (machine learning module), pandas (data analysis library), and NumPy (scientific computing tool), to name a few. As those projects are the basis for a variety of applications, automated testing is fundamental to ensure software quality with sustainable evolution and maintenance [5, 14].

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*",[5] 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).

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

- from unittest to pytest: "*Need to stop using unittest's TestCase and use pytest instead*" (dvc, issue 1819).
- 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).
- unittest tests to pytest: "*This project started using pytest to run tests in #55. However, existing tests were left untouched, and still are written in unittest style*" (marshmallow_dataclass, issue 56).

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

To support the migration, some tools help the conversion from unittest test cases to pytest ones. We can find tools developed by the community but are not popular, [6] [7] with less than 10 stars. There is another one, unittest2pytest[8] that can be considered more popular (90 stars, 13 contributors), however with limited transformations available (`remove_class` and `self_assert`, from the documentation).

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.

2.4 Related Work

There is a vast literature covering library and framework migration and evolution. Migration is also studied at higher levels, for example, the migration between programming languages and between programming language versions. The following sections present relevant work related to test framework migration.

[6] https://github.com/dannysepler/pytestify
[7] https://github.com/hanswilw/codemod-unittest-to-pytest-asserts
[8] https://github.com/pytest-dev/unittest2pytest

2.4.1 API Evolution and Migration

Prior research investigates API migration/evolution in Java [2, 43, 1, 8, 9, 18, 21, 34, 6], Android [28, 24, 4, 23, 38, 16, 13], JavaScript [30, 12, 45, 11, 6, 44], and Python [41, 47], to name a few. Alrubaye *et al.* [2] introduced an open source tool that detects potential library migration code changes between Java third-party libraries. The tool also collects library documentation of methods involved in the migration. Its evaluation against manually validated migrations showed to be very effective.

There is also multiple studies regarding API evolution and deprecation. Hora *et al.* [18] perform a large empirical study on the Pharo ecosystem. Pharo is a pure object-oriented dynamically typed programming language. The authors wanted to understand how client systems react to API changes, how many are affected, and how long it takes for the changes to propagate. They analyzed 118 API changes and found that 53% impacted other systems and API changes and deprecation present different characteristics.

Sawant *et al.* [34] extend studies on API changes in the Pharo Smalltalk ecosystem by analyzing a dataset of more than 25,000 clients of five popular Java APIs and 60 clients of the JDK API. Some of their goals are to know how fast Android API changes, how dependent client code on Android APIs is, how long it takes to adopt new APIs, and investigate API stability. Among the findings, they report that, for all the APIs, only a minority of clients upgrade/change the API version they use and that "deletion" was the most popular way to react to a deprecated entity.

McDonnell *et al.* [28] performed a case study of the co-evolution behavior of Android API on GitHub projects. They detected that Android APIs are evolving fast, that fast-evolving APIs are more used by clients, but clients do not follow the pace of API evolution. They also found that the propagation time of newer versions is about 14 months, which is slower than the average of 3 months for API release interval.

Nascimento *et al.* [30] surveyed 109 JavaScript developers and a mining study on 320 popular JavaScript projects to investigate how developers deprecate APIs since Javascript does not have a native deprecation mechanism. Results suggest that there is no standard solution to deprecate JavaScript APIs and that developers resort to multiple solutions to achieve this goal.

Zhang *et al.* [41] explore the evolution of Python framework APIs and the compatibility issues in client applications. They analyzed 288 releases of six popular Python frameworks and 5,538 open-source projects that used them. Results showed that the API evolution in Python systems considerably differs from those of Java framework APIs. They also implemented a tool, PYCOMPAT, that effectively detects compatibility issues caused by the misuse of evolved framework APIs in Python applications.

Zerouali and Mens [46] analyzed the usage of eight testing-related libraries in 4,532

open-source Java projects hosted on GitHub. They analyzed specific pairs of libraries and identified if and when there was a replacement in a project's lifecycle. The authors observed that many projects tend to use multiple libraries together and even though some libraries are more popular, other libraries increase their popularity over time. Also, they noticed permanent and temporary migrations between competing libraries. There are key differences between Zerouali's and Mens' work and this master dissertation. Their testing-related libraries include mock libraries, they do not study in a fine-grained analysis the code transformation in the migration and they do not perform a qualitative investigation.

2.4.2 Programming Language Migration

Malloy and Power [25] investigated to which degree Python 2 systems had migrated to Python 3, as there is no native backward compatibility. They developed a compliance analyzer and ran it in a previously studied dataset of Python applications. Results indicated that developers are not exploiting the new features and advantages of Python 3 and confining themselves to a language subset, an intersection of the two versions, to preserve backward compatibility.

In a related research line, Martinez and Mateus [27] 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 used a history-based tool to detect commits that had migrated code in open-source Android projects. Aiming for a qualitative approach, they emailed and interviewed 98 developers that did the migration to assess the reasons why they migrated. Overall, the authors found that the migration occurred to access features only available in Kotlin and to obtain a 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 in Python applications. Therefore, this work contributes to the literature on library and framework migration with a novel study on testing framework migration, particularly in the Python ecosystem.

2.5 Final Remarks

This chapter presented an overview of unittest and pytest, which are the most popular testing frameworks in Python. We also explored how the migration from unittest to pytest can be performed. Finally, we presented the related work in the context of software evolution and migration.

Chapter 3

Study Design

In this chapter, we describe the study design to answer the proposed research questions. First, we provide an overview of the importance of the Python ecosystem (Section 3.1). We describe the design to answer research questions 1 to 4 (quantitative study) in Section 3.2 and to answer research questions 5 to 7 (qualitative study) in Section 3.3. We detail the research questions and their rationales in Section 3.4 and present final remarks in Section 3.5.

3.1 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[1] and TIOBE[2] 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, for instance, which are highly well-tested. *Third*, the testing landscape in Python is dominated by unittest [40] and pytest [32]; 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.

[1]GitHub ranking: https://git.io/JnOlr
[2]TIOBE ranking: https://www.tiobe.com/tiobe-index

3.2 Quantitative Study: How Developers Migrate

3.2.1 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 stars, which is a metric largely adopted in the software mining literature as a proxy of popularity [7, 35]. In this process, we took special care to filter out non-software projects, such as tutorials, examples, and samples, among others. The dataset under study includes broadly adopted systems, 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.

3.2.2 Detecting Testing Frameworks Over Time

To explore how software systems migrate from unittest to pytest, first, we need to discover the testing frameworks 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 [40] and pytest [32] 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 reference unittest only.
4. *pytest* → *pytest*: the current and past versions reference pytest only.
5. *pytest* → *unittest*: the present version of the system references unittest only, but its past versions referenced pytest only.

6. *pytest → unittest & pytest*: the present version of the system references both unittest and pytest, but its past versions referenced pytest only.

7. *other*: the current and/or past versions of the system reference neither unittest nor pytest. For example, when a system does not use one of these test frameworks or does not have tests at all.

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. No "reverse migration" was found, but a few projects (7 out of 100) that started with pytest, currently have indications of using both frameworks. However, it is important to reassure readers that the main focus of this work is not to investigate this case.

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`). Moreover, the search extends to unittest fixtures (*e.g.,* `setUp`, `setUpClass`, `tearDown`, `tearDownClass`, etc.) as well as assert usage (*e.g.,* `self.assert*`).

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 2.3a changes the test from unittest to pytest, but the new source code (right-side) has no explicit reference to the term *pytest*.[3] 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, files provided by CI/CD tools are also mined. Specifically, we consider the configuration file formats of the most popular CI/CD tools (*i.e.,* Travis, CircleCI, and GitHub Actions) are considered, as well as references to *pytest* in their configuration files (*e.g.,.travis.yml*, *config.yml* and other configuration and script files). For example, considering the aforementioned case of Figure 2.3a, 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`.[4]

Evaluation 1. We evaluate the precision of the proposed method to correctly detect the testing frameworks. First, we selected 10 Python systems and manually classified them according to the usage of the testing frameworks. These 10 systems are not part of the top

[3]Code just after the migration: https://git.io/JcSQn
[4]CI/CD file with pytest: https://git.io/JcS7E

100 systems mined in this study. The 10 systems are also presented in the dataset. For this purpose, we 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 3.1, our method correctly detected the testing frameworks in past and presents versions, achieving a precision of 100%.

Table 3.1: 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%

3.2.3 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 [32]). 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 2.3a and 2.3b. In those commits, 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 3.1 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 we detect the first migration commit (commit *m1*) and it ends when we detect the last mi-

gration commit (commit *mn*) and the system does not rely anymore on unittest. Recalling our definitions:

- If a project has never started a migration, it is *not migrated*;
- If a project has started the migration but never ended, it is *ongoing*;
- If project has started and ended the migration, it is *migrated*.

As pytest allows incremental migration [32], 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*).

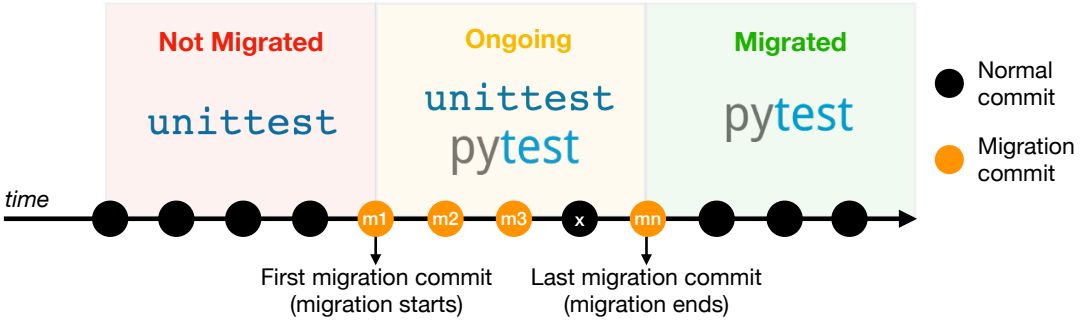

Figure 3.1: Overview of the migration stages.

Providing more examples, if a system had unittest references in the past, has only pytest references in the present, and *does not have* migration commits, it is *not* considered a migrated system by our definitions.

To detect migration commits from unittest to pytest, we assess the version history of the software systems. We rely on the PyDriller [36] 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 [40, 32], 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.,* `@unitest.expectedFailure`) and adds pytest expected failure (*i.e.,* `@pytest.mark.xfail`).

One or more transformations may happen in a single migration commit. Figure 2.3a presents an example with *assert* migrations, while Figure 2.3b has *assert* and *fixture* migrations. Project Gaphor, for example, has a migration commit[5] with three transformations: *assert*, *fixture*, and *import* migrations.

Our transformation rules are strict in the sense that a commit that has only refactoring evidence is not considered a migration commit. That is, to be considered a *migration commit*, it *must* have removal of a unittest statement and addition of the pytest equivalent. Although strict, this strategy led to less noise to identify systems that were migrating from those that only relied on both testing frameworks.

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, that are not part of the top-100 set. We then run the proposed method and manually analyze their migration commits. Specifically, for each detected migration commit, the authors of this work 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 3.2 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%.

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

Table 3.2: 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%

3.3 Qualitative Study: Why Developers Migrate

3.3.1 Assessing Reasons for the Migration

To answer and explore the migration reasons (RQ5) and guidelines (RQ6), we perform a qualitative analysis of: (1) GitHub issues/pull requests and (2) web resources from a Grey Literature Review, such as blog posts, Q&A forums, and documentation.

Issues and pull requests from GitHub. First, we assess GitHub issues and pull requests to 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.

Web Resources from Grey Literature Review. Second, to evaluate other web resources in the wild, we follow the guidelines proposed by Garousi et. al. [15] to perform a Grey Literature Review (GLR) following the steps:

1. Search process: We employ only one general web search engine, Google Advanced Search Engine, considering that it includes results from multiple outlet types. We searched the query `migrate OR move OR convert OR change OR switch unittest to pytest`, filtering by date and English language, which returned approximately 723 results.

2. Web resource selection: For the inclusion criteria, we considered: (a) is the producer reputable or has expertise in the area? (b) does the source mention advantages or disadvantages, compare both frameworks, or show tips to perform the migration? (c) is it an article published between 1/1/2018 and 1/1/2022 and written in English?, and (d) is it among the top-100 results? We assess the authority of the producer, date, and

methodology and we include a result if it fits all criteria from (a) to (d). For exclusion criteria, we filtered out the result if: (e) the material is not text-based or (f) the result is a copy of another resource. From the top-100 results, we find that 32 met all criteria; the majority was excluded due to criterion (b).

3. Data Extraction: The first author of this study carefully read all the material and organized in a tabular visualization to ease the thematic analysis process. We extracted sections that mentioned information related to the frameworks or the migration. After that, we identified the main topics of each result, based on explicit and implicit descriptions. Here, we also kept track of migration guidelines and code examples.

4. Data Validation: First, we identified 27 common topics in the selected resources. Both authors of this study reviewed the topics, then merged and renamed them, resulting in 17 topics. Whenever possible, we adopted the same topics defined while analyzing GitHub issues/PRs to keep consistency and ease the comparison process.

We adopted thematic analysis to classify the explanations of the issues/PRs and the web resources, with the following steps [10]: (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 done by the author of this dissertation, while steps 4 and 5 were done together with the advisor until consensus was achieved.

3.3.2 Exploring the Migration Process

In this final analysis, we perform a survey with developers who performed the migration from unittest to pytest. Here, we aim to better understand their perspective on the migration process. To address the last research question (RQ7), we collected the list of migration commit authors from the 27 ongoing and migrated projects. We contacted those developers by sending the following email:

1. Why has this project migrated from unittest to pytest?
2. How hard is the migration from unittest to pytest? What are the migration bottlenecks, if any?
3. (migrated systems) Do you have any advice for those projects starting the migration to pytest?
3. (ongoing systems) Why has this project started but not fully concluded the migration yet?

We collected 53 valid emails and sent them to developers, receiving 5 responses (*i.e.,* close to 10% of response rate). Those answers are then used to assess our final RQ.

3.4 Research Questions

RQ1 (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 3.2.2. **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.

RQ2 (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 3.2.3. **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.

RQ3 (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 3.3. 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 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 3.3: 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 Figure 3.1)
Migration density	Ratio of migration commits during the migration (*i.e., migration commits / total commits during the migration duration*)

RQ4 (transformations). We assess the five transformations that happen in the migration commits, *i.e.,* assert, fixture, import, skip, and failure (see Section 3.2.3). 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.

RQ5 (reasons). We focus on finding advantages and disadvantages that motivate developers on the migration. This assessment is done by searching on the GitHub API (to collect issues and pull requests) and Google Advanced Search Engine (to collect web resources). For this purpose, we follow the methodology described in Section 3.3.1. **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 plan to migrate their projects.

RQ6 (guidelines). We analyze the web resources selected with Google Advanced Search to verify whether there are code examples, tips, or a step-by-step guides to aid developers to perform the migration, also following the design described in Section 3.3.1. **Rationale:** The goal is to understand if there is support from the community, in which form this support is given, and to have a taste of how hard it can be to migrate.

RQ7 (process). We conducted a survey, as detailed in Section 3.3.2, with migration commit authors to understand what are their opinions on how hard is migrating and what bottlenecks one can encounter while migrating. **Rationale:** By surveying developers who are actually performing (or performed) the migration, we can verify if their point of view is the same or is supported by the empirical data collected in the previous research questions.

The dataset contaning the 100 systems under study, the 10 projects used in the recall and precision analysis for the quantitative study, and the GitHub issues, pull requests, and websites selected for the qualitative research is publicly available at `https://doi.org/10.5281/zenodo.6737376`.

3.5 Final Remarks

This chapter described the study design of this master dissertation. First, we introduced the Python ecosystem and the selected projects. Then, we detailed the method for the quantitative and qualitative studies. Finally, we listed and presented rationales for the investigated research question.

Chapter 4

How Developers Migrate

In this chapter, we present the quantitative results on how developers migrate from unittest to pytest. We answer research questions about the extension (Section 4.1), frequency (Section 4.2), duration (Section 4.3), and code transformation (Section 4.4). Then, we present the threats to validity (Section 4.5) and a summary of the findings as final remarks (Section 4.6).

4.1 RQ1 (extension): To what extent are unittest and pytest adopted in the Python ecosystem over time?

We selected the 100 most popular Python projects using the number of stars as a proxy for popularity. We can see some of them are maintained by big organizations *e.g.,* Google, Facebook, Microsoft, Apache, and NVIDIA, and some well-known machine learning tools *e.g.,* pandas, scikit-learn, pytorch geometric and pytorch lightning, and frameworks *e.g.,* Django and Flask.

Table 4.1 summarizes the testing frameworks used by the 100 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 4.1: 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 4.2 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 4.2: Number of files with framework references in the last version. Between parentheses: the median per system.

	#Files in the Present		
Testing Framework	**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

RQ1 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 and 27% of the unittest test files have references to pytest code.

4.2 RQ2 (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 4.3 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 4.3: 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 4.4. 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 (36), Ansible (33), and Pandas (23). Together, the top-5 have 232 migration commits, which represents 70% of the total.

Table 4.4: 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

RQ2 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.

4.3 RQ3 (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 4.5). 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, while 2 systems have 6 and 3 migration commits, migrating in 3 and 5 days, respectively. 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.

Table 4.5: 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%)

In addition, table 4.5 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.

The threshold to determine whether the system is *fast* or *slow* was defined based on the gap present in the data. When looking at the duration in days, there is a well-defined jump from 5 to 132 days. The same reasoning is applied to the *dense* and *sparse* labels, where there is an interval between 0.24% and 16.67%.

Next, we assess *when* the migration has started in each system (Table 4.6). 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 4.6: 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%

These results are relevant to understand how the migration can impact the project development. However, these starting points can be influenced by pytest release date. Pytest has a long history, and it might be hard to pinpoint the exact date of its creation given it evolved from other initiatives, but we will consider September 2004 as its debut. Among the migrated systems, all have their first commit after this date — the oldest are from 2009.

RQ3 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.

4.4 RQ4 (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. We only focus on migrated systems because their results will be stable and final, while ongoing projects could still significantly change their migration patterns until the end of the process. Recall that the 11 migrated systems have a total of 102 migration commits and that one migration commit can have one or more transformations. Table 4.7 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 [32], thus it may explain its high frequency of fixture migrations.

Table 4.7: Frequency of transformations in migrated systems.

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 measured in lines of code (LOC), as summarized in Table 4.8. We present the absolute value of line additions and deletions, their difference (represented by the delta column), and the percentage of the difference

compared to the original amount of lines of code (delta difference / unittest deletions). 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. The overall conclusion is that test code decreases after migration.

Table 4.8: Size of transformations in migrated systems.

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

RQ4 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.

4.5 Threats to Validity

Construct 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 3.2.2). Similarly, the method to detect migration commits was also manually evaluated (see Section 3.2.3), 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.

External Validity. *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.

4.6 Final Remarks

In this chapter, we presented the quantitative results to assess *how* developers migrate from unittest to pytest. We mined the history of the top 100 most popular Python systems to answer research questions regarding the extension, frequency, duration, and code transformations of the migration. We highlight the following findings:

- *Extension*: We found that found that 20% of the studied systems rely on unittest, 23% on pytest and 34% on both frameworks. From those that started with unittest, 66% now rely on pytest.
- *Frequency*: We detected that 59 systems started with unittest and now 39 rely on pytest or both frameworks. From those, 11 (28%) have fully migrated to pytest and 16 (41%) are still migrating.
- *Duration*: There is not a common pattern on migration duration. Some projects migrated up to one week while others took years to conclude. It was also observed that systems start the migration in different development phases, but most tend to concentrate the migration effort.
- *Transformations*: As expected, the code that was most migrated were assertion statements (90%), followed by fixtures (18%) and imports (13%). In general, the migrated test code is 34% smaller than the original one.

Chapter 5

Why Developers Migrate

In the previous chapter, we explored the quantitative aspects of the migration from unittest to pytest. Figure 5.1 helps us recall the current distribution of studied systems that use unittest and pytest: 20% rely on unittest, 23% on pytest and 34% on both frameworks. We also provide empirical evidence that test migration is common in the Python ecosystem. However, it is not clear the motivations behind the migration.

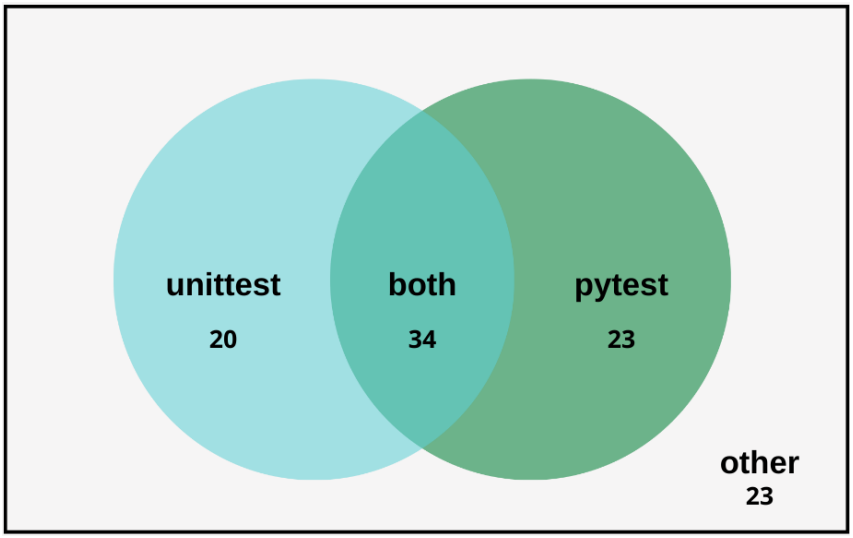

Figure 5.1: Testing frameworks adopted in the 100 studied Python systems.

In this chapter, we assess the results of the qualitative study. Specifically, we explore migration reasons (Section 5.1), guidelines (Section 5.2), and process (Section 5.3). We conclude the chapter presenting the threats to validity (Section 5.4) and final remarks (Section 5.5).

5.1 RQ5 (reasons): Why is code migrated from unittest to pytest?

5.1.1 Issues and Pull Requests from GitHub

To better understand the rationales behind the migration, we now assess the explanations provided by the developers themselves, first analyzing GitHub Issues and Pull Requests. Table 5.1 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 in the GitHub ecosystem.

Table 5.1: Migration advantages and disadvantages in the GitHub ecosystem.

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		
Parameterized tests	6		
Plugin integration	6		
Other	16		

Advantages. According to the GitHub ecosystem, 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*".[1] 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*".[2]

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*".[3]

Next, we have the categories *easier maintenance* and *fixture flexibility/reuse*, both with 8 occurrences. In project Metron, the developer mentions regarding maintainabil-

[1] https://git.io/JKiPm
[2] https://git.io/JKiXV
[3] https://git.io/JiZt7

ity: "*It would be nice to migrate the test over to pytest-django to get rid of most of the boilerplate code*".[4] 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*".[5]

Developers also present many other reasons to migrate to pytest, including *built-in features*, *popularity*, *cleaner reports*, *parameterized tests*, and *plugin integration*. For instance, in 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*".[6]

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*".[7]

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*".

5.1.2 Web Resources from Grey Literature Review

To complement the prior analysis on GitHub, we expand our search to have a more complete understanding on the reasons of the migrating process. Given that there are no extensive academic studies discussing the migration of testing frameworks in Python systems, a Grey Literature Review (GLR) is suited for this case since practitioners' productions can help us gain insights into the process [20]. For instance, in a website the author states: *"In order to increase readability and reduce repetition, I favor pytest over*

[4] https://git.io/JKid1
[5] https://git.io/JK6rv
[6] https://git.io/JKi7L
[7] https://git.io/JKiEc

unittest. PyTest offers some nice features to make writing tests faster and cleaner".[8] To better understand the Python community view, we investigate its productions and perceptions via a GLR considering distinct web resources.

To understand why developers may propose or undergo the migration, we also assess the advantages and disadvantages of web resources in the wild. From the 32 analyzed websites, most are personal blog posts (16), institutional websites (6), or Q&A forums (4). Table 5.2 presents the summary of the 14 advantages and 5 disadvantages detected.

Advantages	#	**Disadvantages**	#
Fixture flexibility	22	Separated package	14
Interoperability	17	Difficult to learn	9
Popularity	15	Implicit mechanics	2
Plugin ecosystem	12	Migration duration	2
Easier syntax	12	Other	2
Less boilerplate	12		
Parameterized tests	12		
Cleaner reports	11		
E2E tests	9		
Categorized tests	7		
Productivity	7		
Functional paradigm	5		
Python independence	3		
Other	6		

Table 5.2: Advantages and disadvantages of pytest from a GLR.

Advantages. According to the selected web resources, *fixture flexibility* (22) is the main advantage of pytest. It replaces unittest `setUp*` and `tearDown*` methods, creating the necessary state to execute a test case. Its modularity and reusability are frequently emphasized by developers: *"Fixtures are more reusable and portable and should help reduce complexity all over"*,[9] *"it leads you toward explicit dependency declarations that are still reusable"*,[10] and *"you can compose them together to create the required state"*,[11] they say.

Pytest is a framework and a test runner compatible with unittest, which means that it is possible to run unittest tests using pytest without changing one line of code. This *interoperability* (17) is frequently mentioned and can be of significant importance for systems starting the migration process, given that developers can first only change the test runner and then slowly add pytest features to the codebase.

[8] https://www.jitsejan.com/moving-from-unittest-to-pytest
[9] https://github.com/opsdroid/opsdroid/issues/1502
[10] https://realpython.com/pytest-python-testing/
[11] https://testdriven.io/blog/flask-pytest/

Next, we have *popularity* (15): in several web articles, [12,13,14] the authors start describing pytest as *"one of the most popular"* testing frameworks nowadays *"due to pytest's wide adoption"*. That also reflects in the large number of mentions of its *plugin ecosystem* (12). Commonly cited plugins are `pytest-django` for testing Django applications, `pytest-xdist`, which runs the test suite in parallel, `pytest-cov`, and `pytest-html`, which produces a coverage report and generates HTML report results, respectively.

Easier syntax (12) and *less boilerplate* (12) code are also associated to pytest because it is considered to be less verbose than unittest, without the need of remembering a list of assert methods or creating tests based on inheritance. For instance, authors say pytest *"makes it easy to test (...) with its intuitive syntax"*[15] and *"requires less boilerplate code so your test suites will be more readable"* .

The Python community also notices built-ind features, which is the case of pytest markers, represented by decorators in the form `@pytest.mark.*`. There are predefined markers but developers can also create their own custom markers and use them to run a subset of tests, for example, mark slow tests or the ones that interact with the database, creating the possibility to have *categorized tests* (7). When using the predefined `@pytest.mark.parametrize` decorator, the same test function runs with different parameters as arguments, creating *parameterized tests* (12) in the test suite.

Developers highlight other important advantages, including *cleaner error reports* (11), *end-to-end tests* (9) due to the integration with other frameworks, as Selenium, and *productivity* (7) increase. Some authors mention that the *functional paradigm* (5) makes tests simpler and the *Python independence* (3) lets it evolve faster than the language.

Disadvantages. On the other hand, being a *separated package* (14) is a downside frequently noted since *"Unittest is a part of Python itself, so it has the advantage that no install is necessary"*.[16] Authors also mention that pytest is more *difficult to learn* (9) because it requires more of Python knowledge and other advanced concepts, like decorators and dependency injection, that makes *"its learning curve is a bit steeper"* because *"some artifacts seem a bit odd at the beginning"*.[17] The *implicit mechanics* (2) is characterized by the lack of clarity due to parameterization or fixture composition, as some patterns can be a pain in large test suites, making it difficult to debug and maintain.

It is interesting to notice how the same fact can have different interpretations. Pytest being a third-party library weighs down the balance, still, this independence enables it to evolve faster. Moreover, fixtures are the main advantage of pytest, however, its poor management due to its flexibility can be a cause of problems.

[12] https://blog.methodsconsultants.com/posts/pytesting-your-python-package/
[13] https://www.wisdomgeek.com/development/web-development/python/testing-python-applications-using-pytest/
[14] https://www.linode.com/docs/guides/python-testing-frameworks-for-software-unit-testing/
[15] https://www.browserstack.com/guide/unit-testing-frameworks-in-selenium
[16] https://www.codemag.com/article/1709091/Improving-Code-Quality-with-Unit-Tests
[17] https://medium.com/worldsensing-techblog/tips-and-tricks-for-unit-tests-b35af5ba79b1

When comparing the reasons detected in the GLR and GitHub, we see that 7 advantages and 2 disadvantages in common. Even though they appear in different positions, this intersection means they are relevant. We only find a few disadvantages in the two approaches, suggesting that developers do not see many downsides when using pytest. The results from both GitHub and GLR complement each other, providing a larger perceptive of migration advantages.

RQ5 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 main concerns are the fact pytest is a separated package and the implicit mechanics.

5.2 RQ6 (guidelines): How common are guidelines to support the migration and what are their content?

When performing our analysis in the GitHub ecosystem, we noticed that some issues/PRs were tagged as *good first issue* to support newcomer contributors but few actually explained how to kick-start or perform the migration. Therefore, in our GLR, we assessed how common are guidelines to support the migration and what are their content. Next, we discuss the main findings.

Tool support. Two (6%) out of 32 GLR resources cite a tool that helps the migration process: unittest2pytest.[18] This tool, part of the pytest ecosystem, helps developers rewrite unittest test cases into pytest test cases. Available transformations are removal of unittest class and migrating `self.assert` statements. It also handles one-line tests and uses context-handlers where appropriate but, on a note for the latter, it mentions that the semantics are different and the output requires manual adjustment. Using unittest2pytest to migrate a single file, for instance, the same example as presented in Figure 2.1b, the result will be as shown in Figure 5.2b. The tool also prints to the standard output a log of the changes made. We can see that it only migrated assert statements.

Migration tips and examples. We also find that 13 (40%) out of 32 GLR resources have at least some tips, code examples, or a step-by-step on how to migrate a system. To understand what is the state of the support for the migration, we manually assessed the

[18] https://github.com/pytest-dev/unittest2pytest

```python
import unittest

class MyWidgetTestCase(unittest.TestCase):
  def setUp(self):
    self.widget = MyWidget('The widget')

  def test_default_widget_size(self):
    self.assertEqual(self.widget.size(), (50,50))

  def test_widget_resize(self):
    self.widget.resize((100,150))
    self.assertEqual(self.widget.size(), (100,150))
```

(a) Initial unittest test with setup.

```python
import unittest

class MyWidgetTestCase(unittest.TestCase):
  def setUp(self):
    self.widget = MyWidget('The widget')

  def test_default_widget_size(self):
    assert self.widget.size() == (50,50)

  def test_widget_resize(self):
    self.widget.resize((100,150))
    assert self.widget.size() == (100,150)
```

(b) Test migrated with unittest2pytest.

resource content. From those, 8 have code examples, 6 have tips, and 2 mentioned steps to perform the migration. Table 5.3 details this analysis.

Code Migration Examples	#	**Migration Tips**	#
Replace assert	7	Incremental Migration	3
Remove unittest subclass	6	Search unittest subclasses	2
Replace fixture	5	Create common fixtures	2
Replace import	3	Use pytest runner	1
Replace raise	2	Use pytest fixtures	1

Table 5.3: Code examples and tips.

Regarding the migration examples, most show how one can transform test assertions from `self.assert*` unittest methods to the plain built-in `assert` statement and how to write functional tests without relying on the inheritance of the `TestCase` class. Then, a more complex example is the migration from `setup` and `teardown` methods to the `@pytest.fixture` decorator to create the state needed to run tests, like presented in Figure 2.3. Less frequently, we have examples showing when one needs to replace the `import unittest` or similar statement by the `import pytest` one, which is only needed when calling a decorator or a specific function, for instance `@pytest.fixture` or `pytest.raises` when asserting exceptions.

Regarding the migration tips, performing *incremental migration* is the most common recommendation. Trying to do all at once, especially in large test codebases, is simply unfeasible, so migrating small portions is the way to go. To check where to start, developers can *search unittest subclasses* and map files and methods that can be migrated. They can also analyze test cases to seek patterns in predefined states before the test is run and then *create common fixtures*, increasing reusability, readability, and maintainability. Authors suggested to add these common fixtures to `conftest.py` files, which are per package files that provide fixtures and other hooks for an entire directory. *Using pytest runner* and *fixtures* to power unittest test are other options to slowly start the migration and leverage its features.

Finally, two resources presented a tutorial to perform the migration. We extracted the non project-specific steps and they say to:

1. *Setup pytest.* Mandatory since pytest is not part of the standard library. However, it can be easily added by the package installer for Python (`pip`).[19]
2. *Rewrite assertions.* One of the most straightforward transformations to do because there is no need to go through the semantics of the test and just make a syntactic replacement.
3. *Move tests out of classes.* Adopt the functional paradigm proposed by pytest and stop inheriting from `unittest.TestCase`.
4. *Move setup operations to fixtures* and benefit from one of pytest's greatest advantage.

RQ6 Conclusion. A significant part of the resources (40%) have some guidance to help practitioners on the migration, including tools and guidelines. We find mainly code migration examples, suggesting the migration of asserts, subclasses, and fixtures. We also find migration tips and detailed steps to support the migration.

5.3 RQ7 (process): What is the migrator's view on the process?

In this final research question, we perform a survey with developers who performed the migration. We contacted 53 developers and received 5 responses. Overall, based on developer perspectives, several factors can influence the migration, such as the size of the codebase, the experience of the team with either framework or with a migration process, and how motivated and confident the team is to undergo the migration. Next, we present a summary of the survey answers:

- **Why performing the migration.** The main reasons are to use pytest features, as fixtures and parameterized tests (2), cleaner syntax (2), popularity (1), extensibility (1), and better performance (1). We recall that many of those advantages are detected in our thematic analysis on GitHub and web resources.
- **How hard is the migration and what are the bottlenecks.** Three developers found the migration to be hard and two considered it easy. From their experience,

[19] https://pypi.org/project/pip/

the main challenges are the size of the test suite (3), manual effort (3), flaky tests (1), and the embracing phase (1) to adopt pytest style.

- **Why the migration is not complete.** They shared various reasons that delays the process: the team only migrated when needed (1) similar to the "Boy Scout Rule" approach [26], the never ending embracing phase where people organically decide to apply pytest style (1), the lack of compatibility with build tools (1), the disagreement among the team if pytest is indeed the best choice for that moment (1), and the lack of people to contribute to the migration (1). Despite being considered *ongoing* systems by our approach, two developers stated that the migration was completed, but only one could provide proof of an announcement saying it was complete. This could demonstrate that the way the team tracks the migration is not explicit or that developers are still not used to the pytest style and continue to inadvertently add unittest structures in tests.

Lastly, two developers mentioned they took advantage of the migration to refactor test code and one shared an interesting problem: the projects they worked on had custom implementation for testing structures, hence *"unittest class methods and pytest fixture methods are not quite the same"* and *"nearly all the test assertions need to be rewritten"*.

RQ7 Conclusion. According to developers who performed the migration, the main reasons to migrate are pytest features and the easier syntax. On the other hand, the size of the codebase and manual effort are predominant bottlenecks.

5.4 Threats to Validity

Construct Validity. *Query construct.* The first threat that can impact the most in the repeatability of the Grey Literature Review is the *term queried* in the search engine in the first place. The way the query was constructed and the inclusion criteria defined might exclude important results, for instance, those that were published out of the specified date range or those that used other words to describe the process. Following the same logic, it can include unwanted results, for example, those that use the term "unittest" to refer to the unit testing practice and not to the framework. However, the authors performed an initial search to check which words were more common and they tested with both more restrictive and more comprehensive terms, but the results seemed to be less accurate.

Internal Validity. *Manual classification of issues/PRs and web resources.* In RQ5, we manually classify the explanations found in issues/PRs and other resources about the migration advantages and disadvantages. In this case, we rely on thematic analysis [10] to reduce the subjectiveness.

Conclusion Validity. *Survey sample size.* The low number of respondents of the survey impact the perspective gained for the process. Although they provided great explained answers, having more is definitely wanted to achieve a more precise conclusion.

5.5 Final Remarks

In this chapter, we analyzed issues and pull requests from GitHub and web resources from a Grey Literature Review, including blog posts, Q&A forums, and documentation. Overall, the analyzed resources highlight as advantages pytest fixtures, interoperability with unittest, easier maintenance, and popularity, but they also mention implicit mechanics and separated package as concerns. Code examples, tips, or tutorials are present in 40% of resources outside GitHub to support the migration process. Finally, in our survey, we also find migration bottlenecks, including the size of the test suite and the manual effort.

Chapter 6

Discussion and Implications

Based on our findings, we present implications for both practitioners (Section 6.1) and researchers (Section 6.2), how the migration can impact on their work, and suggestions of unexplored themes that arise from these discussions.

6.1 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 (RQ1). While some systems have fully migrated (11), others are still migrating (16), suggesting that the migration is not always straightforward (RQ2). Most projects (8 out of 11) that fully migrated concentrate the migration commits and start the migration in distinct development phases (RQ3). Overall, the migrated test code is 34% smaller than the original one, suggesting that there is less test code to be maintained (RQ4). 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* (RQ5). 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 (RQ2). 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 RQ3, 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 RQ5, 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* [37]. In the dataset of RQ5, 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, 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.

After assessing multiple resources, tips, and tutorials, we may question: *is there a best way to do the migration*? As mentioned by various authors, it depends on the project characteristics, but it is possible to indicate some steps that can guide the work:

1. *Setup pytest and use its runner.* Pytest's interoperability with unittest is one of its main advantages, so developers can start adding pytest to the codebase by using its runner, without modifying any test. Some modifications in internal guides and in CI/CD tools may be needed in this step.

2. *Perform the syntactic migration.* In this step, one can rewrite assertions, search unittest subclasses, and remove them. Empirically, assert migration is the most performed and both are the two most exemplified. Developers can use their preferred text editor or the unittest2pytest tool to replace `self.assert*` methods by the plain `assert` statement. Similarly, they can use the same approach to remove `TestCase` subclasses and only have test functions.

3. *Migrate context-dependent structures.* Transforming setup methods in fixtures or migrating other context-dependent structures is probably one of the most time consuming steps of the migration, depending on how many context managers and setup

operations are already in place and the size of the codebase. While migrating, developers can also observe and identify common fixtures and add them to the `conftest.py` file.

4. *Clean-up and finishing phase.* Lastly, there could be some unnecessary `import` statements remaining in test files, for example. At this point, there could be an announcement to inform developers the migration ended and the adoption of linting rules to reinforce new practices.

Why developers migrate. We investigated multiple resources to understand the reasons behind the migration. Common topics in the top 5 advantages are *Fixture flexibility*, *Interoperability*, and *Easier syntax*, which strengthens the importance of these features and their impact in motivating developers to start the migration. Survey answers also corroborate this finding, making *fixtures* and the *syntax* the main reasons highlighted in pytest. We see that being a *separated package* and *requiring advanced Python knowledge* to learn pytest is frequent concern, but the *implicit mechanics* and *migration duration* is also significant for developers.

When reviewing GLR resources, we found several elaborated articles mentioning advantages, disadvantages, and many showing code examples or tips to perform the migration. Regarding the count of all themes per resource, it is possible to compare the discussion in each group of resources. While in GitHub issues and pull requests have a ratio of 1.6 theme per resource, in the GLR websites the ratio is 5.6 themes per resource, 3.5 times higher than the former. This result leads us to conclude that developers make a deeper analysis of the framework outside the GitHub ecosystem.

6.2 For Researchers

Why the migration is not concluded. We found that dozens of popular systems started but did not conclude the migration (RQ2). 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 (RQ1). This raises the following question that can be addressed by further research: *why are some migrations started but not completed?* In RQ5, 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.

Tools and techniques to document and automate the migration. While assessing the articles, only 2 quickly mention a tool, unittest2pytest, that could aid the migration. This tool, which is also referred in pytest documentation, does the assert migration and the removal of `unittest.TestCase` subclasses, but requires manual adjustments if there are tests using context managers. That is, the tool only guarantees syntactic migrations. To migrate context-dependent structures, for instance `with self.assertRaises` and fixtures, one would need a tool that accounts for the syntax and the semantics, however, that would be more difficult to develop and the result would need to be reviewed anyway. That lead us to the question: *are tools not relevant for the migration?* Some hypothesis that can be investigated in future work are (i) developers do not see value in them, given that syntactic migration could be easily done with the help of text editors and context-dependent structures will need their review; (ii) the tools are not well promoted; (iii) their efficacy do not meet developers' expectations.

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 [33, 17, 39]). 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 (RQ2). 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 [48, 42, 29, 22, 38, 16, 13]. Therefore, we envision that novel tools and techniques can be proposed by researchers to automate the migration and reduce the migration delay.

6.3 Final Remarks

In this chapter, we detailed implications for practitioners and researchers. For practitioners, we discussed the migration advantages and disadvantages, the challenges of keeping track of the migration, and what kind of guidelines they can expect from resources. Finally, for researchers, we proposed new questions to be investigated, to understand why the migration is not concluded, and insights to automate the migration.

Chapter 7

Conclusion

This chapter concludes this master dissertation. We provide an overview of our the empirical study in Section 7.1 and we propose future work in Section 7.2.

7.1 Overview and Contributions

We presented an empirical study to assess *how* and *why* developers migrate from unittest to pytest. In our quantitative study to explore how developers migrate, 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. Next, we provided a qualitative study to assess why developers migrate. Despite being a separate package and requiring advanced Python knowledge, we find several pytest advantages, including its built-in features as fixtures, interoperability, and the easy syntax. Resources from the Grey Literature Review provided further insights about pytest pros and cons and code examples to migrate. Nevertheless, they both lack a set of best practices empirically tested. Most migration authors share that the size of the codebase and manual effort are the main bottlenecks they experienced.

The contributions of this master dissertation can be summarized as follows:

- It provides the first empirical study to assess the migration of *testing* frameworks in the Python ecosystem.
- It explored how code is migrated and the reasons for the migration.
- It proposes practical implications for practitioners, by discussing practices, guidelines, and reasons, and for researchers, by suggesting other paths worth exploring.

7.2 Future Work

We explored the migration major reasons and the content of the migration content We also assessed the migrator's view of the process. Still, there are more questions worthy of further investigation. For instance, we do not have a clear idea of how developers track the migration and whether the Python community has a set of good practices to migrate.

In our survey, one developer talked about the embracing phase, which leads to other questions: how do teams enforce and keep the style after migrating? Does it demand effort from maintainers, or does it happen naturally? Do they use any metrics or benchmarks to understand how the system changed? Do the migration influence the increase in the test coverage? How do they follow up the migration? These questions can be further investigated, to characterize, from the point of view of developers, what happens during and after the migration.

Bibliography

[1] Hussein Alrubaye, Deema Alshoaibi, Eman Alomar, Mohamed Wiem Mkaouer, and Ali Ouni. How does library migration impact software quality and comprehension? an empirical study. In Sihem Ben Sassi, Stéphane Ducasse, and Hafedh Mili, editors, *Reuse in Emerging Software Engineering Practices*, pages 245–260, Cham, 2020. Springer International Publishing.

[2] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali 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)*, pages 414–417, 2019.

[3] Lívia Barbosa and Andre Hora. How and why developers migrate python tests. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1–11, 2022.

[4] Gabriele Bavota, Mario Linares-Vasquez, Carlos Eduardo Bernal-Cardenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. The impact of API change-and fault-proneness on the user ratings of Android apps. *IEEE Transactions on Software Engineering*, 41(4):384–407, 2014.

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

[6] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to break an api: cost negotiation and community values in three software ecosystems. In *International Symposium on Foundations of Software Engineering (FSE)*, pages 109–120, 2016.

[7] Hudson Borges, Andre Hora, and Marco Tulio Valente. Understanding the factors that impact the popularity of GitHub repositories. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344, 2016.

[8] Aline Brito, Marco Tulio Valente, Laerte Xavier, and Andre Hora. You broke my code: Understanding the motivations for breaking changes in APIs. *Empirical Software Engineering*, 25:1458–1492, 2020.

[9] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. Why and how Java developers break APIs. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 255–265, 2018.

[10] Daniela S Cruzes and Tore Dyba. Recommended steps for thematic synthesis in software engineering. In *International Symposium on Empirical Software Engineering and Measurement*, pages 275–284, 2011.

[11] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the evolution of technical lag in the npm package dependency network. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 404–414, 2018.

[12] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *International Conference on Mining Software Repositories (MSR)*, pages 181–191, 2018.

[13] Mattia Fazzini, Qi Xin, and Alessandro Orso. Automated API-usage update for Android apps. In *International Symposium on Software Testing and Analysis*, pages 204–215, 2019.

[14] Michael Feathers. *Working Effectively with Legacy Code*. Prentice Hall Professional, 2004.

[15] Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology*, 106:101–121, 2019.

[16] Stefanus A Haryono, Ferdian Thung, Hong Jin Kang, Lucas Serrano, Gilles Muller, Julia Lawall, David Lo, and Lingxiao Jiang. Automatic Android deprecated-API usage update by learning from single updated example. In *International Conference on Program Comprehension*, pages 401–405, 2020.

[17] Andre Hora, Nicolas Anquetil, Stéphane Ducasse, and Simon Allier. Domain specific warnings: Are they any better? In *International Conference on Software Maintenance (ICSM)*, pages 441–450, 2012.

[18] Andre Hora, Romain Robbes, Marco Tulio Valente, Nicolas Anquetil, Anne Etien, and Stephane Ducasse. How do developers react to API evolution? a large-scale empirical study. *Software Quality Journal*, 26(1):161–191, 2018.

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

[20] Fernando Kamei, Igor Wiese, Crescencio Lima, Ivanilton Polato, Vilmar Nepomuceno, Waldemar Ferreira, Márcio Ribeiro, Carolline Pena, Bruno Cartaxo, Gustavo

Pinto, and Sérgio Soares. Grey literature in software engineering: A critical review. *Information and Software Technology*, 138:106609, 2021.

[21] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, 23(1):384–417, 2018.

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

[23] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. Cid: Automating the detection of api-related compatibility issues in android apps. In *International Symposium on Software Testing and Analysis*, pages 153–163, 2018.

[24] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Characterising deprecated android apis. In *International Conference on Mining Software Repositories (MSR)*, pages 254–264, 2018.

[25] B.A. Malloy and J.F Power. An empirical analysis of the transition from Python 2 to Python 3. In *Empirical Software Engineering*, page 751–778. Springer International Publishing, 2019.

[26] Robert C. Martin and James O. Coplien. *Clean code: a handbook of agile software craftsmanship*. Prentice Hall, Upper Saddle River, NJ [etc.], 2009.

[27] Matias Martinez and Bruno Gois 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.

[28] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of API stability and adoption in the Android ecosystem. In *International Conference on Software Maintenance*, pages 70–79, 2013.

[29] Sichen Meng, Xiaoyin Wang, Lu Zhang, and Hong Mei. A history-based matching approach to identification of framework evolution. In *International Conference on Software Engineering (ICSE)*, pages 353–363, 2012.

[30] Romulo Nascimento, Eduardo Figueiredo, and Andre Hora. JavaScript API Deprecation Landscape: A Survey and Mining Study. *IEEE Software*, 2021.

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

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

[33] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Domain-specific program checking. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 213–232. Springer, 2010.

[34] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. On the reaction to deprecation of clients of 4+ 1 popular java apis and the jdk. *Empirical Software Engineering*, 23(4):2158–2197, 2018.

[35] Hudson Silva and Marco Tulio Valente. What's in a GitHub star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112–129, 2018.

[36] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. Pydriller: Python framework for mining software repositories. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 908–911, 2018.

[37] Xin Tan, Minghui Zhou, and Zeyu 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*, pages 398–409, 2020.

[38] Ferdian Thung, Stefanus A Haryono, Lucas Serrano, Gilles Muller, Julia Lawall, David Lo, and Lingxiao Jiang. Automated Deprecated-API Usage Update for Android Apps: How Far Are We? In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 602–611, 2020.

[39] Kristín Fjóla Tómasdóttir, Mauricio Aniche, and Arie Van Deursen. The adoption of JavaScript linters in practice: A case study on ESLint. *IEEE Transactions on Software Engineering*, 46(8):863–891, 2018.

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

[41] Jiawei Wang, Li Li, Kui Liu, and Haipeng 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*, pages 233–244, 2020.

[42] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. Aura: a hybrid approach to identify framework evolution. In *International Conference on Software Engineering (ICSE)*, pages 325–334, 2010.

[43] Shengzhe Xu, Ziqi Dong, and Na Meng. Meditor: Inference and application of api migration edits. In *International Conference on Program Comprehension (ICPC)*, pages 335–346, 2019.

[44] Rodrigo Elizalde Zapata, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, Kenichi Matsumoto, and Akinori 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)*, pages 559–563, 2018.

[45] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús González-Barahona. An empirical analysis of technical lag in npm package dependencies. In *International Conference on Software Reuse*, pages 95–110. Springer, 2018.

[46] Ahmed Zerouali and Tom Mens. Analyzing the evolution of testing library usage in open source Java projects. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 417–421, 2017.

[47] Zhaoxu Zhang, Hengcheng Zhu, Ming Wen, Yida Tao, Yepang Liu, and Yingfei Xiong. How do python framework apis evolve? an exploratory study. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 81–92, 2020.

[48] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining api mapping for language migration. In *International Conference on Software Engineering*, pages 195–204, 2010.