Empirical Software Engineering manuscript No.
(will be inserted by the editor)

How and Why Developers Implement OS-Specific Tests

Ricardo Job · Andre Hora

Received: date / Accepted: date

Abstract **Context.** Real-world software systems are often tested in multiple operating systems (OSs). Consequently, developers may need to handle specific OS requirements in tests. For example, different OSs have distinct file path name conventions (*e.g.,* between Windows and Unix), thus, the tests should be adapted to run differently depending on whether the OS is Windows or Unix. In this context, an *OS-specific test* is a test that identifies the OS it will be executed. OS-specific tests may execute different lines of code of the application depending on the OS they are running. **Objective.** In this paper, we provide the first empirical study to assess OS-specific tests, exploring *how* and *why* developers implement this kind of test. This knowledge can help us understand OS-specific tests and the challenges faced by developers when testing for multiple operating systems. **Method.** We mine 100 popular Python systems and assess their OS-specific tests both quantitatively and qualitatively. We propose five research questions to assess the frequency, location, target, operations, and reasons. **Results.** (1) We find that OS-specific tests are common: 56% of the analyzed Python projects have OS-specific tests and Windows is the most targeted OS. (2) We detect that OS verification happens more frequently in test decorators (65%) than in test code (35%). (3) OS-specific tests target a diversity of code, including file/directory, network, and permission/privilege. (4) Developers may perform multiple operations in OS-specific tests, including calling OS-specific APIs, mocking OS-specific objects, and suspending execution. (5) We find that OS-specific tests are implemented mostly to overcome unavailable external resources, unsupported standard libraries, and flaky tests.

Ricardo Job
Unidade de Informática, Instituto Federal da Paraíba (IFPB), Cajazeiras, Brazil,
Department of Computer Science, Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, Brazil, E-mail: ricardo.job@ifpb.edu.br

Andre Hora
Department of Computer Science, Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, Brazil, E-mail: andrehora@dcc.ufmg.br

Conclusions. Finally, based on our findings, we discuss practical implications for practitioners and researchers, including the relation of OS-specific tests with test smells, CI/CD, technical debt, and flaky tests. We also discuss the efforts to test on Windows properly and propose a novel refactoring to improve some instances of OS-specific tests.

Keywords Software Testing · Test Smells · Technical Debt · Python · Mining Software Repositories

1 Introduction

Software testing is a key activity in modern software development. Having a good test suite is fundamental to ensuring software quality and sustainable software evolution [6, 18, 19, 25]. When software systems grow over time and become more complex, test suites should also evolve to accommodate novel tests and requirements [43].

In practice, real-world software systems are often tested in multiple operating systems (OSs). For this purpose, developers may need to handle specific OS requirements in tests. For example, different operating systems have distinct path name conventions, mainly between Windows and Unix-style paths. Thus, developers should adapt the tests to run differently depending on whether the OS is Windows or Unix. In this context, an *OS-specific test* is a test that identifies the operating system it will be executed. Consequently, OS-specific tests may execute different lines of code of the application depending on the OS they are run. While OS-specific tests may make the test suite more flexible [29], they may change test behavior according to the executed OS. Another aspect is that developers may not be able to fully execute OS-specific tests on their local machines because multiple operating systems would be required. In practice, OS-specific tests need to be executed in CI/CD tools or containers to get the proper test result. This has side effects on code coverage metrics because distinct application code may be executed depending on the target OS. Indeed, in large and complex test suites, this is an unavoidable issue. For example, the Python testing documentation states that developers may not be able to get 100% coverage due to platform-specific code [24]: "*Do realize, though, that getting 100% coverage is not always possible. There could be platform-specific code that simply will not execute for you [...]*".[1]

Figure 1 presents a snippet of an OS-specific test extracted from the Django web framework. The test `test_serialize_pathlib`[2] calls the API `sys.platform`[3] to identify the current OS (line 512) and verifies that path names properly work in Windows and other OSs. The `if` block verifies if the OS is Windows and performs Windows-specific checks and assertions, while

[1] https://devguide.python.org/testing/coverage

[2] https://github.com/django/django/blob/f3c89744cc801cc7d134bca9958c4a74aa76380f/tests/migrations/test_writer.py#L512-L521

[3] https://docs.python.org/3/library/sys.html#sys.platform

the `else` block runs for other OSs, like Linux and macOS. Notice that the OS check happens directly in the test code, however, it can also happen in `@skipif` test decorators.

```
512         if sys.platform == "win32":
513             self.assertSerializedEqual(pathlib.WindowsPath.cwd())
514             path = pathlib.WindowsPath("A:\\File.txt")
515             expected = ("pathlib.PureWindowsPath('A:/File.txt')", {"import pathlib"})
516             self.assertSerializedResultEqual(path, expected)
517         else:
518             self.assertSerializedEqual(pathlib.PosixPath.cwd())
519             path = pathlib.PosixPath("/path/file.txt")
520             expected = ("pathlib.PurePosixPath('/path/file.txt')", {"import pathlib"})
521             self.assertSerializedResultEqual(path, expected)
```

Fig. 1: Snippet of an OS-specific test in the Django project (`test_serialize_pathlib`).

We found that OS verification is common in tests: we analyzed the top-100 most popular Python systems on GitHub and detected that over 50% have OS-specific tests. Despite being largely used in the Python ecosystem, we are not yet aware of how developers implement these tests or the reasons for their implementation. This knowledge can be used to understand OS-specific tests better, discover the challenges faced by developers when testing for multiple operating systems, and support the development of novel guidelines and tools to correct OS-specific tests.

In this paper, we provide the first empirical study to assess OS-specific tests. Our goal is to understand *how* and *why* developers implement OS-specific tests. We explore the usage of APIs to identify the current operating systems in both test code and test decorators. Our case study on 100 Python systems enables us to answer the following research questions:

– *RQ1: How frequent are OS-specific tests?* We aim to better understand to what extent OS-specific tests occur in real-world software projects and which operating systems are most targeted. We find that OS-specific tests are common: 56% of the analyzed Python projects have OS-specific tests, and Windows is the most targeted OS.
– *RQ2: Where are OS-specific tests implemented?* We target to assess where APIs to identify OSs are most frequently located in tests: test code or test decorators. So far, we are not aware of where developers use this kind of API or what their preferred solution is. We find that APIs to identify OSs are used more frequently located in test decorators (65%) than in test code (35%). The pytest decorator `@skipif` is the most used to skip tests depending on the operating system.
– *RQ3: What code is targeted in OS-specific tests?* We aim to explore what code is more likely to have OS-specific tests. This may reveal the kind of code that deserves more attention from developers when implementing tests for multiple operating systems We find that OS-specific tests target

a diversity of code, including file/directory, OS process, network, permission/privilege, numeric precision, and timezone. The most frequent code targeted by OS-specific tests are file/directory (36.01%), third-party dependency (25.65%), and OS process (11.14%).
- *RQ4: What operations are performed in OS-specific tests?* Our goal is to uncover operations developers perform in OS-specific tests. This may provide the basis to reason whether these operations are best or bad practices. Developers perform five operations in OS-specific tests: set OS-specific value (62.68%), skip test (17.66%), call OS-specific API (13.39%), mock OS-specific object (3.42%), and suspend execution (2.85%).
- *RQ5: Why are OS-specific tests implemented?* We aim to uncover the reasons behind the implementation of OS-specific tests. So far, it is unclear why developers implement OS-specific tests in the wild. We find that OS-specific tests are implemented mostly to overcome unavailable external resources (29.58%), unsupported standard libraries (18.08%), and flaky tests (13.85%). Other reasons include distinct path convention (11.74%), specific CI/CD environment (10.56%), and unsupported file operation (6.34%).

Finally, based on our findings, we discuss practical implications for practitioners and researchers. First, (1) we discuss the relation of OS-specific tests with existing test smells like *tests with conditional logic* [29, 32, 33] and *rotten green tests* [3, 16], showing that OS-specific tests may give us false confidence that the code under test is valid. (2) We elaborate on the fact that OS-specific tests may be actually executed in CI/CD environments. (3) We discuss the OS-specific tests in light of technical debt and self-admitted technical debt (SATD) [34, 36], providing evidence that some OS-specific tests can indicate sub-optimal implementations that should be fixed later. (4) We elaborate on the efforts to test on Windows properly and recommend that developers rely on OS-independent APIs to fix some OS-specific tests related to path conventions. (6) We elaborate on the fact that developers may create OS-specific tests to skip flaky tests [17, 22, 27] instead of fixing the non-determinism and recommend that future research on flaky tests should take into account OS-specific needs. Finally, (6) we propose a novel refactoring to improve some instances of OS-specific tests.

Contributions: The contributions of this paper are twofold: (1) we provide the first empirical study to quantitatively and qualitatively explore the usage of OS-specific tests and (2) we propose actionable implications for research and practitioners.

Structure: Section 2 presents the study design, while Section 3 details the results. Section 4 discusses the results and implications. Section 5 presents the threats to validity and Section 6 details the related work. Finally, Section 7 concludes the paper.

2 Study Design

2.1 Study Overview

Figure 2 presents an overview of our study design to analyze OS-specific tests, which includes five major steps. First, we select the software systems to be analyzed (Section 2.2). Second, we detect the APIs that are able to identify the operating systems (Section 2.3). Third, we explore how the APIs to identify the OS can be used by developers (Section 2.4). Finally, we detect systems with OS-specific tests (Section 2.5) and analyze the detected OS-specific tests both quantitatively and qualitatively (Section 2.6).

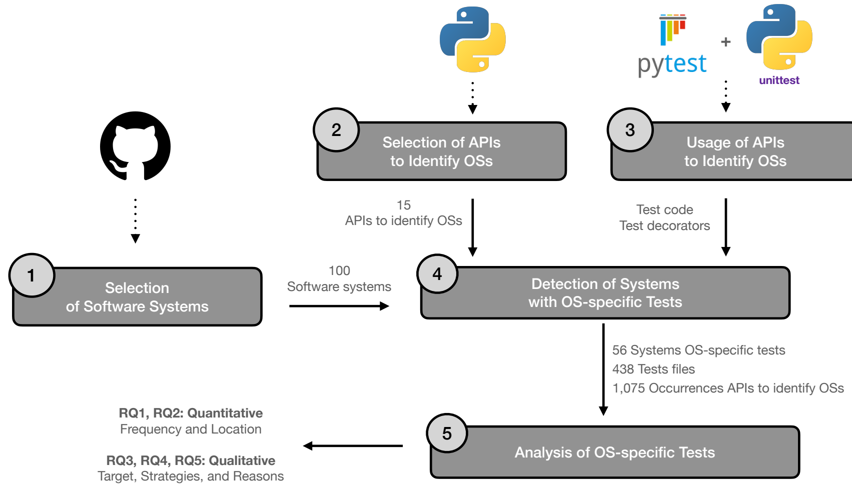

Fig. 2: Overview of the study design.

2.2 Selection of Software Systems

In this research, we aim to assess real-world and relevant software systems. We select systems written in Python for two reasons. First, Python is among the most important programming languages nowadays according to both GitHub and TIOBE rankings. Second, Python has a rich software ecosystem with widely adopted projects to support web development, machine learning, and data analysis, to name a few. We select 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 [8, 9].

To support our case study, we rely on the GitHub Search tool (GHS) [13].[4] We started with the top-220 most popular Python repositories hosted on GitHub according to the number of stars. Then, we applied the following exclusion criteria: (1) forked repositories, (2) repositories without tests, and (3) tutorials, examples, and sample projects, as detailed in Figure 3. Notice that in this process, we took special care to filter out projects without tests because those are not relevant to our research. We also filter out non-software projects, such as tutorials, examples, and code samples.

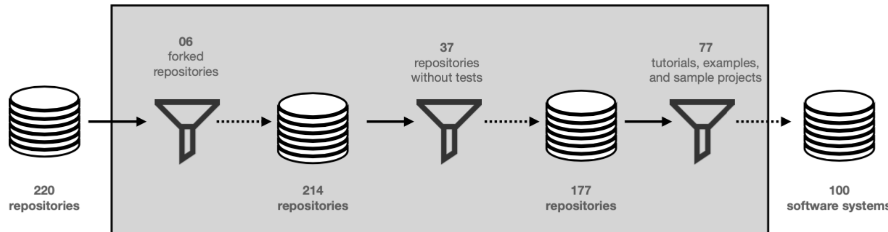

Fig. 3: Filtering criteria to select the 100 software systems.

The 100 selected software systems are presented in our publicly available dataset [1]. On the median, these systems have 17,203 stars, 3,710 commits, and 84 test files. In total, the 100 selected systems have 20,957 test files. Table 1 presents the top-10 most popular selected systems. They include projects that are broadly adopted worldwide, such as Django, Flask, and scikit-learn.

Table 1: Top-10 most popular selected software systems.

Pos	Project	Stars	Commits	Test Files	With OS-Specific Tests
1	ytdl-org/youtube-dl	113,613	18,612	34	1
2	nvbn/thefuck	75,785	1,636	203	1
3	django/django	64,786	30,840	1,888	20
4	pallets/flask	60,620	4,847	48	1
5	keras-team/keras	56,080	7,308	361	2
6	ansible/ansible	53,831	52,820	1,073	5
7	scikit-learn/scikit-learn	50,393	28,503	294	5
8	tiangolo/fastapi	50,004	2,409	510	0
9	psf/requests	49,041	6,139	14	1
10	ageitgey/face_recognition	47,432	238	2	0

[4] https://seart-ghs.si.usi.ch

2.3 Selection of APIs to Identify OSs

An *OS-specific test* is a test that identifies the current operating system it will be executed. We classify a test as OS-specific when it calls an API to identify the operating system. For example, the test `test_serialize_pathlib` presented in Figure 1 has a call to the API `sys.platform` which returns a string with the current operating system.

To properly detect OS-specific tests, we need to find APIs that can identify the operating systems. For this purpose, we manually inspected the *Generic Operating System Services*[5] document, which is provided by the official Python documentation and summarizes all OS-related interfaces supported by the Python Standard Library. We find three relevant libraries that can be used to identify operating systems: `sys`,[6] `os`,[7] and `platform`.[8] These libraries provide miscellaneous platform and operating system interfaces. Next, we perform a manual inspection of their documentation to spot APIs that identify the current operating systems. For example, the API `os.uname`[9] is selected because it has a clear description of its goal: *"Returns information identifying the current operating system"*. After this process, we find 15 APIs to identify the current OS, as summarized in Table 2. Notice that ten APIs belong to the `platform` library, three to the `os`, and two to the `sys`.

Table 2: Selected APIs to identify OSs.

Library	**API**
sys	`sys.platform` `sys.getwindowsversion`

os	`os.name`
	`os.supports_bytes_environ`
	`os.uname`

platform	`platform.platform`
	`platform.system`
	`platform.version`
	`platform.uname`
	`platform.win32_edition`
	`platform.win32_ver`
	`platform.win32_is_iot`
	`platform.mac_ver`
	`platform.libc_ver`
	`platform.freedesktop_os_release`

[5] https://docs.python.org/3/library/allos.html
[6] https://docs.python.org/3/library/sys.html
[7] https://docs.python.org/3/library/os.html
[8] https://docs.python.org/3/library/platform.html
[9] https://docs.python.org/3/library/os.html#os.uname

2.4 Usage of APIs to Identify OSs

When implementing tests, developers can use APIs to identify the current operating system in (1) test code or (2) test decorators. Next, we provide an overview of both solutions.

Test code. Figure 4 presents an example in which an API is used in the test code. In this case, the test `test_magic_parse_options` relies on the API `os.name` (line 168) to adapt the expected result according to the OS. It is important to note that developers may use APIs to identify the current OS in test methods and test support methods. Figure 5 presents an example in which the API `os.name` is used in the support method `setUp` (line 1,034).

```
161    def test_magic_parse_options():
162        """Test that we don't mangle paths when parsing magic options."""
163        ip = get_ipython()
164        path = 'c:\\x'
165        m = DummyMagics(ip)
166        opts = m.parse_options('-f %s' % path,'f:')[0]
167        # argv splitting is os-dependent
168        if os.name == 'posix':
169            expected = 'c:x'
170        else:
171            expected = path
172        assert opts["f"] == expected
```

Fig. 4: OS-specific test in the IPython project (`test_magic_parse_options`).

```
1030        def setUp(self):
1031            super().setUp()
1032            po_file = Path(self.PO_FILE)
1033            po_file_tmp = Path(self.PO_FILE + ".tmp")
1034            if os.name == "nt":
1035                # msgmerge outputs Windows style paths on Windows.
1036                po_contents = po_file_tmp.read_text().replace(
1037                    "#: __init__.py",
1038                    "#: .\\__init__.py",
1039                )
1040                po_file.write_text(po_contents)
1041            else:
1042                po_file_tmp.rename(po_file)
1043            self.original_po_contents = po_file.read_text()
```

Fig. 5: OS-specific test in the Django project (`test_extraction.py`).

Test decorators. The other option is to use the APIs to identify the current operating system in test decorators. The two major Python frameworks (unittest [40] and pytest [35]) allow the usage of decorators to skip tests if certain conditions hold [4]. Figure 6 presents an example in which the API

`sys.platform` is used in the pytest decorator `@skipif` (line 152) to skip the test if the OS is Windows. Notice that the developer may provide a reason to skip the test. In this case, the developer explains that the test is "*not supported on Windows yet*" (line 152). Note that a *test decorator* is basically a syntactic sugar for a conditional. For instance, the example presented in Figure 6 is equivalent to having the test code within the conditional `if sys.platform != "win32"`.

```
152     @pytest.mark.skipif(sys.platform == "win32", reason="not supported on windows yet")
153     def test_sanitize_path():
154         path, is_remote = cp._sanitize_path("r:default-project", "/")
155         assert path == "/default-project"
156         assert is_remote
157
158         path, _ = cp._sanitize_path("r:foo", "/default-project")
159         assert path == "/default-project/foo"
160
161         path, _ = cp._sanitize_path("foo", "/default-project")
162         assert path == "foo"
```

Fig. 6: OS-specific test in the Lightning project (`test_sanitize_path`).

To identify all possible decorators that can be used to skip tests conditionally, we perform a manual inspection of both unittest [40] and pytest [35] documentation. Table 3 presents the identified decorators: three in unittest (`skipIf`, `skipUnless`, and `expectedFailure`) and two in pytest (`skipif` and `xfail`).

Table 3: Selected test decorators.

Testing Framework	Decorator

unitttest	`@unittest.skipIf`
	`@unittest.skipUnless`
	`@unittest.expectedFailure`

pytest	`@pytest.mark.skipif` `@pytest.mark.xfail`

2.5 Detection of Systems with OS-specific Tests

We implemented an AST (Abstract Syntax Tree) analysis tool to mine Python source code and detect OS-specific tests. The tool uses the standard AST library[10] to detect the presence of the selected APIs to identify OSs (see Table 2) both in test code and test decorators. We ran the proposed tool on the 100

[10] https://docs.python.org/3/library/ast.html

selected systems and detected 438 test files with OS-specific tests and 1,075 occurrences of APIs to identify OSs (372 in test code and 703 in test decorators). Among the top-10 projects presented in Table 1, we find that eight projects have OS-specific tests: django (20), scikit-learn (5), ansible (5), keras (2), flask (1), youtube-dl (1), thefuck (1), and requests (1).

Our results are publicly available [1]. Our tool is publicly available at `https://github.com/ricardojob/OSTDetector`.

2.6 Analysis of OS-specific Tests

2.6.1 RQ1: How frequent are OS-specific tests?

In this first research question, we assess the frequency of OS-specific tests. We also explore what are the most adopted APIs to identify OSs and the most targeted OS. To identify the target OS, we mapped the possible value returned by the APIs, like *win32* for Windows and *darwin* for macOS. For example, line 512 in Figure 1 shows that `sys.platform` returns `win32`, which is mapped to Windows.

Rationale: We aim to better understand to what extent OS-specific tests happen in real-world software projects and what are the most targeted operating systems. So far, it is not clear the extension and frequency of this phenomenon. If this is common, it may bring to light novel discussions, for example, whether this is (not) a best practice and how to correct OS-specific tests properly. Moreover, it may reveal the most problematic operating systems from the test perspective. If OS-specific tests are concentrated on certain OSs, this may bring to light opportunities to overcome such limitations.

2.6.2 RQ2: Where are OS-specific tests implemented?

In our second research question, we analyze where the APIs to identify OSs are located: in test code or test decorators. When it happens in the test code, we also verify the occurrences in the test method and test support methods. When it happens in test decorators, we explore what decorators are adopted (*e.g.,* `skipif`, `xfail`, etc.).

Rationale: We aim to assess where APIs to identify OSs are most and least frequently located in tests: test code or test decorators. So far, we are not aware of where developers use this kind of API or what their preferred solution is. Using APIs to identify OSs in test code can make the test more complex, but able to run on multiple OSs and more flexible [29]. On the other hand, using APIs to identify OSs in test decorators can make the developers' intention explicit because they can detail the reason, but it makes the test less flexible. In addition, tests annotated with these decorators are flagged in the test result report, contributing to the test documentation.

2.6.3 RQ3: What code is targeted in OS-specific tests?

In this RQ, we perform a qualitative analysis to understand what code is targeted in OS-specific tests. We recall that we found 438 test files with OS-specific tests with 1,075 occurrences of APIs to identify OSs. Among those 1,075 occurrences, 372 happen in test code, while 703 occur in test decorators. We manually inspected all 372 occurrences of APIs to identify OSs that happen in *test code*. We focused on test code because APIs to identify OSs are embedded in the source code, thus, we can better analyze the tested context surrounding the API usage. For example, in Figure 1, the test code shows an OS-specific test dealing with path name manipulation. We adopted thematic analysis [11] to classify the tested context, with the following steps: (1) initial reading of the test code with OS-specific tests, (2) generating a first code for each test code, (3) searching for themes among the proposed codes, (4) reviewing the themes to find opportunities for merging, and (5) defining and naming the final themes. The first three steps were performed by the first author of the paper, while steps 4 and 5 were done together by both authors of the paper until consensus was achieved.

Rationale: We aim to explore what code is more likely to have OS-specific tests. This may reveal the kind of code that deserves more attention from developers when implementing tests for multiple operating systems. Moreover, this information may provide the basis for the proposal of dedicated solutions to improve the OS-specific tests. For example, if a certain kind of code faces a large concentration of OS-specific tests, this can indicate the lack of proper abstractions in that context.

2.6.4 RQ4: What operations are performed in OS-specific tests?

In this RQ, we conduct a qualitative analysis to understand what operations are performed in OS-specific tests to deal with multiple operating systems. As in RQ3, we focus on analyzing the *test code* because they make explicit the tasks. We also manually inspected all 372 occurrences of APIs to identify OSs that happen in *test code* and filtered out 36 cases with unclear codes. Finally, we manually classified 336 (372 - 36) relevant codes with thematic analysis [11], following the same steps described in RQ3.

Rationale: We aim to uncover operations developers perform in OS-specific tests to handle multiple OSs. This may provide the basis to reason whether these operations are best or bad programming practices that should be promoted or avoided by developers.

2.6.5 RQ5: Why are OS-specific tests implemented?

Lastly, we analyze the reasons for implementing OS-specific tests. Here, we focus on the analysis of the *test decorators* because they may contain a message in which developers can explain the reason for skipping testing on certain OS. We recall that a test decorator can be seen as a syntactic sugar for a

conditional, with the advantage of having the skip reason. That is, a test with the decorator `@skipif(sys.platform == "win32")` is equivalent to a test code with `if sys.platform != "win32"`. Figure 6 presents an example with the pytest decorator `@skipif`. We manually inspected all 703 messages in test decorators and filtered out 277 cases with generic and unclear messages. Finally, we manually classified 426 (703 - 277) relevant messages with thematic analysis [11], following the same steps described in RQ3.

Rationale: We aim to uncover the reasons behind the implementation of OS-specific tests. So far, it is unclear why developers implement OS-specific tests in the wild. Analyzing the messages provided by test decorators is an opportunity to assess the reasons as stated by the developers themselves. This information can reveal the specific issues faced by developers, opening room for the development of dedicated solutions to overcome such problems.

3 Results

3.1 RQ1: Frequency of OS-specific tests

Table 4 summarizes the frequency of OS-specific tests. We find OS-specific tests in 56 out of the 100 analyzed Python projects. These 56 projects have a total of 438 test files with OS-specific tests and 1,075 occurrences of APIs to identify OSs. On the median, each project has 3.5% of the test files with OS-specific tests and 6 occurrences of APIs to identify OSs.

Table 4: Summary of OS-specific tests.

Analyzed projects	100
Projects with OS-specific tests	56
Test files with OS-specific tests (total)	438
Test files with OS-specific tests (median, absolute)	4
Test files with OS-specific tests (median, relative)	3.5%
Occurrences of APIs to identify OSs (total)	1,075
Occurrences of APIs to identify OSs (median)	6

Table 5 details the top-10 projects with the highest percentage of test files with OS-specific tests. Pipenv (virtual environment management tool) is the project with the highest percentage: 8 out of 49 test files (16.3%) have OS-specific tests. Next, we have MkDocs (static site generator) with 16% and Horovod (distributed training framework) with 13.5%. The top-5 projects are completed with NumPy (package for scientific computing) with 12.8% and Rich (text formatting library) with 11.9%. It is worth noting that project Ray (framework for scaling AI and Python applications) has the highest absolute number of test files with OS-specific tests, 127, which represents 11.3% of test files. Many projects are libraries, frameworks, and packages. Therefore, they

are likely to be used by users in multiple operating systems, and consequently, they should be tested accordingly. Table 5 also presents the domain of the projects. Indeed, we notice multiple nontrivial domains, including packaging, machine learning, HTTP client/server, and web framework.

Table 5: Top-10 projects with most OS-specific tests.

Pos	Project	Domain	Test Files	With OS-Specific Tests	
				#	%
1	pypa/pipenv	packaging	49	8	16.3
2	mkdocs/mkdocs	documentation	25	4	16.0
3	horovod/horovod	machine learning	59	8	13.5
4	numpy/numpy	scientific computing	266	34	12.8
5	textualize/rich	text formatting	67	8	11.9
6	ray-project/ray	machine learning	1,122	127	11.3
7	aio-libs/aiohttp	HTTP client/server	64	7	10.9
8	tornadoweb/tornado	web framework	49	5	10.2
9	ipython/ipython	interactive computing	119	12	10.1
10	matplotlib/matplotlib	visualization	127	12	9.4

Next, we present the most used APIs to identify the OSs (Table 6). The 1,075 occurrences are mostly concentrated in three APIs: `sys.platform` (72.5%), `os.name` (13.4%), and `platform.system` (12.9%). Other APIs represent only 1.2% of the cases.

Table 6: Most used APIs to identify OSs.

Pos	API	#	%
1	`sys.platform`	779	72.5
2	`os.name`	144	13.4
3	`platform.system`	139	12.9
4	Others	13	1.2
All		1,075	100

Lastly, we assess what are the most targeted operating systems in the APIs to identify OSs. Table 7 shows that Windows is by far the most targeted OS: 68.2% of the APIs refer to Windows.[11] In contrast, Linux and macOS happen in only 12.3% of the cases. Other operating systems represent a minority of the cases (7.2%).

[11] Notice that the total in Table 7 is 1,118, which is higher than the 1,075 occurrences of the APIs to identify OSs. This happens because one single occurrence of an API to identify OS may target multiple OSs, *e.g.,* `sys.platform in ['linux', 'win32']`.

Table 7: Most targeted OSs.

Pos	Target OS	#	%
1	Windows	762	68.2
2	macOS	138	12.3
2	Linux	138	12.3
3	Others	80	7.2
All		1,118	100

***Summary RQ1**: OS-specific tests are common in the analyzed Python projects. We find that 56% of the analyzed systems have test files with OS-specific tests. `sys.platform` is the most used API to identify the operating system in tests, while Windows is the most targeted OS.*

3.2 RQ2: Location of OS-specific tests

In this RQ, we explore where OS-specific tests are implemented, that is, in test code (for example, in `if` blocks) or test decorators (for example, in `@skipif` decorators). Among the 1,075 occurrences of APIs to identify OSs, 372 (35%) are located in test code and 703 are in test decorators (65%). Table 8 summarizes the APIs to identify OSs by the target OS. Again, Windows is the most targeted OS both in test code and test decorators.

Table 8: Location of APIs to identify OSs by the target OS.

Pos	Target OS		Code		Decorator	
	Name	#	#	%	#	%
1	Windows	762	249	33	513	67
2	macOS	138	55	40	83	60
2	Linux	138	32	23	106	77
3	Others	80	70	87.5	10	12.5
All		1,118	406		712	

Among the API occurrences that happen in *test code*, 40.5% are located in test methods, while 59.5% are located in test support methods, like `setUp`, `tearDown`, or any other helper methods.

Considering the API occurrences that happen in *test decorators* (Table 9), we find that `@pytest.mark.skipif` (provided by pytest) is the most used to skip tests depending on the OS, with 90.61% of the cases. This high usage of pytest is in line with prior literature that suggests that pytest is currently the most used Python testing framework [4]. It is followed by the equivalent decorator in unittest `@unittest.skipIf`, with 6.83%. Other decorators such as `@xfail` and `@skipUnless` are rarely used with APIs to identify OSs.

Table 9: Most adopted decorators.

Testing Framework	Decorator	#	%
pytest	`@pytest.mark.skipif`	637	90.61
	`@pytest.mark.xfail`	6	0.85
unittest	`@unittest.skipIf`	48	6.83
	`@unittest.skipUnless`	12	1.71
All		703	100.00

Summary RQ2: *APIs to identify operating systems in tests are used more frequently in test decorators (65%) than in test code (35%). The pytest decorator* ***`@skipif`*** *is the most used to skip tests depending on the operating system.*

3.3 RQ3: Target of OS-specific tests

Our manual classification of the test code shows that OS-specific tests target mainly six categories: file/directory, third-party dependency, OS process, environment variable, permission/privilege, and network. As presented in Table 10, file/directory is the top one (36.01%), followed by third-party dependency (25.65%), and OS process (11.14%). Table 10 also details the frequency per operating system. Overall, we notice that Windows (232) is the most frequent, followed by macOS (82) and Linux (18). Next, we describe each category.

Table 10: Targets of OS-specific tests.

Category	Frequency		Operating System			
	#	%	Windows	macOS	Linux	Others
File/Directory	139	36.01	106	6	6	21
Third-Party Dependency	99	25.65	35	57	3	4
OS Process	43	11.14	33	0	1	9
Environment Variable	22	5.70	14	7	1	0
Permission/Privilege	19	4.92	9	3	0	7
Network	17	4.40	8	2	2	5
Others	47	12.17	27	7	5	8
All	386	100.00	232	82	18	54

File/Directory. This category includes OS-specific tests that handle file and directory access, like dealing with filesystem paths, I/O operations, and pathname manipulations. Figure 7 presents an OS-specific test in scikit-learn[12]

[12] File/Directory in scikit-learn: https://github.com/scikit-learn/scikit-learn/blob/00032b09c7feea08edd4486c522c2d962f9d52ec/sklearn/utils/tests/test_testing.py#L585

that verifies with the API `os.name` if the OS is not NT (*i.e.*, not the Windows family). The test then verifies if a temporary folder does not exist when it is not run on Windows (see lines 593-594 and 601-602). Figure 8 presents another OS-specific test in Cookiecutter[13] that checks the current OS with the API `sys.platform` and writes a file in the `.bat` (for Windows) or `.sh` (for other OSs, like Linux and macOS) format.

```
585  def test_tempmemmap(monkeypatch):
586      registration_counter = RegistrationCounter()
587      monkeypatch.setattr(atexit, "register", registration_counter)
588
589      input_array = np.ones(3)
590      with TempMemmap(input_array) as data:
591          check_memmap(input_array, data)
592          temp_folder = os.path.dirname(data.filename)
593      if os.name != "nt":
594          assert not os.path.exists(temp_folder)
595      assert registration_counter.nb_calls == 1
596
597      mmap_mode = "r+"
598      with TempMemmap(input_array, mmap_mode=mmap_mode) as data:
599          check_memmap(input_array, data, mmap_mode=mmap_mode)
600          temp_folder = os.path.dirname(data.filename)
601      if os.name != "nt":
602          assert not os.path.exists(temp_folder)
603      assert registration_counter.nb_calls == 2
```

Fig. 7: OS-specific test of the category *file/directory* (scikit-learn, `test_tempmemmap`).

```
33      if sys.platform.startswith('win'):
34          post = 'post_gen_project.bat'
35          with Path(hook_dir, post).open('w') as f:
36              f.write("@echo off\n")
37              f.write("\n")
38              f.write("echo post generation hook\n")
39              f.write("echo. >shell_post.txt\n")
40      else:
41          post = 'post_gen_project.sh'
42          filename = os.path.join(hook_dir, post)
43          with Path(filename).open('w') as f:
44              f.write("#!/bin/bash\n")
45              f.write("\n")
46              f.write("echo 'post generation hook';\n")
47              f.write("touch 'shell_post.txt'\n")
```

Fig. 8: OS-specific test of the category *file/directory* (Cookiecutter, `make_test_repo`).

[13] File/Directory in Cookiecutter: https://github.com/cookiecutter/cookiecutter/blob/cf81d63bf3d82e1739db73bcbed6f1012890e33e/tests/test_hooks.py#L33

Third-Party Dependency. This category contains OS-specific tests that deal with third-party dependencies, like libraries and other resources that are required but are not available on the target OS. Figure 9 presents an OS-specific test in Numpy[14] that does not import a Fortran dependency in Windows. In this case, the developer clearly states in the comment: *"we are not currently able to import the Python-Fortran interface module on Windows [...]"*.

```
64        # we are not currently able to import the Python-Fortran
65        # interface module on Windows / Appveyor, even though we do get
66        # successful compilation on that platform with Python 3.x
67        if sys.platform != "win32":
68            # check for sensible result of Fortran function; that means
69            # we can import the module name in Python and retrieve the
70            # result of the sum operation
71            return_check = import_module(modname)
72            calc_result = return_check.foo()
73            assert calc_result == 15
74            # Removal from sys.modules, is not as such necessary. Even with
75            # removal, the module (dict) stays alive.
76            del sys.modules[modname]
```

Fig. 9: OS-specific test of the category *third-party dependency* (Numpy, `test_f2py_init_compile`).

OS Process. It refers to OS-specific tests that perform operations related to creating and managing OS processes. For instance, in the test `test_cache_return_value_per_process` of MLflow, the developer states: *"Skip the following block on Windows which doesn't support os.fork"*.[15] This test uses the function `os.fork`[16] provided by the `os` module, which forks a child process and is not available on Windows. Indeed, many functions of the `os` module have limited availability. For instance, a test support method of MLflow[17] uses the functions `os.getpgid` and `os.killpg`, and, like in the previous example, there is a check for the OS: `if os.name != "nt":`.

Environment Variable. This category includes OS-specific tests that manage environment variables depending on the operating system. For example, Figure 10 presents a test support method of project Ray, which sets the

[14] Third-Party Dependency in Numpy: https://github.com/numpy/numpy/blob/2ef217d279d13afa2399efee864b9f11f4096aa7/numpy/f2py/tests/test_compile_function.py#L64-L76

[15] OS Process in MLflow: https://github.com/mlflow/mlflow/blob/7948686166b954e238d55f858c8c9194ec3c006f/tests/utils/test_process_utils.py#L48-L49

[16] https://docs.python.org/3/library/os.html#os.fork

[17] OS Process in MLflow: https://github.com/mlflow/mlflow/blob/7948686166b954e238d55f858c8c9194ec3c006f/tests/helper_functions.py#L306-L308

environment variable `SYSTEMROOT` for Windows.[18] In the test `test_from_prefixed_env_nested` of project Flask there is a check to ensure that Windows environment variables are uppercase: *"Windows env var keys are always uppercase"*.[19] In project Horovod, the environment variable `OBJC_DISABLE_INITIALIZE_FORK_SAFETY` is required to initialize the test in the macOS correctly. In this case, the developer commented: *"Spark will fail to initialize correctly locally on Mac OS without this"*.[20]

```
14 ⌄ def build_env():
15        env = os.environ.copy()
16        if sys.platform == "win32" and "SYSTEMROOT" not in env:
17            env["SYSTEMROOT"] = r"C:\Windows"
18
19        return env
```

Fig. 10: OS-specific test of the category *environment variable* (Ray, `build_env`).

Permission/Privilege. It contains OS-specific tests that need to deal with permissions and privileges, for instance, to execute blocks of code. For example, Figure 11 presents the test `test_renew_files_propagate_permissions` of project Certbot, which sets distinct permissions depending on the target OS.[21] Similarly, in project Poetry, the developer explains that a certain call requires privileges for execution: *"os.symlink requires either administrative privileges or developer mode on Win10 [...]"*.[22]

Network. It includes OS-specific tests that handle networking and connections. For example, the test `test_connection_refused` of Salt performs special checks for connection refused in Windows. The developer comments: *"This is usually "Connection refused". On Windows, strerror is broken and returns "Unknown error""*.[23] In project locust, the test `test_web_options` sets the

[18] Environment Variable in Ray: https://github.com/ray-project/ray/blob/10861d9f2ef19e845186b8925053a11c6812a161/python/ray/tests/test_tls_auth.py#L14-L19

[19] Environment Variable in Flask: https://github.com/pallets/flask/blob/daf6966c89b280725439d2951beb88640c473154/tests/test_config.py#L90-L98

[20] Environment Variable in Horovod: https://github.com/horovod/horovod/blob/f356349204f05ca9256c33dc4d4831798a0b8479/test/integration/test_spark.py#L66-L68

[21] Permission/Privilege in Certbot: https://github.com/certbot/certbot/blob/5149dfd96e9b57b98551670c203d7b5f986a9c32/certbot-ci/certbot_integration_tests/certbot_tests/test_main.py#L269-L278

[22] Permission/Privilege in Poetry: https://github.com/python-poetry/poetry/blob/d5f83fffc9c5813a589f7ef928fe31549171954e/tests/packages/test_locker.py#L1136-L1139

[23] Network in Salt: https://github.com/saltstack/salt/blob/2bd55266c8ecc929a3a0a9aec1797a368c521072/salt/ext/tornado/test/simple_httpclient_test.py#L341-L348

```
269        if os.name != 'nt':
270            # On Linux, read world permissions + all group permissions
271            # will be copied from the previous private key
272            assert_world_read_permissions(privkey2)
273            assert_equals_world_read_permissions(privkey1, privkey2)
274            assert_equals_group_permissions(privkey1, privkey2)
275        else:
276            # On Windows, world will never have any permissions, and
277            # group permission is irrelevant for this platform
278            assert_world_no_permissions(privkey2)
```

Fig. 11: OS-specific test of the category *permission/privilege* (Certbot, `test_renew_files_propagate_permissions`).

interface to 127.0.0.1. In this case, the developer explains: *"MacOS only sets up the loopback interface for 127.0.0.1 and not for 127.*.*.*"*.[24]

Other. We also find six other categories that occur less frequently in our dataset: *(i)* **Numeric Precision** (14 out of 386) handles numeric precision-related issues;[25] *(ii)* **Timeout** (11 out of 386) sets timeout/sleep values to wait for some execution;[26] *(iii)* **Parallelism** (8 out of 386) manages parallelism or multiprocessing;[27] *(iv)* **Timezone** (6 out of 386) handles local time and UTC;[28] *(v)* **Memory Allocation** (6 out of 386) manages memory;[29] and *(vi)* **Location** (2 out of 386) provides locale-related treatments.[30]

> ***Summary RQ3***: *OS-specific tests target a diversity of code, including file/directory, OS process, network, permission/privilege, numeric precision, and timezone. The most frequent code targeted by OS-specific tests are file/directory (36.01%), third-party dependency (25.65%), and OS process (11.14%).*

[24] Network in locust: https://github.com/locustio/locust/blob/5fc187cd7d387d53ccee43c6e187e3d10520c8d5/locust/test/test_main.py#L767-L769

[25] Numeric Precision in Numpy: https://github.com/numpy/numpy/blob/2ef217d279d13afa2399efee864b9f11f4096aa7/numpy/f2py/tests/test_array_from_pyobj.py#L164

[26] Timeout in Mypy: https://github.com/python/mypy/blob/a9ee618f3a941098b24156eb499db5684fcfc261/mypyc/test/test_run.py#L179

[27] Parallelism in scikit-learn: https://github.com/scikit-learn/scikit-learn/blob/00032b09c7feea08edd4486c522c2d962f9d52ec/sklearn/decomposition/tests/test_sparse_pca.py#L142

[28] Timezone in Ansible: https://github.com/ansible/ansible/blob/a84b3a4e7277084466e43236fa78fc99592c641a/test/support/integration/plugins/modules/timezone.py#L107

[29] Memory Allocation in Ray: https://github.com/ray-project/ray/blob/10861d9f2ef19e845186b8925053a11c6812a161/python/ray/tests/test_advanced_9.py#L143

[30] Location in Gooey: https://github.com/chriskiehl/gooey/blob/be4b11b8f27f500e7326711641755ad44576d408/gooey/tests/__init__.py#L48

3.4 RQ4: Operations found in OS-specific tests

Our manual classification of the test code shows that developers perform five operations in OS-specific tests: set OS-specific value, skip test, call OS-specific API, mock OS-specific object, and suspend execution. As presented in Table 11, set OS-specific value is the most frequent (62.68%), followed by skip test (17.66%), and call OS-specific API (13.39%). Table 11 also presents the frequency per OS. Overall, we note that Windows (233) is the most frequent, followed by macOS (38) and Linux (22). Next, we describe each operation.

Table 11: Operations found in OS-specific tests.

Category	Frequency		Operating System			
	#	%	Windows	macOS	Linux	Others
Set OS-Specific Value	220	62.68	145	28	17	30
Skip Test	62	17.66	37	8	3	14
Call OS-Specific API	47	13.39	35	2	1	9
Mock OS-Specific Object	12	3.42	8	0	0	4
Suspend Execution	10	2.85	8	0	1	1
All	351	100.00	233	38	22	58

Set OS-Specific Value. This case occurs when the test checks the operating system and sets a specific value depending on the OS to ensure the test's correct execution. For example, Figure 12 presents an OS-specific test in Django[31] that verifies if the OS is NT (*i.e.,* the Windows family) and sets different values for the path variable `cwd_prefix`. Similarly, in project Numpy,[32] multiple command values are set according to the target OS.

```
70            if os.name == "nt":
71                # #: .\path\to\file.html:123
72                cwd_prefix = "%s%s" % (os.curdir, os.sep)
73            else:
74                # #: path/to/file.html:123
75                cwd_prefix = ""
```

Fig. 12: OS-specific test with the operation *Set OS-Specific Value* (Django, `test_extraction`).

Skip Test. This case happens when the test verifies the OS and then skips it, not executing the test in the current OS. We find three main ways to apply this operation in the test code. Developers skip the test with `return` and `raise`

[31] Set OS-Specific Value in Django: https://github.com/django/django/blob/2eb1f37260f0e0b71ef3a77eb5522d2bb68d6489/tests/i18n/test_extraction.py#L70
[32] Set OS-Specific Value in Numpy: https://github.com/numpy/numpy/blob/2ef217d279d13afa2399efee864b9f11f4096aa7/numpy/tests/test_scripts.py#L18

statements or call **skip** APIs provided by the testing framework directly in the test code. Figure 13 presents an example in project Youtube-dl[33] with a `return` statement, while Figure 14 presents an example in Sanic[34] with a `raise` statement. Figure 15 shows a case of skip in project Numpy.[35]

```
212     def test_sanitize_path(self):
213         if sys.platform != 'win32':
214             return
215
```

Fig. 13: OS-specific test with the operation *skip test* (Youtube-dl, `test_utils`).

```
16  def double_dotted_directory_file(static_file_directory: str):
17      """Generate double dotted directory and its files"""
18      if sys.platform == "win32":
19          raise Exception("Windows doesn't support double dotted directories")
```

Fig. 14: OS-specific test with the operation *skip test* (Sanic, `test_static`).

```
13  def get_module(tmp_path):
14      """ Add a memory policy that returns a false pointer 64 bytes into the
15      actual allocation, and fill the prefix with some text. Then check at each
16      memory manipulation that the prefix exists, to make sure all alloc/realloc/
17      free/calloc go via the functions here.
18      """
19      if sys.platform.startswith('cygwin'):
20          pytest.skip('link fails on cygwin')
21      if IS_WASM:
22          pytest.skip("Can't build module inside Wasm")
```

Fig. 15: OS-specific test with the operation *skip test* (Numpy, `test_mem_policy`).

Call OS-Specific API. This case happens when the test checks the current OS and calls an OS-specific API. Figure 16 presents an example in Matplotlib[36] that calls the API `win32api.GenerateConsoleCtrlEvent` for Win-

[33] Skip Test in Youtube-dl: https://github.com/ytdl-org/youtube-dl/blob/213d1d91bfc4a00fefc72fa2730555d51060b42d/test/test_utils.py#L213

[34] Skip Test in Sanic: https://github.com/sanic-org/sanic/blob/af678010628cd76a57e7a53e114f25d5c00e931a/tests/test_static.py#L18

[35] Skip Test in Numpy: https://github.com/numpy/numpy/blob/2ef217d279d13afa2399efee864b9f11f4096aa7/numpy/core/tests/test_mem_policy.py#L19

[36] Call OS-Specific API in Matplotlib: https://github.com/matplotlib/matplotlib/blob/e8101f17d8a7d2d7eccff7452162c02a27980800/lib/matplotlib/tests/test_backend_qt.py#L93

dows and other APIs to Unix-like OSs, such as `os.kill`, `os.getpid`, and `signal.SIGINT`. This special treatment is required to properly handle the keyboard interrupt event `CTRL+C` both in Windows and Unix-like OSs. Figure 17 presents an example in Django[37] that checks the current OS and calls distinct path APIs for Windows (`WindowsPath`) and Unix-like (`PosixPath`).

```
92      def interrupter():
93          if sys.platform == 'win32':
94              import win32api
95              win32api.GenerateConsoleCtrlEvent(0, 0)
96          else:
97              import signal
98              os.kill(os.getpid(), signal.SIGINT)
99
100     target = getattr(plt, target_name)
101     timer = threading.Timer(1, interrupter)
```

Fig. 16: OS-specific test with the operation *call OS-specific API* (Matplotlib, `test_backend_qt`).

```
508         # Concrete path objects work on supported platforms.
509         if sys.platform == "win32":
510             self.assertSerializedEqual(pathlib.WindowsPath.cwd())
511             path = pathlib.WindowsPath("A:\\File.txt")
512             expected = ("pathlib.PureWindowsPath('A:/File.txt')", {"import pathlib"})
513             self.assertSerializedResultEqual(path, expected)
514         else:
515             self.assertSerializedEqual(pathlib.PosixPath.cwd())
516             path = pathlib.PosixPath("/path/file.txt")
517             expected = ("pathlib.PurePosixPath('/path/file.txt')", {"import pathlib"})
518             self.assertSerializedResultEqual(path, expected)
```

Fig. 17: OS-specific test with the operation *call OS-specific API* (Django, `test_writer`).

Mock OS-Specific Object. This operation occurs when the test verifies the current OS and performs mock-related operations to emulate missing or hard to test objects [29, 31, 37, 38]. For example, Figure 18 presents an OS-specific test in IPython[38] that mocks the modules `os.system` in Windows and `subprocess.call` in Unix-like OSs. We find occurrences of this operation in

[37] Call OS-Specific API in Django: https://github.com/django/django/blob/2eb1f37260f0e0b71ef3a77eb5522d2bb68d6489/tests/migrations/test_writer.py#L509-L518

[38] Mock OS-Specific Object in IPython: https://github.com/ipython/ipython/blob/a418f38c4f96de1755701041fe5d8deffbf906db/IPython/core/tests/test_interactiveshell.py#L667

multiple large projects such as Pipenv,[39] Ray,[40] and Numpy.[41] In the comments, the developers indicate the reason to mock: (1) *"Emulate [...] to check that we handle both cases correctly"*, (2) *"Win impl relies on kbhit() instead of select() so the pipe trick won't work."*, and (3) *"Context manager to emulate os.name != 'posix' "*.

```
665     @pytest.mark.parametrize("magic_cmd", ["pip", "conda", "cd"])
666     def test_magic_warnings(magic_cmd):
667         if sys.platform == "win32":
668             to_mock = "os.system"
669             expected_arg, expected_kwargs = magic_cmd, dict()
670         else:
671             to_mock = "subprocess.call"
672             expected_arg, expected_kwargs = magic_cmd, dict(
673                 shell=True, executable=os.environ.get("SHELL", None)
674             )
675
676         with mock.patch(to_mock, return_value=0) as mock_sub:
677             with pytest.warns(Warning, match=r"You executed the system command"):
678                 ip.system_raw(magic_cmd)
679             mock_sub.assert_called_once_with(expected_arg, **expected_kwargs)
```

Fig. 18: OS-specific test with the operation *mock OS-specific object* (IPtyhon, `test_interactiveshell`).

Suspend Execution. It happens when the test checks the OS and suspends execution for a given number of seconds, with methods like `time.sleep`. Figure 19 presents an example in project Mypy[42] in which the API `time.sleep` is used to suspend the execution as a workaround for Linux platforms. Notice that the developer recognizes the issue: *"Figure out a better approach, since this slows down tests"*. Similarly, in project Ray, the developer suspends execution to avoid flaky tests: *"Set to 40s on Windows and 20s on other platforms to avoid flakiness"*.[43]

[39] Mock OS-Specific Object in Pipenv: https://github.com/pypa/pipenv/blob/babd428d8ee3c5caeb818d746f715c02f338839b/pipenv/patched/pip/_vendor/colorama/tests/initialise_test.py#L130
[40] Mock OS-Specific Object in Ray: https://github.com/ray-project/ray/blob/10861d9f2ef19e845186b8925053a11c6812a161/python/ray/tests/test_usage_stats.py#L524
[41] Mock OS-Specific Object in Numpy: https://github.com/numpy/numpy/blob/2ef217d279d13afa2399efee864b9f11f4096aa7/numpy/distutils/tests/test_exec_command.py#L74
[42] Suspend Execution in Mypy: https://github.com/python/mypy/blob/a9ee618f3a941098b24156eb499db5684fcfc261/mypyc/test/test_run.py#L179
[43] Suspend Execution in Ray: https://github.com/ray-project/ray/blob/10861d9f2ef19e845186b8925053a11c6812a161/python/ray/tests/test_runtime_env_working_dir_3.py#L26

```
170            for operations in steps:
171                # To make sure that any new changes get picked up as being
172                # new by distutils, shift the mtime of all of the
173                # generated artifacts back by a second.
174                fudge_dir_mtimes(WORKDIR, -1)
175                # On Ubuntu, changing the mtime doesn't work reliably. As
176                # a workaround, sleep.
177                #
178                # TODO: Figure out a better approach, since this slows down tests.
179                if sys.platform == "linux":
180                    time.sleep(1.0)
```

Fig. 19: OS-specific test with the strategy *suspend execution* (Mypy, `test_run`).

> ***Summary RQ4***: *We find that developers perform five operations in OS-specific tests: set OS-specific value (62.68%), skip test (17.66%), call OS-specific API (13.39%), mock OS-specific object (3.42%), and suspend execution (2.85%).*

3.5 RQ5: Reasons to implement OS-specific tests

In this final RQ, we manually classified the reasons found in test decorators for implementing OS-specific tests. As presented in Table 12, the most common reasons are unavailable external resources (29.58%), unsupported standard library (18.08%), and flaky test (13.85%). Other rationales include distinct path convention (11.74%), required CI/CD environment (10.56%), and unsupported file operation (6.34%). Table 12 details the frequency per operating system. Overall, we see that Windows (285) is the most frequent, followed by macOS (71) and Linux (59). Next, we detail each rationale.

Table 12: Reasons to implement OS-specific tests according to developers.

Reasons	**Frequency**		**Target OS**			
	#	%	Windows	macOS	Linux	Others
Unavailable External Resource	126	29.58	62	46	17	1
Unsupported Standard Library	77	18.08	68	0	0	9
Flaky Test	59	13.85	36	1	22	0
Distinct Path Convention	50	11.74	50	0	0	0
Required CI/CD Environment	45	10.56	6	24	15	0
Unsupported File Operation	27	6.34	27	0	0	0
Others	42	9.86	36	0	5	1
All	426	100.00	285	71	59	11

Unavailable External Resource. In this rationale, the developers explain that certain external resources (like libraries, frameworks, and tools) are re-

quired to execute the tests, but they are not available on the target OS. For example, in the projects Ansible and Ray, the developers directly mention the required dependencies: *"macOS requires passlib"* [44] and *"No py-spy on Windows"*. [45] Passlib is a password hashing library, while py-spy is a sampling profiler. Also in project Ray, the developer explains that the tested feature is not supported on the target OS due to the unavailable database: *"Feature not supported Windows because Redis is not officially supported by Windows"*. [46] Redis is an in-memory database.

Unsupported Standard Library. In this rationale, the developers indicate that some APIs provided by the Python Standard Library are not supported on the target OS. Indeed, several APIs provided by the Python Standard Library have availability restrictions, mainly in low-level modules, like `os`,[47] `asyncio`,[48] and `socket`.[49] Among the 77 occurrences of this category, 70 are related to limitations on Windows. For example, in the project Loguru, the developer explains: *"Windows does not support forking"*, [50] while in project Ray, the developer states: *"Windows signal handling not compatible"*. [51]

Flaky Test. In this rationale, the developers directly state that the test method is flaky (non-deterministic) on the target OS. It is important to note that flaky tests are a problem because they damage regression testing and their failures can be hard to reproduce due to the non-determinism [17, 27]. For example, in project Ray, the developer simply states: *Flaky on Mac. Issue #27562.* [52]

Distinct Path Convention. Different operating systems have distinct path name conventions. The difference happens mainly between Windows and Unix-style paths. In this category, all 50 instances are related to Windows. For example, in project Sanic, the developer states a path convention that is not supported in Windows: *"Windows does not support double dotted directories"*. [53]

[44] Unavailable External Resource in Ansible: https://github.com/ansible/ansible/blob/a84b3a4e7277084466e43236fa78fc99592c641a/test/units/utils/test_encrypt.py#L177

[45] Unavailable External Resource in Ray: https://github.com/ray-project/ray/blob/10861d9f2ef19e845186b8925053a11c6812a161/python/ray/tests/test_dashboard_profiler.py#L99

[46] Unavailable External Resource in Ray: https://github.com/ray-project/ray/blob/10861d9f2ef19e845186b8925053a11c6812a161/python/ray/tests/test_multi_node_3.py#L534

[47] https://docs.python.org/3/library/os.html

[48] https://docs.python.org/3/library/asyncio.html

[49] https://docs.python.org/3/library/socket.html

[50] Unsupported Standard Library in Loguru: https://github.com/delgan/loguru/blob/a4a6264d4b7ac4822a49244f72f8ee8995497dba/tests/test_multiprocessing.py#L597

[51] Unsupported Standard Library in Ray: https://github.com/ray-project/ray/blob/10861d9f2ef19e845186b8925053a11c6812a161/python/ray/tests/test_cli.py#L480

[52] Flaky Test in Ray: https://github.com/ray-project/ray/blob/10861d9f2ef19e845186b8925053a11c6812a161/python/ray/tests/test_runtime_env_working_dir_4.py#L22

[53] Distinct Path Convention in Sanic: https://github.com/sanic-org/sanic/blob/af678010628cd76a57e7a53e114f25d5c00e931a/tests/test_static.py#L620

Other cases are related to limited number of characters: "*Windows supports at most 260 characters in a path*"[54] and unsupported special characters: "*Windows does not support '*' in filename*".[55]

Required CI/CD Environment. In this rationale, the developers indicate that a certain CI/CD environment is required to execute the tests. For example, in project Ray, the developer explains that the test method must be run on a Linux CI machine: *"Requires PR wheels built in CI, so only run on Linux CI machines"*. [56]

Unsupported File Operation. This rationale comprises different unsupported file operations on the target OS, such as deleting, moving, and creating files and directories. For example, in project Django, the developer states that open files cannot be moved in Windows: *"Windows doesn't support moving open files"*. [57]

Others. Other rationales occur less frequently in our dataset, for example, explanations related to performance and runtime issues.

> ***Summary RQ5***: *OS-specific tests are implemented mostly to overcome unavailable external resources (29.58%), unsupported standard libraries (18.08%), and flaky tests (13.85%). Other reasons include distinct path convention (11.74%), required CI/CD environment (10.56%), and unsupported file operation (6.34%).*

4 Discussion and Implications

OS-specific tests and test smells. Overall, we find that OS-specific tests are common in the analyzed Python projects. For instance, RQ1 presented that 56% of the analyzed projects have test files with OS-specific tests. The literature reports that *tests with conditional logic* is a test that contains code that may or may not be executed. It is a test smell because branches within the test method will change the test behavior and make the test harder to understand and maintain [29, 32, 33]. Thus, we can see OS-specific tests that rely on conditional logic as a test smell because they contain branches that will change test behavior according to the executed OS. The root cause of this smell is the so-called *flexible test*, that is, a test that verifies different functionality

[54] Distinct Path Convention in Django: https://github.com/django/django/blob/2eb1f37260f0e0b71ef3a77eb5522d2bb68d6489/tests/file_storage/tests.py#L873

[55] Distinct Path Convention in Loguru: https://github.com/delgan/loguru/blob/a4a6264d4b7ac4822a49244f72f8ee8995497dba/tests/test_filesink_retention.py#L195

[56] Required CI/CD Environment in Ray: https://github.com/ray-project/ray/blob/10861d9f2ef19e845186b8925053a11c6812a161/python/ray/tests/test_runtime_env_conda_and_pip_4.py#L91

[57] Unsupported File Operation in Django: https://github.com/django/django/blob/2eb1f37260f0e0b71ef3a77eb5522d2bb68d6489/tests/file_storage/test_inmemory_storage.py#L53

depending on when or where it is run [29], like distinct operating systems. OS-specific tests may also lead to *rotten green tests*, that is, tests that pass because some or all of its assertions are not actually executed [3,16]. We have manually analyzed the OS-specific tests and detected 51 instances of *rotten green tests*, for example, in projects scikit-learn,[58] Numpy,[59] and Bokeh.[60] Therefore, we show that OS-specific tests may give us false confidence that the code under test is valid when, in fact, that code may not have been tested at all [3,16].

OS-specific tests and CI/CD. It is important to notice that OS-specific tests may be executed in CI/CD environments or containers. For example, GitHub Actions, the most popular CI/CD environment [15], can execute code in Ubuntu Linux, Windows, and macOS operating systems. Therefore, CI/CD environments would allow the developers to run OS-specific tests across different operating systems to cover all logic. We have manually inspected the 56 projects that contain OS-specific tests and verified whether they were using GitHub Actions (*i.e.,* include the `.github/workflows` directory). We found that 48 projects rely on GitHub Actions and have testing configurations. Among those, 30 projects tested in three OSs (Linux, Windows, and macOS), 13 projects tested in two OSs, and 5 projects tested in only one OS. Therefore, at least 30 of 54 projects have CI/CD environments properly configured to run the tests in three operating systems, minimizing the risks of *rotten green tests*. Notice, however, that locally the problem persists: developers may not be able to fully execute OS-specific tests on their local machines because multiple OSs would be required.

OS-specific tests and technical debt. Technical debt is used to express sub-optimal source code implementations that are introduced for short-term benefits and that should be fixed later [34,36]. Among the categories detected in our qualitative analysis, we found that some are related to sub-optimal implementations. For example, RQ4 presented that developers may *suspend execution* in OS-specific tests, which can slow down the tests. Indeed, RQ5 showed that one reason to implement OS-specific tests is to overcome *flaky tests*. Motivated by these findings, we explored whether developers themselves recognize OS-specific tests as technical debt, the so-called *self-admitted technical debt* (SATD) [34,36]. We analyzed the test comments of the OS-specific tests, looking for terms such as "fixme", "todo", and "workaround". We found 15 occurrences of self-admitted technical debt in OS-specific tests. For example, in project Mypy, the developer comments: *"TODO: Figure out a bet-*

[58] *rotten green tests* in scikit-learn: https://github.com/scikit-learn/scikit-learn/blob/00032b09c7feea08edd4486c522c2d962f9d52ec/sklearn/utils/tests/test_testing.py#L593

[59] *rotten green tests* in Numpy: https://github.com/numpy/numpy/blob/2ef217d279d13afa2399efee864b9f11f4096aa7/numpy/distutils/tests/test_misc_util.py#L64

[60] *rotten green tests* in Bokeh: https://github.com/bokeh/bokeh/blob/49bf94929297af7ee8a6bd2b3283c380be5e117d/tests/unit/bokeh/util/test_browser.py#L97

ter approach, since this slows down tests".[61] In Pipenv, the developer states: "*TODO: Implement this message for Windows*".[62] In Matplotlib the developer notes: *"FIXME: This should be enabled everywhere once Qt5 is fixed on macOS to not resize incorrectly"*.[63] Therefore, we provide evidence that some OS-specific tests can indicate maintenance problems and are sub-optimal implementations that should be fixed later, like any other technical debt.
Efforts to properly test on Windows. In RQ1, we detected that Windows is the most targeted operating system in OS-specific tests: 68.2% of the APIs to identify the OS refer to Windows. In RQ2, we found that Windows is the most targeted OS both in test code and test decorators. Moreover, RQs 3, 4, and 5 show that Windows is the most targeted operating system. Overall, we presented dozens of concrete cases of Windows-related problems, for example:

- Numpy: *"we are not currently able to import the Python-Fortran interface module on Windows [...]"*.
- Sanic: *"Windows does not support double dotted directories"*.
- Django: *"Windows doesn't support moving open files"*.
- Salt: *"This is usually "Connection refused". On Windows, strerror is broken and returns "Unknown error""*.
- Ray: *"Feature not supported Windows because Redis is not officially supported by Windows"*.

Our results shed light on the numerous issues faced by developers when testing on Windows, showing that testing on this OS seems more challenging than testing on Unix-like OSs. Therefore, projects that target multiple OSs should be aware of the caveats to test on Windows properly. As a concrete issue, we found that OS-specific tests are commonly related to path convention issues between Windows and Unix: 100% (50 out of 50) of the OS-specific tests related to *distinct path convention* happen for Windows (RQ5). Based on our findings, we recommend that developers rely on OS-independent APIs to fix some OS-specific tests related to path conventions. For this purpose, the Python Standard Library provides OS-independent APIs that can handle subtle OS differences. For example, the `os` module provides a set of APIs that support path manipulation, so the developers do not need to hard-code OS information in the test. The API `os.sep`[64] returns the character used by the OS to separate pathname components, which is "/" for Unix and "\\" for Windows paths. Similarly, the API `os.linesep`[65] provides the string used to separate lines, such as "\n" for Unix and "\r\n" for Windows. Other kinds of OS-specific tests may have particular solutions.

[61] *self-admitted technical debt* in Mypy: https://github.com/python/mypy/blob/a9ee618f3a941098b24156eb499db5684fcfc261/mypyc/test/test_run.py#L179
[62] *self-admitted technical debt* in Pipenv: https://github.com/pypa/pipenv/blob/babd428d8ee3c5caeb818d746f715c02f338839b/tests/integration/test_run.py#L45
[63] *self-admitted technical debt* in Matplotlib: https://github.com/matplotlib/matplotlib/blob/e8101f17d8a7d2d7eccff7452162c02a27980800/lib/matplotlib/tests/test_backends_interactive.py#L185
[64] https://docs.python.org/3/library/os.html#os.sep
[65] https://docs.python.org/3/library/os.html#os.linesep

Skipping flaky tests on certain OSs instead of fixing the non-determinism. Flaky tests are non-deterministic tests that damage regression testing and their failures can be hard to reproduce [17, 22, 27]. RQ4 presented that developers may *suspend execution* in OS-specific tests for a given number of seconds to avoid flaky tests. In RQ5, we found that one important reason to implement OS-specific tests is to overcome *flaky tests*. That is, developers fully skip the test execution on an operating system to avoid non-determinism. It is important to recall this is not a best practice, that is, instead of properly testing on the target OS or fixing the non-deterministic issue, developers opt to simply not test on a particular OS. Consequently, this may lead to a false confidence that the tested feature is valid, but, in fact, it is untested on a certain OS. Thus, we recommend that future research on flaky tests should take into account OS-specific needs as a key aspect that may contribute to non-determinism. For example, researchers can rely on the APIs to identify OSs (such as the ones presented in Table 2) to identify signs of non-determinism and look for explanations on test decorators (such as the ones presented in Table 3) to detect rationales.

Refactoring OS-specific tests. The literature presents that a general solution to fix *tests with conditional logic* includes refactoring tests into separate modules (*e.g.,* one for each operating system), making each test module fully executable on the target OS [3, 16, 29]. Considering the specific case of OS-specific tests, RQ2 presented that test decorators (65%) are used more frequently in OS-specific tests than in test code (35%), while RQ4 showed that developers may skip the test with commands embedded in the test code, for example, using `return` and `raise` statements. It is important to recall that tests annotated with `@skipif` decorators are flagged in the test result report (contributing to the test documentation) and avoid using OS checks directly in the test code (making the test less complex). Moreover, using test decorators can make the developers' intention explicit because developers can detail the reason for skipping the test on a certain OS. Thus, another possible refactoring is transforming OS-specific tests that check the OS in the *test code* into their *test decorators* counterparts, whenever possible. For example, Figure 13 shows that the test `test_sanitize_path`[66] checks the OS and returns, not executing the test for Unix-like OSs. This test could be safely refactored to use the `@skipif` decorator instead, benefiting from the advantages of test decorators. In this case, the OS verification in the test code could be replaced by the test decorator `@skipif(sys.platform != "win32")`. We detected multiple similar cases, for example, in projects Ray[67] and IPython.[68] Thus, we envision that

[66] Possible refactoring in Youtube-dl: https://github.com/ytdl-org/youtube-dl/blob/213d1d91bfc4a00fefc72fa2730555d51060b42d/test/test_utils.py#L213

[67] Possible refactoring in Ray: https://github.com/ray-project/ray/blob/10861d9f2ef19e845186b8925053a11c6812a161/python/ray/tests/test_tempfile.py#L78

[68] Possible refactoring in IPython: https://github.com/ipython/ipython/blob/a418f38c4f96de1755701041fe5d8deffbf906db/IPython/core/tests/test_interactiveshell.py#L571

future research in the context of refactoring could leverage this information to detect novel refactoring opportunities and safely transform the test code.

5 Threats to Validity

5.1 Construct Validity

Selection of APIs to identify OSs. We manually inspected the Python Standard Library documentation to detect API to identify OSs. We ended up selecting 15 APIs from three modules (`sys`, `os`, and `platform`), which provide a set of interfaces to identify the current OSs and platforms. As this selection is purely based on reading the official Python documentation, the risks of false positives (wrong APIs) and false negatives (missing APIs) are reduced.

Selection of the projects. We target the analysis of real-world and popular projects. For this purpose, we select the top-100 most popular Python software systems hosted on GitHub according to the number of stars, which is a metric commonly used in the software mining literature as a proxy of popularity [8,9]. We recall that we excluded three types of repositories: (1) forked repositories, (2) repositories without tests, and (3) tutorials, examples, and sample projects. Therefore, we minimize the risks of analyzing less popular projects.

5.2 Conclusion Validity

Number of analyzed projects. We analyzed 100 popular and real-world Python software systems. These systems are credible and relevant, as they were selected based on the GitHub star metric [8, 9]. Despite these observations, we encourage future research to reproduce our experiments in a larger collection of projects to have an even more precise overview of the OS-specific tests.

Number analyzed OS-specific tests. In RQ1 and RQ2, we quantitatively analyzed *all* instances of OS-specific tests detected in the 100 projects. Similarly, in RQ3, RQ4, and RQ5, we analyzed *all* OS-specific tests that happened in test codes (372 cases) and test decorators (703 cases). That is, for these quantitative and qualitative analyses, it was not needed to select a sample since all instances were analyzed. For a large number of tests, we recommend that researchers should randomly select statistically significant samples.

5.3 Internal Validity

Manual classification of test code (RQ3 and RQ4). In RQ3 and RQ4, we manually analyzed all 372 occurrences of APIs to identify OSs in test code to derive the target and operations. We recall that we only analyzed the test code because those are the cases in which the target APIs are used within the

source code, thus, we have more context about the API usage. To minimize the subjectivity of the manual classification, we adopted thematic analysis [12].

Manual classification of messages in test decorators (RQ5). In RQ5, we also manually classified the messages found in test decorators. We started with 703 messages and took special care to filter out 277 generic and unclear messages that did not properly explain the reason to skip the test on a certain OS (*e.g.,* vague messages like "*Fail on Windows*"). As in RQ3, we also relied on thematic analysis [12] to reduce the subjectiveness.

5.4 External Validity

OS-specific tests in other programming languages. In this paper, we explored OS-specific tests in Python. However, OS-specific tests may occur in any other programming language that provides APIs to identify the operating systems. For example, in JavaScript, developers can use the APIs `navigator.userAgent`[69] and `navigator.platform`[70] to identify the platform on which the user's browser is running, while Node.js provides the API `process.platform`.[71] In Java, the developers can use the API `System.getProperty("os.name")`,[72] while Go provides the API `runtime.GOOS`.[73] Notice that these APIs are provided by the standard libraries, meaning they are easily accessible to developers. Therefore, the investigated phenomenon may happen not only in Python but also in other programming languages, such as JavaScript, Java, and Go, to name a few. Moreover, other languages, such as C/C++ and Go, may allow OS identification at compile time. For example, C/C++ has predefined compile macros about the OSs,[74] while Go has build constraints.[75] However, these compile-time solutions are more related to building the application for a target OS (*i.e.,* not testing the application), thus, they are not in the scope of our study.

Generalization of the results. In this study, we analyzed 100 real-world Python systems. These projects are among the most popular in the Python ecosystem (*e.g.,* Django, scikit-learn, and Flask, to name a few), thus, they are relevant and credible systems. Despite these observations, our findings–as usual in empirical software engineering–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 and programming languages.

[69] https://developer.mozilla.org/en-US/docs/Web/API/Navigator/userAgent
[70] https://developer.mozilla.org/en-US/docs/Web/API/Navigator/platform
[71] https://nodejs.org/api/process.html#process_process_platform
[72] https://docs.oracle.com/javase/tutorial/essential/environment/sysprop.html
[73] https://pkg.go.dev/runtime#GOOS
[74] https://github.com/cpredef/predef/blob/master/OperatingSystems.md
[75] https://pkg.go.dev/cmd/go#hdr-Build_constraints

6 Related Work

Software testing is an essential practice in modern software development, which ensures quality and sustainable software evolution [6, 18, 19, 25]. However, creating and maintaining a good test suite can be challenging, as it must evolve to accommodate new tests and prevent regressions in complex systems over time [43]. Due to this complexity, test suites may experience test smells [5, 20] or flakiness [21, 27]. Indeed, one research topic that is directly related to our study is the area of test smells. Test smells are bad programming practices and can indicate potential design problems in test code [41]. Previous studies have analyzed the impact of these practices on maintenance [5, 30], as well as the impact of refactoring test smell on internal quality attributes from a developer perspective [14]. For example, Bavota *et al.* [5] presented two empirical studies to analyze the presence of test smells and their impact on software maintenance. The main results indicate that test smells are widely spread across the analyzed software systems and have a strong negative impact on the comprehensibility of test suites and production code. In addition to the negative impact, Palomba *et al.* [30] showed that occurrences of bad smells are often found with different types of design problems, besides being related to system attributes such as size or number of classes. Likewise, OS-specific tests can indicate problems related to maintenance, as detailed in our discussion of test smell and technical debt. Aljedaani *et al.* [2] conducted a Systematic Mapping Study about test smell detection tools. In practice, multiple tools are proposed to detect test smells, such as TestQ [10], tsDetect [33], PyNose [42], Pytest-Smell [7], and SniffTest [28]. We could not find any test smell related to the presence of OS-specific tests on the publicly available tools.

Several studies provide catalogs of test smells [41] [32], for example, *Conditional Test Logic* [32], *Rotten Green Tests* [3, 16], and *Ignored Tests* [32]. *Conditional Test Logic* is a test that includes conditional test logic, or one or more control statements [32]. *Rotten Green Tests* refer to tests that include some assertions that may not be executed [3, 16]. This happens usually when a condition branch is bound to assertions, but it is not run. In fact, OS-specific tests may lead to both *Conditional Test Logic* and *Rotten Green Tests*, as discussed in Section 4. *Ignored Tests* occurs when the developer suppresses the test execution. In particular, it is studied in the Java language and happens when a test method or class contains the `@Ignore` annotation [32]. In RQ5, we presented multiple reasons that we found in test decorators like `@skipif`. Those skipping decorators are used to skip the tests based on certain conditions (like the running OS), differently from *Ignored Tests* which fully ignore and skip the test.

Another research topic related to ours is flaky tests [17, 21, 27, 39]. Test flakiness is related to the non-deterministic behavior of software tests. In practice, these tests may create multiple problems during regression testing and their failures can be hard to reproduce. Luo *et al.* [27] conducted the first extensive empirical study on flaky tests by analyzing 51 Apache open-source projects. The major contributions of this research include: detecting the most common

root causes of test flakiness, identifying approaches that could manifest flaky behavior, classifying them into ten categories, and describing strategies that developers use to fix flaky tests. These categories were later extended by Eck *et al.* [17]. The authors proposed four additional categories, including *Platform Dependency*, which happen when random test failures occur only on specific platforms. Similarly, Hashemi *et al.* [22] conducted an empirical study of flaky tests in JavaScript, investigating the causes and common fixing strategies. Among the causes, the authors documented *Operating Systems* when a test fails due to a run in a specific OS or OS version while passing on others. A common strategy to deal with flaky tests is simply skip, ignore, or disable them. Developers may also exclude flaky and platform-specific code from coverage reports [23,24]. Our findings in RQ4 and RQ5 contribute to this research line. We find that developers may suspend execution in OS-specific tests for a given number of seconds to avoid flaky tests (RQ4). Moreover, we detect that developers may not test on certain OSs (with `@skipif` decorators) to avoid non-determinism (RQ5).

In summary, we find no study specifically focused on exploring OS-specific tests. However, it is important to notice that OS-specific tests are closely related to other studied areas in software testing, including test smells and flaky tests. Therefore, this research contributes to the software testing literature with a novel category of tests and shows its relation to other important research areas of the software testing landscape.

7 Conclusion

In this paper, we provided an empirical study to investigate how and why developers implement OS-specific tests in Python. We mined 100 popular Python systems and assessed their OS-specific tests both quantitatively and qualitatively. Our major findings can be summarized as follows:

- RQ1: OS-specific tests are present in 56% of the analyzed Python projects, and Windows is the most targeted OS.
- RQ2: APIs to identify operating systems in tests happen more frequently in test decorators (65%) than in test code (35%).
- RQ3: OS-specific tests target mostly file/directory (36.01%), third-party dependency (25.65%), and OS process (11.14%).
- RQ4: Multiple operations are performed in OS-specific, including set OS-specific value (62.68%), skip test (17.66%), and call OS-specific API (13.39%).
- RQ5: OS-specific tests are implemented mostly to overcome unavailable external resources (29.58%), unsupported standard libraries (18.08%), and flaky tests (13.85%).

Based on our findings, we discussed practical implications for practitioners and researchers, including the relation of OS-specific tests with test smells, technical debt, and flaky tests. We also discussed the efforts to test on Windows

properly and proposed a novel refactoring to improve some instances of OS-specific tests.

In future work, we plan to extend this research to other programming languages. It is important to note that OS-specific tests may occur in any other programming language that provides APIs to identify the operating systems, such as JavaScript, Java, and Go. We also plan to analyze version control data to explore how OS-specific tests are created and possibly removed over time in repositories. This would provide the basis to understand better how developers actually fix OS-specific tests, possibly supporting the development of novel techniques to correct them. Lastly, we plan to study "OS-specific" issues in the application code in addition to the test code. As a first effort in this direction, we explored the usage of platform-specific APIs in both test and application code [26]. The initial results suggest that developers may also use APIs to identify operating systems in the application code.

Acknowledgment

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

Conflict of Interest

The authors declared that they have no conflict of interest.

Data Availability Statements

Our dataset is publicly available: https://doi.org/10.5281/zenodo.10120045.

References

1. OS-Specific Tests Dataset (2024). URL https://doi.org/10.5281/zenodo.10120045
2. Aljedaani, W., Peruma, A., Aljohani, A., Alotaibi, M., Mkaouer, M.W., Ouni, A., Newman, C.D., Ghallab, A., Ludi, S.: Test Smell Detection Tools: A Systematic Mapping Study. In: International Conference on Evaluation and Assessment in Software Engineering, pp. 170–180. ACM (2021). DOI 10.1145/3463274.3463335
3. Aranega, V., Delplanque, J., Martinez, M., Black, A.P., Ducasse, S., Etien, A., Fuhrman, C., Polito, G.: Rotten green tests in Java, Pharo and Python. Empirical Software Engineering **26**(6), 130 (2021). DOI 10.1007/s10664-021-10016-2
4. Barbosa, L., Hora, A.: How and why developers migrate python tests. In: International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 538–548 (2022). DOI 10.1109/SANER53432.2022.00071
5. Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., Binkley, D.: An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In: International Conference on Software Maintenance (ICSM), pp. 56–65 (2012). DOI 10.1109/ICSM.2012.6405253
6. Beck, K.: Test-driven development: by example. Addison-Wesley Professional (2003)

7. Bodea, A.: Pytest-Smell: A smell detection tool for python unit tests. In: International Symposium on Software Testing and Analysis, pp. 793–796. ACM (2022). DOI 10.1145/3533767.3543290
8. Borges, H., Hora, A., Valente, M.T.: Understanding the factors that impact the popularity of GitHub repositories. In: International Conference on Software Maintenance and Evolution, pp. 334–344 (2016). DOI 10.1109/ICSME.2016.31
9. Borges, H., Valente, M.T.: What's in a GitHub star? understanding repository starring practices in a social coding platform. Journal of Systems and Software **146**, 112–129 (2018). DOI https://doi.org/10.1016/j.jss.2018.09.016
10. Breugelmans, M., Van Rompaey, B.: Testq: Exploring structural and maintenance characteristics of unit test suites. In: International Workshop on Advanced Software Development Tools and Techniques, p. 11 (2008)
11. Cruzes, D.S., Dyba, T.: Recommended steps for thematic synthesis in software engineering. In: International Symposium on Empirical Software Engineering and Measurement, pp. 275–284 (2011). DOI 10.1109/ESEM.2011.36
12. Cruzes, D.S., Dyba, T.: Recommended steps for thematic synthesis in software engineering. In: International Symposium on Empirical Software Engineering and Measurement, pp. 275–284 (2011). DOI 10.1109/ESEM.2011.36
13. Dabic, O., Aghajani, E., Bavota, G.: Sampling projects in github for MSR studies. In: International Conference on Mining Software Repositories (MSR), pp. 560–564. IEEE (2021). DOI 10.1109/MSR52588.2021.00074
14. Damasceno, H., Bezerra, C., Coutinho, E., Machado, I.: Analyzing Test Smells Refactoring from a Developers Perspective. In: Brazilian Symposium on Software Quality, pp. 1–10. ACM (2022). DOI 10.1145/3571473.3571487
15. Decan, A., Mens, T., Mazrae, P.R., Golzadeh, M.: On the use of github actions in software development repositories. In: International Conference on Software Maintenance and Evolution (ICSME), pp. 235–245 (2022). DOI 10.1109/ICSME55016.2022.00029
16. Delplanque, J., Ducasse, S., Polito, G., Black, A.P., Etien, A.: Rotten Green Tests. In: International Conference on Software Engineering (ICSE), pp. 500–511. IEEE (2019). DOI 10.1109/ICSE.2019.00062
17. Eck, M., Palomba, F., Castelluccio, M., Bacchelli, A.: Understanding flaky tests: The developer's perspective. In: European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 830–840. ACM (2019). DOI 10.1145/3338906.3338945
18. Feathers, M.: Working Effectively with Legacy Code. Prentice Hall Professional (2004)
19. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Professional (2018)
20. Greiler, M., Zaidman, A., Van Deursen, A., Storey, M.A.: Strategies for avoiding text fixture smells during software evolution. In: Working Conference on Mining Software Repositories (MSR), pp. 387–396. IEEE (2013). DOI 10.1109/MSR.2013.6624053
21. Gruber, M., Lukasczyk, S., Krois, F., Fraser, G.: An Empirical Study of Flaky Tests in Python. In: IEEE Conference on Software Testing, Verification and Validation (ICST), pp. 148–158. IEEE (2021). DOI 10.1109/ICST49551.2021.00026
22. Hashemi, N., Tahir, A., Rasheed, S.: An Empirical Study of Flaky Tests in JavaScript. In: International Conference on Software Maintenance and Evolution (ICSME), pp. 24–34. IEEE (2022). DOI 10.1109/ICSME55016.2022.00011
23. Hora, A.: What code is deliberately excluded from test coverage and why? In: International Conference on Mining Software Repositories, pp. 392–402 (2021)
24. Hora, A.: Excluding code from test coverage: practices, motivations, and impact. Empirical Software Engineering **28**(1), 1–33 (2023)
25. Hora, A.: Test Polarity: Detecting Positive and Negative Tests. In: International Conference on the Foundations of Software Engineering, pp. 537–541 (2024)
26. Job, R., Hora, A.: Availability and Usage of Platform-Specific APIs: A First Empirical Study. In: International Conference on Mining Software Repositories, pp. 27–31 (2024)
27. Luo, Q., Hariri, F., Eloussi, L., Marinov, D.: An empirical analysis of flaky tests. In: International Symposium on Foundations of Software Engineering, pp. 643–653. ACM (2014). DOI 10.1145/2635868.2635920

28. Maier, F., Felderer, M.: Detection of test smells with basic language analysis methods and its evaluation. In: International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 897–904. IEEE (2023). DOI 10.1109/SANER56733.2023.00108
29. Meszaros, G.: xUnit test patterns: Refactoring test code. Pearson Education (2007)
30. Palomba, F., Di Nucci, D., Panichella, A., Oliveto, R., De Lucia, A.: On the diffusion of test smells in automatically generated test code: An empirical study. In: International Workshop on Search-Based Software Testing, pp. 5–14. ACM (2016). DOI 10.1145/2897010.2897016
31. Pereira, G., Hora, A.: Assessing mock classes: An empirical study. In: International Conference on Software Maintenance and Evolution, pp. 453–463 (2020)
32. Peruma, A., Almalki, K., Newman, C.D., Mkaouer, M.W., Ouni, A., Palomba, F.: On the distribution of test smells in open source Android applications: an exploratory study. In: International Conference on Computer Science and Software Engineering, pp. 193–202. IBM Corp. (2019)
33. Peruma, A., Almalki, K., Newman, C.D., Mkaouer, M.W., Ouni, A., Palomba, F.: Ts-Detect: An open source test smells detection tool. In: European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1650–1654. ACM (2020). DOI 10.1145/3368089.3417921
34. Potdar, A., Shihab, E.: An exploratory study on self-admitted technical debt. In: International Conference on Software Maintenance and Evolution, pp. 91–100. IEEE (2014)
35. Pytest: https://docs.pytest.org (November, 2023)
36. Sierra, G., Shihab, E., Kamei, Y.: A survey of self-admitted technical debt. Journal of Systems and Software **152**, 70–82 (2019)
37. Spadini, D., Aniche, M., Bruntink, M., Bacchelli, A.: To mock or not to mock? an empirical study on mocking practices. In: International Conference on Mining Software Repositories, pp. 402–412 (2017)
38. Spadini, D., Aniche, M., Bruntink, M., Bacchelli, A.: Mock objects for testing java systems. Empirical Software Engineering **24**, 1461–1498 (2019)
39. Thorve, S., Sreshtha, C., Meng, N.: An Empirical Study of Flaky Tests in Android Apps. In: International Conference on Software Maintenance and Evolution (ICSME), pp. 534–538. IEEE (2018). DOI 10.1109/ICSME.2018.00062
40. Unittest: https://docs.python.org/3/library/unittest.html (November, 2023)
41. Van Deursen, A., Moonen, L., Van Den Bergh, A., Kok, G.: Refactoring test code. In: International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001), pp. 92–95 (2001)
42. Wang, T., Golubev, Y., Smirnov, O., Li, J., Bryksin, T., Ahmed, I.: Pynose: A test smell detector for python. In: International Conference on Automated Software Engineering (ASE), pp. 593–605. IEEE (2021). DOI 10.1109/ASE51524.2021.9678615
43. Winters, T., Wright, H., Manshreck, T.: Software engineering at google: Lessons learned from programming over time (2020)