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

Excluding Code from Test Coverage: Practices, Motivations, and Impact

Andre Hora

Received: date / Accepted: date

Abstract Test coverage measures the percentage of code that is covered (and uncovered) by tests. In practice, not all code is equally important for coverage analysis, like code that will not be executed during tests. Some coverage tools provide support for code exclusion from coverage reports, however, we are not yet aware of what code tends to be excluded, the reasons behind it, and the impact on coverage analysis. In this paper, we provide an empirical study to understand code exclusion *practices*, *motivations*, and *impact* on test coverage. We first mine popular Python projects that adopt test coverage to assess code exclusion *practices*. We find that (1) over 1/3 of the projects perform coverage exclusion, (2) 75% of the code are already created using the exclusion feature, and (3) developers exclude non-runnable, debug-only, and defensive code, but also platform-specific and conditional importing. Next, we explore the *motivations* behind the exclusions and the importance of test coverage in current days: (4) most code is excluded because it is already untested, low-level, or complex and (5) distinct test coverage recommendations are available for developers, such as covering the change, exploring coverage reports, and increasing coverage. Lastly, we assess the *impact* of code exclusion on test coverage. We detect that (6) code exclusion may impact test coverage, decreasing the number of statements that should be covered by tests and (7) test coverage can be refined by following code exclusion recommendations. Based on our findings, we discuss implications for both practitioners and researchers to improve coverage analysis, tools, and documentation as well as inspire novel research on test coverage.

Keywords Software Testing · Test Coverage · Software Evolution · Software Maintenance · Software Repository Mining

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

1 Introduction

Test coverage measures which parts of a program are actually executed during a test run, that is, the percentage of code that is covered (and uncovered) by tests [44]. Coverage measurement is used to assess the test effectiveness [16, 21, 27] and provides benefits to the developer workflow by offering an objective, industry-standard metric with actionable data [9, 27]. For instance, it can be used to identify untested areas of the code, to ensure that frequently changing code is covered, to facilitate code review, and to make sure that tests are not getting worse over time [9, 20]. Indeed, many coverage tools are available nowadays for most languages, for example, Coverage.py [16] for Python, JaCoCo [28] and Cobertura [8] for Java, Jest [23] and Istanbul [26] for JavaScript, to name a few.

In practice, not all code is equally important for coverage analysis. For example, code that will never be executed during tests is irrelevant for coverage analysis [9, 16]. Consequently, this type of code can actually harm coverage reports [9,16]. Some coverage tools provide native support to exclude code from coverage analysis, that is, the developer can deliberately flag the code to be ignored. Coverage.py[1] and Istanbul[2], for instance, provide features to filter out one or more lines from coverage reports. Despite being provided by mainstream coverage tools, we are not yet aware of *what code tends to be excluded* from test coverage reports nor *the reasons behind the exclusions*. The *impact of code exclusion* on coverage reports is also unknown. This knowledge can be used to understand code coverage exclusion, supporting the production of more accurate coverage reports. Moreover, this can also facilitate code review, for example, when coverage is integrated into the code review process [9].

In this paper, we provide an empirical study to better understand code exclusion *practices*, *motivations*, and *impact*. We focus on assessing what code is excluded from coverage reports, why, and what is the impact. For this purpose, we first mine 55 popular Python projects that adopt test coverage and assess their usage of code coverage exclusion. Specifically, we propose three research questions to assess code exclusion **practices**:

- *RQ1: How frequent is code excluded from test coverage?* Over one-third of the analyzed projects (20 out of 55) perform deliberate code coverage exclusion. In total, those projects use the exclusion feature in 534 cases.
- *RQ2: When is code excluded from test coverage?* Most code is excluded from coverage analysis since its creation (75%), meaning they are already created using the exclusion feature. In 25% of the cases, the exclusion feature is added over time (24 days later, on the median).
- *RQ3: What code is excluded from test coverage?* Most of the excluded code happens in conditional statements (42%) and exception handling (29%). Developers tend to exclude non-runnable, debug-only, and defensive code, but also platform-specific and conditional importing.

[1] https://coverage.readthedocs.io/en/coverage-5.3/excluding.html
[2] https://github.com/gotwarlost/istanbul#ignoring-code-for-coverage

Next, we explore the **motivations** behind the exclusions and the importance of test coverage in current days. For this purpose, we assess commit messages to detect exclusion rationales and test documentation to find test coverage recommendations:

- *RQ4: Why is code excluded from test coverage?* We find that most code is excluded because it is already untested (22%), low-level (20%), or complex (15%). Other rationales are related to deprecation/legacy code, parallelism, trivial/safe code, and non-determinism.
- *RQ5: What recommendations are provided by test coverage documentation?* Distinct test coverage recommendations are available for developers, such as covering the change, exploring coverage reports, and increasing coverage. Among the analyzed documentation, we find no recommendation for code exclusion on test coverage.

Finally, we assess the **impact** of code exclusion on test coverage. Specifically, we explore the impact of not having code exclusion on test coverage and the impact of using recommended code exclusion. In this case, we run the test suite of the selected projects in several configurations and assess the coverage results to answer the following research questions:

- *RQ6: What is the impact of not having code exclusion on test coverage?* Code exclusion may impact test coverage, decreasing the number of statements that should be covered by tests and the missing statements. For example, we find one test coverage that increased from 90% to 99% due to the usage of code exclusion.
- *RQ7: To what extent test coverage can be improved with recommended code exclusion?* Coverage can be refined by following code exclusion recommended by Coverage.py. We find that 8 out of 10 systems decreased the number of statements that should be covered by tests when their exclusions were properly configured (no matter if the project already had exclusions or not). Those statements refer to fair cases to be removed from coverage, like non-runnable and debugging-only code.

Based on our findings, we discuss several implications for both practitioners and researchers to improve coverage tools, testing guidelines, and coverage analysis as well as inspire novel research on test coverage. For practitioners, we discuss (1) the enhancement of coverage tools with mandatory explanations for the exclusion features; (2) the proposal of project guidelines to enforce explanations when using the exclusion feature; (3) the improvement of test coverage tools' documentation with novel exclusion examples; (4) the detection of trivial/safe candidates for coverage exclusion to produce more accurate test coverage reports; (5) the focus on test quality and careful on increasing test coverage at any cost; and (6) tips to properly increase test coverage. For researchers, we discuss (7) techniques to spot biased coverage reports; (8) to detect project-specific test coverage exclusion; and (9) novel studies on test coverage and automated test case generation.

Paper extension. This paper is an extension of our prior research [22]. This study extends the previous in the following three points. First, we extend the motivation study with a novel research question to understand the test coverage recommendations (RQ5). We provide a novel study to assess the impact of code exclusion on test coverage, specifically, the impact of no exclusion (RQ6) and the impact of recommended exclusion (RQ7). Thus, this study extends the prior one with three novel research questions.

Contributions. The contributions of this paper are fourfold: (i) we provide a study to assess code exclusion practices on coverage analysis; (ii) we present the common reasons for exclusions and test coverage recommendations for developers; (iii) we provide a study to assess the real impact of code exclusion on coverage analysis; and (iv) we propose a vast discussion and implications for practitioners and researchers.

Organization. Section 2 motivates the study. Section 3 presents the study design, while Sections 4-6 details the results. Section 7 discusses the implications. Lastly, Section 8 details the threats to validity, Section 9 discusses the related work, and Section 10 concludes the paper.

Data Availability The datasets generated during and/or analysed during the current study are available in the Zenodo repository, https://doi.org/10.5281/zenodo.5703716.

2 Test Coverage in a Nutshell

Software testing is a key activity in modern software development. Test coverage is largely adopted nowadays to support software testing. For example, 43 (86%) out of the top-50 most popular Python software projects hosted in GitHub use Coverage.py [16]. In addition to the large number of tools and benefits mentioned in the previous section (*e.g.,* identify untested code, ensure important code is covered, etc.), there is a tendency nowadays to present online coverage reports, integrating them in CI/CD workflows and facilitating code review. For example, industrial-scale tools are available to generate detailed coverage analysis for most programming languages, such as Codecov [10] and Coveralls [17]. This way, many open-source projects hosted on GitHub expose their coverage reports to the public via badges, *e.g.,* 

. The popular machine learning project *scikit-learn*[3] has an overall 98% coverage and its report is publicly available by Codecov.[4] In addition, to the coarse-grained view, coverage can also be assessed at fine-grained levels (*e.g.,* for files or commits) and tracked over time to ensure tests are getting better [21]. This way, at fine-grained levels, fine-configuring coverage analysis is even more important because few lines of code can have a large impact on the analysis.

Another benefit of coverage analysis is to support code review. For example, developers at Google state that coverage analysis can facilitate the code review

[3] https://github.com/scikit-learn/scikit-learn

[4] https://codecov.io/github/scikit-learn/scikit-learn

process: "*[...] embedding code coverage into your code review process makes code reviews faster and easier*" [9]. They present that during code review it is important to see not only coverage numbers but also each covered line highlighted to make sure that the most important code is covered [9]. During this process, ideally, the coverage analysis should be as clean as possible to avoid noisy data: "*Not all code is equally important, for example, testing debug log lines is often not as important*" [9].

Overall, test coverage is widespread in the software industry. Better understanding coverage exclusion practices can reveal novel use cases that should be adopted by developers as well as harmful cases that should be avoided. This can support, for example, the production of more accurate coverage reports and warn about the existence of biased ones. Morever, assessing and detecting code coverage exclusion practices can improve code review workflow by eliminating possible noisy code.

3 Study Design

In this section, we detail our study design. First, we present the selected coverage tool (Section 3.1) and the selected systems (Section 3.2). Next, we detail the three studies of this paper to assess the practices (Section 3.3), motivations (Section 3.4), and impact (Section 3.5) of code exclusion on test coverage.

3.1 Test Coverage Tool

In this study, we assess test coverage in the Python ecosystem. We select Python due to several reasons. *First*, Python is among the most important programming languages nowadays according to both GitHub[5] and TIOBE[6] rankings. *Second*, Python has a rich software ecosystem with worldwide adopted projects, like web frameworks, machine learning libraries, and data analysis libraries, to name a few. *Third*, the most popular coverage tool in Python, Coverage.py [16], is recommended by the official Python documentation,[7] making it the *de facto* coverage tool for Python and an "almost" native library; this does not happen in other popular programming languages like Java and JavaScript, in which several tools are available.

Coverage.py provides a feature to exclude one or more lines of code from coverage reports. There are two main solutions to use this feature: based on code comments or based on configuration files. Figure 1(a) shows an example in which code is excluded via a code comment (*i.e.,* `#pragma: no cover`). In this case, the `if debug` clause is excluded from reporting [16], that is, it is not counted as uncovered lines. Figure 1(b) presents an example in which a configuration file is used for coverage exclusion. In this case, the developers do

[5] GitHub ranking: https://bit.ly/2XHn2PY
[6] TIOBE ranking: https://www.tiobe.com/tiobe-index
[7] https://docs.python.org/3/library/trace.html

not need to flag the source code directly, but only indicate the patterns to be excluded.

```
function1()
if debug: # pragma: no cover
    msg = "log message"
    log_message(msg)
function2()
```

(a) Code comment

```
[report]
exclude_lines =
    pragma: no cover
    def __repr__
    if debug:
    raise NotImplementedError
    if 0:
    if __name__ == .__main__.:
```

(b) Configuration file

Fig. 1: Examples of coverage exclusion on coverage.py.

3.2 Case Studies

We aim to study relevant and real-world software systems. Thus, we first select the top-50 most popular Python software systems hosted on GitHub based on the stars metric [3], which is largely adopted in the software mining literature as a proxy of popularity. To add more relevant projects, we also select the top-20 most downloaded Python packages in the Python Package Index (PyPI) [36]; this ranking is obtained from the PyPI Stats.[8] After merging the two lists of systems (50+20), we have 68 unique and highly popular Python projects. We find that 80% of those projects (55 out of 68) rely on Coverage.py. This high rate confirms that coverage analysis is frequent in the Python ecosystem.

Finally, we verify how many projects adopt the coverage exclusion feature. We detect that 20 out of those 55 (36%) projects use coverage exclusion. This ratio of over one-third shows that the usage of the exclusion feature is common among the projects that rely on test coverage. The 20 projects are listed in Table 1: it includes popular projects as scikit-learn (43.2K stars), Home Assistant (37.6K stars), and CPython (34.8K stars), to name a few. It also contains the most downloaded projects in the Python ecosystem, for example, pip (2B downloads), DateUtil (1.8B downloads), and setuptools (1.6B downloads). Those numbers, thus, reinforce the relevance and impact of the selected projects.

3.3 Study 1: Code Exclusion Practices

To better understand the practices of code exclusion, we propose three research questions, as detailed in the following sections. We focus on code exclusion via

[8] https://pypistats.org/top

Table 1: Selected software systems.

scikit-learn/scikit-learn	home-assistant/core
python/cpython	apache/superset
tiangolo/fastapi	pypa/pipenv
encode/django-rest-framework	sqlmapproject/sqlmap
huge-success/sanic	ray-project/ray
willmcgugan/rich	plotly/dash
cookiecutter/cookiecutte	DateUtil/DateUtil
pypa/setuptools	pypa/pip
pypa/wheel	huggingface/transformers
binux/pyspider	locustio/locust

the pragma feature because this is the solution in which the developers can explicitly flag the code to be removed. Moreover, with pragma feature, the developer may fine-select the code to be removed from coverage analysis.

3.3.1 RQ1 (frequency)

We first assess the frequency of code exclusion (via the pragma feature) on test coverage. For this purpose, we analyze the last version of the 20 selected projects looking for references to the code exclusion feature provided by Coverage.py. We compute both the number of files and individual occurrences.

Rationale: We aim to understand to what extent the code exclusion feature is adopted in practice. Over-adoption may indicate, for instance, that developers are excluding a large portion of code from coverage reports or that developers are fine-configuring them (*e.g.,* systematically excluding all possible code). On the other hand, under-adoption may suggest, for example, that the feature is not broadly known by the community or are deliberately not adopted.

3.3.2 RQ2 (time)

In the second research question, we analyze when code is excluded from test coverage. We assess the exclusion occurrences in the version history of the 20 selected projects and compute when they were added in code. That is, for each line of code including the comment `#pragma: no cover`, we verify the commit and the date that added it (we rely on the PyDriller [39] mining tool to assess this data). This way, two cases can happen: (1) the comment has been created with the code and (2) the comment has been added later to the code. For example, in file `concurrency.py` of project fastapi, the comment was created with the code, as illustrated in Figure 2(a).[9] On the other hand, in file `color.py`, project Rich, the code was created in December 3[10] and the comment was added 4 days later, in December 7[11], as presented in

[9] Commit URL: https://bit.ly/3m5PDtf
[10] Commit URL: https://bit.ly/2V4orPI
[11] Commit URL: https://bit.ly/2V0IkYb

Figure 2(b). In this RQ, we compute the frequency of both cases; when the second case happens, we also measure the delay between the code addition and the comment addition, *i.e.,* 4 days in the previous example.

```
24 +     except ImportError:  # pragma: no cover
```

(a) Comment created with code (fastapi, commit: 3f9f4a0f).

```
14       - if TYPE_CHECKING:
      14 + if TYPE_CHECKING:  # pragma: no cover
```

(b) Comment added later (Rich, commit: 0ffdd2f2).

Fig. 2: Examples of code exclusion addition.

Rationale: We aim to discover whether developers are likely to add the exclusion feature when the code is created or later. The former may suggest that the practice is known beforehand and the code excluded since its conception, while the latter may indicate that developers may adapt the code over time.

3.3.3 RQ3 (excluded code)

Next, we assess what code is excluded from test coverage both quantitatively and qualitatively. For this purpose, we first analyze the code being flagged as excluded and classify its statement. For example, the code in Figure 2(a) refers to an *exception handling* statement, while Figure 2(b) presents a *conditional* statement. After detecting the most common statements, we look for further explanations in the documentation about the excluded code. For example, after inspecting the Python documentation, we detect that `TYPE_CHECKING`, in Figure 2(b), is a constant used by third-party static type checkers and this code is non-runnable.

Rationale: We aim to reveal and better understand both the excluded statements and their goals. While some scenarios are well-known to be excluded from test coverage (*e.g.,* non-runnable and debugging-only code) [16], we are not sure how frequent those cases are. Moreover, we are not aware of whether other cases exist. Revealing novel exclusion scenarios may support the improvement of test coverage documentation and aid developers when fine-configuring their reports. On the other hand, it may also reveal unexpected and possibly harmful cases that should be avoided.

3.4 Study 2: Code Exclusion Motivations

In this analysis, we focus on better understanding the reasons behind code exclusion (RQ4) and the importance of test coverage in current days (RQ5).

3.4.1 RQ4 (reasons)

Here, we analyze why code is excluded from test coverage. To assess the reasons, we inspect commit messages and code comments to detect rationales behind exclusions. We rely on the GitHub API to collect the RQ4 data; we expanded our dataset because we could not find enough relevant commit messages in the 20 projects. We then queried in the GitHub API for the occurrence of "*pragma: no cover*" in commit messages, and we manually inspected the first 250 results. After filtering out false-positives (*i.e.,* occurrences of pragmas not in the context code exclusion, like in code examples) and results from less popular projects (*i.e.,* projects with less than 10 stars [3]), we find 38 commit messages with rationales. We also find 3 cases in which the explanation was placed in the code itself, totaling 41 occurrences. We adopted thematic analysis to classify these messages, with the following steps [4, 18]: (1) initial reading of the messages, (2) generating a first code for each message, (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.

Rationale: We aim to uncover the reasons behind the usage of the coverage exclusion feature. So far, it is not clear why developers exclude code from coverage analysis. This information can inspire, for example, the improvement of coverage analysis. On the other hand, if the rationale is not convincing, this can reveal that developers are actually excluding code that could be tested and covered, which is not a best practice. The latter may shed light on practices that are actually biasing coverage reports.

3.4.2 RQ5 (recommendations)

To complement the prior analysis and get further insights regarding the importance of test coverage in current days, we now explore the coverage documentation. Specifically, we look for recommendations and guidelines for developers in the context of test coverage. To find those recommendations, we read the test and coverage documentation of the 20 selected projects. For example, the CPython project has a dedicated documentation section to improve test coverage [25]. Similarly, the scikit-learn project has documentation on how to test and improve test coverage [40]. As in the previous RQ, we adopted thematic analysis to classify the recommendations.

Rationale: Test coverage is highly encouraged in some software projects. For this purpose, projects may make available recommendations on how to contribute taking into account test coverage. We aim to reveal those recommendations and whether code exclusion practices are somehow documented. As a side effect, we are also interested in better understating whether this "pressure" to get high coverage may lead to the adoption of code exclusion.

3.5 Study 3: Code Exclusion Impact

Finally, we assess the impact of code exclusion on test coverage. Specifically, we explore the impact of not having code exclusion on test coverage (RQ6) and the impact of using recommended code exclusion (RQ7). To automate this analysis, we rely on the code exclusion via the configuration file feature.

For this purpose, we need to run the tests of the target systems and analyze the impact of code exclusion on coverage results. Running the full test suite is not always trivial and straightforward as the analyzed projects are large systems that may require complex testing environments. To avoid those complex cases that are harder to reproduce, we define the following inclusion criteria to select systems:

- The system has clear documentation on how to run the tests.
- The tests are self-contained and do not require large external dependencies.
- The tests pass with success.

This way, we filtered out projects that did follow those criteria. For example, the Ray documentation highlights its complexity: "*The full suite of tests is too large to run on a single machine*".[12] The Pip project requires several external dependencies to run successively: "*Running pip's entire test suite requires supported version control tools (subversion, bazaar, git, and mercurial) to be installed*".[13] The CPython documentation also states its complexity: "*There could be platform-specific code that simply will not execute for you, errors in the output, etc*".[14]

After filtering out those projects, we randomly select 5 projects originally with code exclusions and 5 projects originally without code exclusions (see Table 2). The projects with code exclusions come from the 20 out of 68 projects that adopt code exclusions; those projects have a total of 5,332 tests and are used to answer RQ6 and RQ7. The projects without code exclusions come from the other 48 projects (*i.e.,* 68–20) that do not adopt code exclusions; those projects have a total of 4,058 tests and are used to answer RQ7.

For each project, we run the tests on the latest available release, test covering the main source code folder (*i.e.,* not the test folder itself to avoid noise), and on Python 3.8.

3.5.1 RQ6 (impact of no exclusion)

In this research question, we assess the impact of not having code exclusion on test coverage. For each system, we compute test coverage in two scenarios: (i) with exclusions and (ii) without exclusions. *With exclusions* presents the original test coverage in which we run the tests preserving the code exclusions as they are. *Without exclusions* presents a modified test coverage in which we

[12] https://docs.ray.io/en/master/getting-involved.html#testing
[13] https://pip.pypa.io/en/latest/development/getting-started/#running-tests
[14] https://devguide.python.org/coverage

Table 2: Projects used in RQ6 and RQ7.

RQ	System	#Tests
Projects with exclusions (used in RQ6 & RQ7)	DateUtil	2,021
	Django REST	1,411
	FastAPI	1,038
	Rich	555
	Cookiecutter	307
	Total	5,332
Projects without exclusions (used in RQ7)	Thefuck	1,802
	Httpie	1,016
	Requests	564
	Flask	477
	Six	199
	Total	4,058

run the tests without the code exclusions. We then compare the number of statements, missing statements, and coverage ratio in the two scenarios.

Rationale: So far, it is not clear to what extent code exclusions may affect coverage reports. In the mining assessment (Study #1), we analyze the presence of code exclusion in source code, however, we do not run the tests to assess its real effect. We aim to fill this gap in this research question.

3.5.2 RQ7 (impact of recommended exclusion)

We assess the impact of using recommended code exclusion on test coverage. This way, we run a modified version of the test coverage with code exclusions recommended by the Coverage.py documentation [16], as presented in Figure 3.

```
[report]
exclude_lines =
    pragma: no cover
    def __repr__
    if self.debug:
    if settings.DEBUG
    raise AssertionError
    raise NotImplementedError
    if 0:
    if __name__ == .__main__.:
    class .*\bProtocol\):
    @(abc\.)?abstractmethod
```

Fig. 3: Example of code exclusions recommended by the Coverage.py documentation [16].

For each system, we compute test coverage in two scenarios: (i) with exclusions and (ii) with recommended exclusions. *With exclusions* presents the original test coverage, while *with recommended exclusions* presents the modified test coverage in which we run the test coverage with the recommended exclusions. As in the previous research question, we contrast the number of statements, missing statements, and coverage ratio in the two scenarios.

Rationale: We aim to assess whether test coverage could be improved if the recommended exclusions were adopted. While RQ6 focuses on the impact of not having code exclusion at all, here, we focus on the impact of having fair code exclusions (as recommended by the documentation).

4 Results: Code Exclusion Practices

In this study, we provide an overview of code exclusion from test coverage. We first assess how frequent code is excluded from coverage analysis (RQ1) and when (RQ2). Then, we analyze the source code to better understand what types of code tend to be excluded (RQ3).

4.1 RQ1: How frequent is code excluded from test coverage?

In this first research question, we assess the frequency of code exclusion on test coverage. Table 3 summarizes the frequency: overall, we find 534 exclusion occurrences (*i.e.,* the code comment `#pragma: no cover`) in 179 source files of the 20 analyzed systems. Pipenv is the project with the most occurrences (221 in 47 files), followed by pip (81 in 17 files) and FastAPI (58 in 18 files). The top-5 is completed with Rich (48) and CPython (40), while the remaining projects have together 86 occurrences.

Table 3: Frequency of code exclusion on test coverage per project.

Pos	System	#Files	#Exclusions	%
1	Pipenv	47	221	41
2	pip	17	81	15
3	FastAPI	18	58	11
4	Rich	33	48	9
5	CPython	16	40	7
6-20	Others	48	86	16
	All	179	534	100

It is worth to notice that the exclusion occurrences may happen in distinct parts of the projects, such as external and local code (Table 4). We find that 70% (377 out of 534) of the exclusion are located in external code.[15] Interest-

[15] We manually inspected all full file names and detected three patterns for external code: *lib*, *vendor*, and *thirdparty*. Thus, the 377 cases refer to file names including these patterns.

ingly, this suggests that exclusion is even broader in the Python ecosystem, *i.e.,* they are not restricted to analyzed projects, but also happen in their dependencies. On the other hand, we find that 30% (157 out of 534) of the exclusions happen in local code. From those 157 local cases, 95 (60%) are located in production code, while 62 (40%) in test code. Independently of the origin (external or local), these numbers suggest that exclusion is broad in the Python ecosystem.

Table 4: Frequency of code exclusion on test coverage per code location.

	Location	#Exclusions	%
All	External	377	70
	Local	157	30
Local only	Production	95	60
	Test	62	40

RQ1 Conclusion: Code is frequently excluded from test coverage analysis: we find 534 individual occurrences in 179 source files of the 20 selected projects.

4.2 RQ2: When is code excluded from test coverage?

Next, we analyze the version history of the 20 selected projects and assess when code is excluded from test coverage. We detect 934 exclusion occurrences over time.[16] In this RQ, we only analyze the occurrences that happen in local code because they are properly versioned and managed by the projects. We exclude the occurrences that happen in external code because their version history may be incomplete and flawed, for example, a third-party code may be simply copied and pasted, losing track of its version history.

This way, we find 309 cases that happen in local code, as summarized in Table 5. In the majority of the occurrences (75%), the exclusion comments were created with the code, that is, the code was already created including the exclusion feature. On the other hand, in 25% of the cases, the exclusions were added later to code, meaning the code was created without the exclusion feature but it was added later.

Considering the 79 occurrences in which developers added the exclusion comment later, the delay to update is 24 days, on the median. Table 6 breaks this analysis into three categories regarding the update time: fast (up to one month), medium (between 2 and 6 months), and slow (over 6 months). We

[16] Notice that this number is larger than the 534 cases of RQ1 because here we are assessing version history, while in RQ1 we only assessed the last version of the systems.

Table 5: When code is excluded from test coverage.

Exclusion was...	#	%
Created with code	230	75
Added later to code	79	25
Total (local only)	309	100

notice that most updates (57%) is fast, in a period up to 30 days (in 8 occurrences, the exclusion feature was added on the same day, just a few hours later). Next, we find that in 23% of the cases the update happens at a medium speed (2-6 months), while 20% are slow (*i.e.,* over 6 months).

Table 6: Delay to add the exclusion feature on test coverage.

Delay	#	%
Fast (0≤days≤30)	45	57
Medium (31≤days≤180)	18	23
Slow (days>181)	16	20
All	79	100

RQ2 Conclusion: Most code is excluded from coverage analysis since its creation (75%), meaning they are already created using the exclusion feature. In 25% of the cases, the exclusion feature is added over time, on the median, 24 days later.

4.3 RQ3: What code is excluded from test coverage?

In this RQ, we focus on a better understanding of what code is excluded from test coverage. Table 7 summarizes the 934 excluded code statements over time. Conditional statement is the top one (396, 42%), followed by exception handling (275, 29%). The top-5 is completed with method call (38, 4%), method definition (36, 3%), and pass statement (31, 3%).

Table 8 presents the most excluded code snippets as they are used in the analyzed systems. The most excluded code is the exception handling `except ImportError:` (131 cases). This is followed by three conditional statement: `else:` (102), `if __name__ == "__main__":` (52), and `if type_checking:` (33). Lastly, the list is completed with the `pass` statement (31). Next, we present more details about each category.

Conditional Statement. Control flow structures like `if` statements are less likely to be covered by tests [45] and in deliberately excluded code this is not different. Table 9 presents the most excluded if statements. The top one is

Table 7: Most excluded code statements.

Pos	Code Statement	#	%
1	Conditional Statement	396	42
2	Exception Handling	275	29
3	Method Call	38	4
4	Method Definition	36	4
5	Pass Statement	31	3
	Other	158	17
	All	934	100

Table 8: Most excluded code snippets.

Pos	Code	#	%
1	`except ImportError:`	131	14
2	`else:`	102	11
3	`if __name__ == "__main__"`	52	5
4	`if TYPE_CHECKING`	33	3
5	`pass`	31	3

`if __name__ == "__main__"`, which happens 52 times and typically represents non-runnable code.[17] The second one (`TYPE_CHECKING`) is a special constant that is assumed to be true by third-party static type checkers, while is false at runtime.[18] The third statement (`MYPY_CHECK_RUNNING`) also relates to static type analysis. Both `TYPE_CHECKING` and `MYPY_CHECK_RUNNING` are typically used in guarded imports.[19] Interestingly, the three aforementioned statements are related to *code that should not be executed at runtime*, which shows the concerns of developers to filter out those cases on test coverage analysis. Lastly, the two remaining if statements are related to filtering *specific platforms* from test coverage, which is detailed in the following lines.

Table 9: Most excluded if statement.

Pos	Code	#
1	`if __name__ == "__main__"`	52
2	`if TYPE_CHECKING`	33
3	`if MYPY_CHECK_RUNNING`	24
4	`if sys.platform.startswith('java')`	10
5	`if not ver_suffix`	7

[17] https://docs.python.org/3/library/__main__.html

[18] https://docs.python.org/3/library/typing.html#constant

[19] *e.g.*, https://bit.ly/36QmUDA

We now explore the most excluded platforms, OSs, and versions (Table 10). Here, we focus on the native APIs `sys`[20] and `os`,[21] which are the most frequently called in the analyzed if statements. Developers rely on the API `sys.platform` to filter out specific platforms (java, win32, and cli) and on the API `os.name` to filter out the operating system dependent modules nt (Windows) and Java. Interestingly, Windows and Java are the only platforms that developers are concerned about excluding from test coverage. The other two APIs, `os.path` and `sys.version_info`, relate to excluding according to specific OS paths and platform versions.

Table 10: Most excluded versions and platforms.

Pos	Code	#	Parameters
1	`if sys.platform`	22	java, win32, cli
2	`if os.path`	13	templates, static
3	`if os.name`	7	nt, java
4	`if sys.version_info`	6	major < 3

A commonly mentioned code snippet that should be excluded from test coverage is *debugging-only code* (Table 11). For example, the Coverage.py documentation [16] illustrates this as a possible scenario to aid developers. This way, we looked for debug code in our dataset, however, we could find only two code snippets: `if use_debug` in project home-assistant/core[22] and `if self.app.debug` in project huge-success/sanic.[23] Another related flag is the verbose one, which is typically used to produce detailed logging information, as in project scikit-learn.[24] Those detailed outputs may make the execution slower, thus, developers may not be concerned about testing (and covering) them.

Table 11: Most excluded debug and verbose code.

Pos	Code	#
1	`if use_debug`	1
2	`if self.app.debug`	1
1	`if verbose`	5
2	`if self.verbose`	1

Exception Handling. Exception handling is known to be hard to test [38]. We also find that developers tend to omit them from coverage analysis. Ta-

[20] https://docs.python.org/3/library/sys.html
[21] https://docs.python.org/3/library/os.html
[22] Commit URL: https://bit.ly/2JHpO4M
[23] Commit URL: https://bit.ly/33O5rd6
[24] Commit URL: https://bit.ly/3lXQ48a

ble 12 presents the most excluded exceptions on test coverage. The top one is `ImportError`, which happens in almost half of the cases, 48% (131 out of 275). This exception is raised when the *import statement fails to load a module*.[25] For example, in project FastAPI,[26] `ImportError` is captured if `asynccontextmanager` is not properly loaded. Next, we see the generic `Exception` (35), which is followed by `AttributeError` (18), `SQLAlchemyError` (14), and `UnicodeDecodeError` (13). Unlike `ImportError`, we could not derive any explanation for excluding those exception handling, and their occurrence seems to be project-specific.

Table 12: Most excluded exceptions.

Pos	Code	#	%
1	`except ImportError`	131	48
2	`except Exception`	35	13
3	`except AttributeError`	18	6
4	`except SQLAlchemyError`	14	5
5	`except UnicodeDecodeError`	13	4
	All	275	100

Method Call. We could not find any excluded meaningful method or function call in our analysis. Most of the 38 detected calls are local and refer to specific methods and functions. To better assess this problem, we analyze the reasons behind exclusions in RQ4.

Method Definition. As presented in Table 13, the most common method definition excluded from test coverage is `__repr__(self)`. This is a native function to compute the "official" string representation of an object and is typically used for debugging.[27] Coverage.py also suggests it to be excluded from test coverage [16]. Notice that the remaining method definitions are project-specific.

Table 13: Most excluded method definition.

Pos	Code	#
1	`def __repr__(self):`	6
2	`def _cygwin_patch(filename):`	3
3	`def test():`	2
4	`def dummy_get_response(request):`	2

Other Statements. Lastly, we present other infrequent statements that are excluded from test coverage (Table 14). First, we show `raise NotImplementedError`: abstract methods should raise this exception when they require derived

[25] https://docs.python.org/3/library/exceptions.html

[26] Commit URL: https://bit.ly/3oA0UTK

[27] https://docs.python.org/3/reference/datamodel.html#object.__repr__

classes to override the method or to indicate that the real implementation still needs to be added while the class is being developed.[28] For instance, it is used in project Dash in the abstract methods `start` and `stop`.[29] Next, we have `AssertionError`, which is raised when an assert statement fails. Both statements can be interpreted as *defensive code*. They are both recommended by the Coverage.py documentation as follows: "*Don't complain if tests don't hit defensive assertion code*" [16].

Table 14: Other excluded code.

Pos	Code	#
1	`raise NotImplementedError`	10
2	`raise AssertionError`	2

Table 15 summarizes the major cases in which developers rely on test coverage exclusion that we discussed in this RQ. We reinforce well-known cases, such as non-runnable, debugging-only, and defensive code. We reveal novel cases, such as platform-specific code and conditional importing.

Table 15: Summary of major cases in which developers exclude code from test coverage.

Category	Examples
Non-runnable code	`if __name__ == "__main__":`
Debugging-only code	`def __repr__(self):`
Defensive code	`raise NotImplementedError`
Platform-specific code	`if sys.platform`
Conditional importing	`except ImportError`
Unclear/project-specific	-

RQ3 Conclusion: Most of the excluded code from test coverage happens in conditional statements (42%) and exception handling statements (29%), which are code snippets known to be harder to test. In summary, developers tend to exclude non-runnable, debugging-only, defensive code, platform-specific, and conditional importing.

[28] https://docs.python.org/3/library/exceptions.html#NotImplementedError
[29] Commit URL: https://bit.ly/37KJ0XB

5 Results: Code Exclusion Motivations

In the previous RQs, we explored code exclusion practices on test coverage. To better understand the rationales behind the exclusions, we assess the explanations provided by the developers themselves (RQ4). Next, in RQ5, to deepen this analysis and get further insights regarding the importance of test coverage nowadays, we explore the coverage documentation and their recommendations for developers.

5.1 RQ4: Why is code excluded from test coverage?

The rationales are inferred from both commit messages and code comments and are summarized in Table 16. The most common explanation refers to untested code (22%), followed by low-level code (20%) and complex code (15%). Other rationales are related to deprecated/legacy code, parallelism, trivial/safe code, and non-determinism. Next, we elaborate on each rationale and present examples.

Table 16: Rationales to exclude code from test coverage.

Rationale	#	%
Untested Code	9	22
Low-level Code	8	20
Complexity	6	15
Deprecated/Legacy Code	5	12
Parallelism	3	7
Trivial/Safe Code	3	7
Non-determinism	2	5
Other	5	12
Total	41	100

Untested Code. The most frequent reason for test coverage exclusion is that the code is already not being tested. That is, developers discover untested code snippets and add the exclusion comment. In this scenario, they are likely interested in inflating the coverage numbers. For example, in project edx-organizations, the developer states: "*Bring coverage up to 100%: Just adds a couple '# pragma no-cover' comments to skip coverage on lines that already weren't covered. Having 99.99% coverage is more annoying than having 100%*".[30] Similarly, in project DateUtil, the developer comments: "*[...] uses nocover pragmas for known-uncovered parts of the tests, so that the baseline is 100%*".[31] Another developer says in project singularity: "*Add a 'no cover' pragma to a untested case*".[32]

[30] Commit URL: https://bit.ly/3mXXREl
[31] Commit URL: https://bit.ly/370DRv3
[32] Commit URL: https://bit.ly/36YjPS6

Low-level Code. Another common explanation is to exclude low-level code from test coverage. In this case, the code may be related to operations to handle compilation, processes, and specific platforms. For instance, in project thewalrus, the developer comments: "*Adds # pragma: no cover to loss_mat since functions is jitted*"[33] (JIT compiles the decorated function on-the-fly to produce efficient machine code). In project nutils, the usage is related to child processes: "*add no cover pragmas to child process code*".[34]

Complexity. Developers also tend to exclude complex code from test coverage. In nlp_profiler, the developer mentions he is skipping the `for` statements: "*added pragma: no cover to few lines in the core module to skip the for-loops blocks*".[35] Likewise, in thewalrus, a recursive function is excluded: "*Adds pragma: no cover to the recursive functions*".[36] In project isort, the developer is very direct: "*Improve test coverage [...] Setuptools commands would be hard to test*".[37] This is not a best practice: developers seem to be using the exclusion feature to avoid testing and yet increasing coverage.

Deprecated/Legacy Code. We also find some occurrences in which deprecated and legacy code are excluded. In project scikit-learn, the developer comments when adding the exclusion feature: "*Don't cover this deprecated method*".[38] Similarly, in project isort, a deprecated flag is excluded: "*Add pragma no cover to deprecated flags check*".[39] In project home-assistant/core, a legacy code is excluded: "*This part of the implementation does not conform to policy regarding 3rd-party libraries, and will not longer be updated*".[40]

Parallelism. In some cases, parallelism may appear as an issue for developers. In nlp_profiler, the developer states: "*setting the run_task() to pragma: no cover as due to some Parallelisation process code-coverage isn't able to capture metrics here*".[41] Likewise, in datacube-core, the developer comments when adding the exclusion feature: "*very rare multi-thread only event [...] Disable test cover*".[42] Like low-level, complex, and deprecated/legacy code, developers seem to be avoiding testing code that is difficult to test.

Trivial/Safe Code. This category is about code that has trivial or safe logic, such as stubs, debugging-only, logging, etc. For example, in project trio, the developer flags a stub function: "*Marked some function stubs with #pragma no cover*".[43] Similarly, in project borgmatic, the developer states that some trivial functions (with no code) should not be tested: "*Add some no-cover*

[33] Commit URL: https://bit.ly/33ZzM8K
[34] Commit URL: https://bit.ly/372NCJp
[35] Commit URL: https://bit.ly/3oGB0xG
[36] Commit URL: https://bit.ly/2W4m6EV
[37] Commit URL: https://bit.ly/2Kfl2vb
[38] Commit URL: https://bit.ly/2W42EIG
[39] Commit URL: https://bit.ly/373or9q
[40] Commit URL: https://bit.ly/2W45L32
[41] Commit URL: https://bit.ly/3naFKeC
[42] Commit URL: https://bit.ly/3oDVRS6
[43] Commit URL: https://bit.ly/3qKhaDs

pragmas on functions that don't need tests".[44] Notice that this category may be underestimated: developers may not write rationales in commit messages when excluding an obvious case from code coverage. This may explain the reason this category is infrequent in this analysis. For example, we started with 250 code exclusions, but in the end, we only manually classify 41 cases. We keep those 41 occurrences because their rationales are explicit, while we filter out the ones with unclear descriptions. The latter ones are candidates to be obvious cases.

Non-determinism. Developers may apply exclusion in non-deterministic or flaky code. In project coala, a code comment states in the excluded statement: "*those branches are only non-deterministically covered.*".[45] In project datalad, the developer mentions: "*Mark skips of flaky assertions as 'pragma: no cover'*".[46] This category also refers to code that is difficult to test.

Other. Finally, we find some infrequent rationales that are grouped together. Here, developers are concerned with excluding specific and challenging cases, for example, with memory issues, utility code, and unreachable code (*e.g.,* abstract methods). For example, in project orix, the developer flags a specific conditional statement that if executed may cause RAM crash: "*Adding a pragma no cover for high ram usage case*".[47]

> *RQ4 Conclusion*: Developers exclude code from test coverage mostly because it is already untested (22%), low-level (20%), or complex (15%). Other rationales are related to deprecated/legacy code, parallelism, trivial/safe code, and non-determinism. Most rationales are indeed related to code that is somehow hard test.

5.2 RQ5: What recommendations are provided by test coverage documentation?

Table 17 summarizes the test and coverage documentation landscape in the 20 selected systems. Most systems (17 out of 20) have documentation for developers on how to test the project. Regarding coverage, we find that 11 out of 20 systems have documentation about coverage analysis. Furthermore, 6 systems have coverage examples for developers on how to install or run coverage. Finally, we also detect that 10 projects make their coverage reports publicly available in Codecov [10] and one in Coveralls [17].

We now focus on the coverage documentation and their recommendations for test coverage, as summarized in Table 18. Overall, we find seven distinct recommendations for developers from a total of 17 occurrences. Notice that

[44] Commit URL: https://bit.ly/2W4BhOp
[45] Commit URL: https://bit.ly/3m1Wxil
[46] Commit URL: https://bit.ly/39YdWpZ
[47] Commit URL: https://bit.ly/2W286LX

Table 17: Summary of testing and coverage documentation.

Test documentation	Yes: 17 No: 3	
Coverage documentation	Yes: 11 No: 9	
Coverage example	Yes: 6 No: 14	
Public coverage tool	Codecov: 10 Coveralls: 1 None: 9	

there is no recommendation in the context of code exclusion. The most common recommendation is *cover the change* (35%). For example, the documentation of project DateUtil states: "*The most important thing to include in your pull request are tests - please write one or more tests to cover the behavior you intend your patch to improve*" [13]. Cookiecutter also highlights the importance of test coverage for code changes: "*Ensure that your feature or commit is fully covered by tests*" [12]. Similarly, the scikit-learn documentation states: "*We expect code coverage of new features to be at least around 90%*" [40].

Table 18: Recommendation on coverage documentation.

Rationale	#	%	
Cover the change	6	35	
Explore coverage reports	5	29	
Increase coverage	2	12	
Do not decrease coverage	1	6	
Prefer whitebox testing	1	6	
Improve branch coverage	1	6	
Run local coverage	1	6	
Total	17	100	

The second most common recommendation is *explore coverage reports* (29%). For example, the FastAPI documentation encourages developers to check coverage reports: "*There is a script that you can run locally to test all the code and generate coverage reports in HTML*" [14]. Cookiecutter also recalls to verify the coverage reports: "*You can also run coverage only report and get HTML report with statement by statement highlighting*" [12].

The third recommendation is *increase coverage* (12%). In this case, the documentation explicitly calls for increasing coverage numbers, that is, adding tests to cover untested areas. In CPython, it is stated: "*A good, easy way to become acquainted with Python's code and to help out is to help increase the test coverage for Python's stdlib. Ideally we would like to have 100% coverage, but any increase is a good one*" [25]. Later, the CPython documentation explains

how to select a code to increase coverage: "*Choosing what module you want to increase test coverage for can be done in a couple of ways [...] You can simply run the entire test suite yourself with coverage turned on and see what modules need help*" [25]. A similar recommendation is provided by scikit-learn, including a workflow to aid the developer: "*Workflow to improve test coverage: 1. Run 'make test-coverage'. The output lists for each file the line numbers that are not tested. 2. Find a low hanging fruit, looking at which lines are not tested, write or adapt a test specifically for these lines. 3. Loop*" [40].

Finally, we have four recommendations with a single occurrence. *Do not decrease coverage* is provided in project Apache Superset: "*Code coverage: Please ensure that code coverage does not decrease*" [24]. The other three recommendations come from CPython and focus on whitebox testing, branch coverage, and local coverage. (1) *Prefer whitebox testing*: "*When in doubt, stick with whitebox testing in order to properly exercise the code*" [25]. (2) *Improve branch coverage*: "*For the truly daring, you can use another powerful feature of coverage.py: branch coverage*" [25]. (3) *Run local coverage*: "*Do make sure, though, that for any module you do decide to work on that you run coverage for just that module. This will make sure you know how good the explicit coverage of the module is from its own [local] set of tests instead of from implicit testing by other code that happens to use the module*" [25].

RQ5 Conclusion: In addition to test documentation, software projects also include coverage documentation and examples. Distinct recommendations are provided for developers, including covering the change, exploring coverage reports, and increasing coverage. We find no recommendation to guide the usage of code exclusion on test coverage.

6 Results: Code Exclusion Impact

In this final study, we assess the real effect of code exclusion on coverage reports. For this purpose, we run the tests of the target systems and analyze the impact of code exclusion on test coverage.

6.1 RQ6: What is the impact of not having code exclusion on test coverage?

Table 19 details the impact of not having code exclusion on test reports. Overall, considering the five studied systems, there are 5,332 tests. The column "With Exclusions" presents the original test coverage in which we keep the exclusions as they are, while the column "Without Exclusions" presents the modified version in which we run the test coverage without the exclusions.

We notice that Rich is the most impacted system. The original test coverage (*i.e.,* with exclusions) reports 6,789 statements and 20 missing ones, leading to a coverage of 99%. However, without code exclusions, the test coverage outputs

7,639 statements and 760 missing ones. This represents +850 statements and +740 missing statements when compared to the original one, which decreases the coverage to 90% (*i.e.,* -9%).

Table 19: Impact of not having code exclusion on test coverage. Stms: statements. Miss: missing statements on test coverage. Cov: test coverage ratio. Table is sorted by the statements Δ.

System	With Exclusions (original)			Without Exclusions				
	Stms	Miss	Cov	Stms	Δ	Miss	Δ	Cov
Rich	6,789	20	99%	7,639	+850	760	+740	90%
Django REST	7,549	413	93%	7,580	+31	427	+14	93%
FastAPI	3,826	7	99%	3,836	+10	17	+10	99%
DateUtil	3,587	418	88%	3,591	+4	420	+2	88%
Cookiecutter	838	0	100%	840	+2	1	+1	99%

To better understand this change, we have manually inspected the code exclusion in Rich. We find that a total of 71 lines are annotated with code exclusion. The exclusion of `if __name__ == "__main__"` is the most frequent, with 40 cases. Next, we have the exclusion of the specific import statement `from typing_extensions import Literal` (4 cases) and the conditional `if WINDOWS` (3 cases). This explains why the number of statements and missing statements increase, while the coverage decreases to 90% when we run the test coverage without code exclusion. As an example, Figure 4 shows a file (emoji.py) in Rich with two code exclusions (lines 13 and 81).

```
12  else:
13      from typing_extensions import Literal  # pragma: no cover
14
15
16  if TYPE_CHECKING:
17      from .console import Console, ConsoleOptions, RenderResult
 …
77      ) -> "RenderResult":
78          yield Segment(self._char, console.get_style(self.style))
79
80
81  if __name__ == "__main__":  # pragma: no cover
```

Fig. 4: Rich code exclusion (file: emoji.py).

The other four systems also present more statements and missing ones when compared to the original test coverage: Django REST (+31/+14), FastAPI (+10/+10), DateUtil (+4/+2), and Cookiecutter (+2/+1). In Cookiecutter, the test coverage slightly decreased, from 100% to 99%. In this case, we find a single exclusion of the conditional `if __name__ == "__main__"`. Overall, as

the number of missing statements is low when compared to the number of statements, their coverage is not affected and remains mostly the same.

RQ6 Conclusion: Code exclusion may impact test coverage, decreasing the number of statements that should be covered by tests and the missing statements. From the five analyzed systems, we find one test coverage that increased from 90% to 99% due to the usage of code exclusion. We recall that the adoption of code exclusion is not necessarily harmful and may be related to fair use cases.

6.2 RQ7: To what extent test coverage can be improved with recommended code exclusion?

In this research question, we compute a modified version of the test coverage with code exclusions recommended by the Coverage.py documentation [16], like non-runnable and debugging-only code, as detailed in Figure 3.

Here, we assess two groups of projects: (1) projects originally with exclusion and (2) projects originally without exclusion.

Projects originally with exclusion. Table 20 details this analysis: the column "With Exclusions" presents the original test coverage, while the column "With Recommended Exclusions" presents the modified version in which we run the test coverage with the recommended exclusions.

We detect that 4 out of 5 projects decreased the number of statements that should be covered by tests, ranging from -52 (DateUtil) to -2 (Cookiecutter). DateUtil is the most impacted system: the original test coverage of DateUtil outputs 3,587 statements and 418 missing ones, leading to a coverage of 88%. With the recommended exclusions, the test coverage reports 3,535 statements and 403 missing ones. That is, there are -52 statements and -15 missing ones when compared to the original one, which increases the coverage from 88% to 89%. Likewise, Django REST also presents a reduction in the number of statements (-38) and missing statements (-10). This suggests that there is a possibility to improve the test coverage just by properly using code exclusions. We recall that those statements refer to fair cases to be removed from test coverage as recommended by the tool documentation. Therefore, those statements, like non-runnable and debugging-only code, are fine to be ignored in test coverage

Projects FastAPI and Cookiecutter presents a small reduction of the number statements, -8 and -2, respectively. Finally, we observe that Rich increases its number of statements (+27) and missing ones (+37). Notice, however, that this increase is smaller than the ones presented in RQ6 (see Table 19), suggesting that their original code exclusions are not so distinct from the recommended ones.

Projects originally without exclusion. Next, we analyze the impact of the recommended exclusions on projects originally without exclusions. Table 21

Table 20: Impact of the recommended exclusions on projects with exclusions. Table is sorted by the statements Δ.

System	With Exclusions (original)			With Recommended Exclusions				
	Stms	Miss	Cov	Stms	Δ	Miss	Δ	Cov
DateUtil	3,587	418	88%	3,535	-52	403	-15	89%
Django REST	7,549	413	93%	7,511	-38	403	-10	93%
FastAPI	3,826	7	99%	3,818	-8	7	0	99%
Cookiecutter	838	0	100%	836	-2	0	0	100%
Rich	6,789	20	99%	6,816	+27	57	+37	99%

details this analysis: the column "Without Exclusions" presents the original test coverage, while the column "With Recommended Exclusions" presents the modified version in which we run the test coverage with the recommended exclusions [16]. Considering the five studied systems, there are 4,058 tests.

We find that 4 out of 5 projects decreased the number of statements that should be covered by tests, ranging from -24 (Httpie) to -2 (Six). However, as the number of missing statements is low when compared to the number of statements, their coverage is not affected and remains mostly the same. For example, in project Httpie, there are -24 statements that should be covered by tests when compared to the original one, but the coverage remains 91%. The same pattern happens for the other four projects, in which coverage remains the same after using the recommended code exclusions. Therefore, if the project does not rely on code exclusions, it can at least run the default set of recommendations to refine coverage analysis.

Table 21: Impact of the recommended exclusions on projects without exclusions. Table is sorted by the statements Δ.

System	Without Exclusions (original)			With Recommended Exclusions				
	Stms	Miss	Cov	Stms	Δ	Miss	Δ	Cov
Httpie	4,166	358	91%	4,142	-24	346	-12	91%
Flask	2,794	238	91%	2,771	-23	230	-8	91%
Thefuck	3,635	457	87%	3,627	-8	455	-2	87%
Six	1,283	286	77%	1,281	-2	286	0	77%
Requests	2,163	799	63%	2,163	0	799	0	63%

RQ7 Conclusion: Test coverage can be refined by following code exclusion recommendations. Overall, 80% of the analyzed projects (8 out 10) decreased the number of statements that should be covered by tests when their exclusions are properly configured. Those statements refer to fair cases to be removed from test coverage as recommended by the documentation, like non-runnable and debugging-only code.

7 Discussion and Implications

Based on the findings provided by RQs1-7, we discuss several implications for both practitioners and researchers.

7.1 For Practitioners

Enhance coverage tools with mandatory explanations for the exclusion feature. We detect several rationales to exclude code from test coverage (RQ4). Most of them are related to code that is hard to test [19, 38, 45], for example, low-level code, complexity, legacy code, parallelism, and non-determinism. In those cases, it seems that developers are using coverage exclusion to avoid testing and yet increase coverage, which is not a best practice. Indeed, as presented in RQ5, testing with coverage in mind and also increasing test coverage is recommended by some projects, thus, this may lead developers to rely on code exclusion. This way, we shed light on this possible harmful practice (*i.e.,* code exclusion) for testing and coverage analysis. One solution to overcome this problem is to improve coverage tools by including mandatory explanations when using the exclusion feature. For example, instead of only flagging the code to be excluded from test coverage, developers would also need to add a comment explaining the exclusion. This way, the rationale would be explicit in the code, better-supporting code review and merge. Indeed, this can be seen as a heavyweight solution in which the tool enforces the explanation. A more lightweight solution is provided in the next implication in which the process enforces the explanation.

Propose project guidelines to enforce explanations when using the exclusion feature. Another solution to avoid the exclusion of code that is hard to test is to rely on project guidelines. We found one project applying this disciplined approach: Coala, a command-line interface [7]. The project testing guideline states: "*If some code is untestable, you need to mark your component code with # pragma: no cover. Important: Provide a reason why your code is untestable*",[48] as presented in Figure 5.

```
# Reason why this function is untestable.
def untestable_func(): # pragma: no cover
    # Untestable code.
    pass
```

Fig. 5: coala guideline for excluding code [7].

Several GitHub issues show how this discipline is taken rigorously in this project. For example, in the next issue, the developer alerts about the over-

[48] Testing guideline of Coala: https://bit.ly/3gAGCGR

usage of coverage exclusion: "*pragma: no cover is being over used to avoid writing test cases, and slipping through review. It should be prevented [...]*".[49] Interestingly, in RQ5, we found no recommendation or guideline for developers in the context of code exclusion. Therefore, we suggest that projects facing a similar dilemma should propose guidelines to enforce explanations when using the exclusion feature or improve their documentation about coverage exclusion. That is, developers would need to clearly explain the reasons they are excluding the code via commit messages or code comments. Also, test documentation can alert the developers about the benefits and drawbacks of using coverage exclusion, suggesting when it should be used and avoided.

Improve test coverage tools' documentation with novel exclusion examples. We find that developers apply test coverage in non-runnable, debugging-only, defensive, platform-specific, conditional importing, and project-specific code (RQ3). While some cases are already well-known and suggested by coverage tools (*e.g.,* debugging-only [16]), others are novel (*e.g.,* conditional importing) or even harmful for testing (*e.g.,* platform-specific). We contacted a core developer of Coverage.py and presented the code typically excluded from test coverage we have found in this study. He agreed that some cases could be added to the tool documentation as usage examples, while other cases should indeed be avoided. Thus, our findings can be used to improve test coverage tools' documentation, guiding developers when flagging their code. If further extended to other programming languages, this empirical study can promote the improvement of other tools as well.

Detect and flag trivial/safe candidates for coverage exclusion to produce more accurate test coverage reports. Despite the harmful cases aforementioned (*e.g.,* complexity, parallelism, etc.), there is a safety net of code in which developers can apply test coverage exclusion to refine their reports. In this context, we find a rationale related to trivial/safe code (RQ4), which may include for example non-runnable and debugging-only code. Those cases can be detected and flagged as excluded in the source code, as explored in RQ7. One solution to handle that is with the configuration file feature, as presented in Figure 1(b), which receives regular expressions and ensures that the matched pattern is excluded from test coverage. However, from the 20 studied projects, we find that 11 do not use this feature, that is, in these projects, the developers prefer to flag the code directly, as in shown Figure 1(a).

To overcome this issue, we propose the in-house detection and flagging of trivial/safe coverage exclusion candidates. We have performed a preliminary analysis, which is summarized in Table 22. Each line of the table presents a trivial/safe case detected in our study; column "#" presents the total occurrences of that case in the projects, while column "Has Exclusion" presents the occurrences that are actually excluded from test coverage. We note a large difference between both metrics, for example, only 10 out of the 1,528 `raise NotImplementedError` statements are flagged as excluded in this preliminary assessment. That is, the 1,518 (1,528 – 10) statements without coverage ex-

[49] Issue URL: https://bit.ly/3gAQCjr

clusion are *potential* candidates to be excluded. This suggests that coverage analysis can be more accurate if such a simple solution is adopted to detect and exclude trivial/safe code. Indeed, test coverage can be refined by following code exclusion recommendations, as suggested by the results of RQ7.

Table 22: Trivial/safe candidates for exclusion.

Code	#	Has Exclusion
`pass`	20,282	31
`if __name__ == "__main__":`	1,973	52
`raise NotImplementedError`	1,528	10
`except ImportError:`	1,299	131
`def __repr__(self):`	806	6
`raise AssertionError`	242	2
`if MYPY_CHECK_RUNNING:`	231	24
`if TYPE_CHECKING:`	116	33

Focus on test quality and be careful about increasing test coverage at any cost. So far, it is clear that some code snippets can be safely excluded from test coverage, while others tend to be more dangerous (RQ4). In any case, including or excluding code is likely to change the coverage report, as presented in our impact assessment (RQ6 and RQ7). In RQ5, we have seen several recommendations about test coverage in popular open-source projects, including covering the change, exploring coverage reports, and increasing coverage. In particular, the latter encourages developers to work on increasing test coverage only. Of course, this is not necessarily an issue, however, developers should also take into account test quality. In this context, it is important to recall that despite being largely adopted in the software industry [9,10,17,27], developers should not strive to achieve "magic" coverage numbers and this should not be a project requirement [9,20,30]. Indeed, one can have great coverage without checking correctness result [44], this way, a high code coverage percentage does not ensure high quality in the tests [9, 20]. Fowler suggests that coverage analysis should be used "*for identifying untested areas of the code, not for assessing the quality of a test suite*" [20].

Use the recommended code exclusions to improve test coverage. In RQ7, we have detected that 80% of the analyzed projects decreased the number of statements that should be covered by tests when their exclusions are properly configured. We recall that those exclusions refer to fair cases to be removed from test coverage as recommended by the documentation [16]. Therefore, we recommend that developers should be aware of those lists of fair exclusions to proper configure their coverage reports.

Tips to properly increase test coverage. With test quality in mind, one may benefit from properly increasing test coverage. In RQ5, we find valuable tips for developers that can be adopted by any software project, like covering the change, exploring coverage reports, not decreasing coverage, improving

branch coverage, and running local coverage. In particular, *run local coverage* recommendation of CPython suggests running coverage of a module with its local tests and not all tests of the suite of the project. This will show how good the test coverage of the module is from the point of view of the local tests instead of the external tests that may use the module [25].

7.2 For Researchers

Techniques to spot biased coverage reports. We have seen that developers tend to exclude code that is hard to test from coverage, such as complex and non-deterministic snippets (RQ3). This may produce misleading coverage reports with biased coverage numbers. In RQ6, we have seen examples of coverage change when using and not using code exclusion. For example, in an extreme case, coverage analysis may present high coverage numbers (*e.g.,* 90%), while in practice its coverage is low (*e.g.,* 60%). If this is true, coverage analysis loses one of its major benefits, which is identifying untested areas of the code [9, 20]. In this scenario, novel techniques can be proposed by the research community to detect biased coverage reports to warn developers about the misuse of code coverage exclusion.

Techniques to detect and enforce project-specific test coverage exclusion. In addition to the trivial/safe cases that can be excluded from coverage analysis, we also detect other more controversial cases that need further investigation. For example, we find that statements including debugging-only, verbose, legacy, deprecated, and dummy code are sometimes excluded from the analysis (RQ3). While one can say that debugging-only and verbose code is safer to be excluded, legacy, deprecated, and dummy code need careful investigation. As a preliminary assessment, we run the patterns presented in Table 23 to explore the potential exclusions in the studied projects. We find a large number of cases that could *potentially* be excluded from coverage analysis, depending on the project practices and policies. However, those are project-specific, for example, the debug pattern may appear as a plethora of cases, such as: `if debugging:`, `if app.debug:`, `if self._debug:`, etc. This way, novel techniques can be proposed by researchers to detect and enforce project-specific test coverage exclusion.

Table 23: Potential candidates for exclusion (depending on the project practices and policies).

Code	Total
`if .*verbose.*`	740
`if .*debug.*`	492
`if .*legacy.*`	140
`if .*deprec.*`	43
`if .*dummy.*`	21

Novel studies on test coverage and automated test case generation. We provide an empirical study to assess code exclusion practices on test coverage. We find that over 1/3 of the studied projects perform coverage exclusion (RQ1) and most code is excluded from reports since its creation (RQ2). In RQ3 and RQ4, we performed a qualitative analysis to explore the excluded code and its rationales. We focused on Python because it is a stable ecosystem regarding coverage analysis, in which a single coverage tool, Coverage.py [16], is the *de facto* one. This is not true for other programming languages in which several tools are available, for example, JaCoCo [28] and Cobertura [8] for Java, whereas Jest [23] and Istanbul [26] for JavaScript. Nevertheless, this opens room for novel research about coverage exclusion practices in other popular programming languages. This knowledge can be used to better understand other software ecosystems as well as can be compared with our findings in the Python ecosystem.

Moreover, our results are also relevant for researchers working in the area of automated test case generation [1,2,5,33,34]. Information about statements that do not need coverage may be exploited during the generation of test cases to avoid wasting time and focusing on more important areas of the code.

8 Threats to Validity

8.1 Internal Validity

Possible duplication when mining code history. During development, it may happen that a file is removed and added again over time. If this file contains code exclusions, they will be counted twice in RQ2 (time analysis). We recognize that this threat may happen in our dataset, however, it should be infrequent because two conditions must be satisfied: (1) a file is removed and added, and (2) this specific file include the code exclusion.

Local code analysis. RQ2 and RQ3 only assess the occurrences that happen in local code because they are properly versioned by the projects. The occurrences that happen in external code are excluded because their version history may be incomplete, which could bias history analysis. We recall that we manually inspected all full file names and detected three patterns for external code: *lib*, *vendor*, and *thirdparty*. Thus, the chance that we miss pattern names is low.

Manual classifications. In RQ4 (reasons) we started with 250 code exclusions, but in the end, we only manually classify 41 cases. We keep those 41 occurrences because their rationales are explicit, while we filter out the ones with poor descriptions. Moreover, the classification of the rationales was performed by one author. We rely on thematic analysis [18] to reduce the subjectiveness. Thematic analysis was also adopted to answer RQ5 and reduce the subjectiveness of the recommendations. It is important to recall that the context of the message, *e.g.,* developers, state of the project, and the source code, is omitted from the analysis. Further studies may take into account this context to refine the detected rationales.

Running the test suites and computing test coverage. In RQ6 and RQ7 (impact assessment) we run the test suites of the 10 selected projects and compute their test coverage. As noticed in our methodology, to be consistent among the systems, we run the tests on the latest available release, test covering the main source code folder (*i.e.,* not the test folder itself), and on Python 3.8.

8.2 External Validity

Assessing code exclusion via the pragma feature In RQs1-4, we assess code exclusion via the pragma feature because this is the solution in which the developers can explicitly flag the code to be removed and fine-select the removed code. We recall that developers may use the configuration file to exclude from coverage analysis, however, with the configuration file solution, we could not perform RQ2 (history analysis) and RQ4 (commit analysis), for example. Thus, to be uniform in our experiments, we decided to use a solution that worked for RQs1-4. Further research may explore the exclusion via the configuration file in more detail.

Code exclusion from test coverage in other programming languages. In this paper, we explored code exclusion from test coverage in the Python ecosystem. Indeed, code exclusion may be performed by any tool that makes this feature available. For example, the coverage tool Istanbul for JavaScript[50] also provides features to filter out one or more lines from coverage reports. Therefore, the investigated phenomenon may happen not only in Python, but in other ones like JavaScript.

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

9 Related Work

Test coverage is a topic largely explored in technical books (*e.g.,* [20,32,41,42]) and research papers (*e.g.,* [6,21,27,29,31,43,45]). Many coverage criteria have been proposed, such as statement, branch, and data-flow [37]. This metric presents several advantages, such as identifying untested code, ensuring that changing code is covered by tests, making sure that tests are not getting worse, and facilitating code review [9,20]. On the other hand, it also has some well-known limitations: a software project can have high coverage without checking correctness result [44], that is, a high code coverage rate does not ensure high

[50] https://github.com/gotwarlost/istanbul#ignoring-code-for-coverage

quality in the tests [9,20,30]. Fowler suggests that coverage analysis should be used *"for identifying untested areas of the code, not for assessing the quality of a test suite"* [20]. Moreover, like any other metric, if a magic number is predefined, developers may strive at any cost to achieve such a number (indeed, we saw that direction in most of the rationales presented in RQ4, in which developers avoid testing hard code to increase coverage numbers). Thus, those definitions should be avoided [9,20,30]. Another solution to assess test quality (and overcome coverage limitations) is mutation testing [9,11,35,44].

Coverage has long been the focus of various software testing research. Some studies assess code coverage evolution [21,31,43]. For example, recently, Hilton *et al.* [21] study the evolution of test coverage with the support of data provided by Coveralls [17]. The authors find that measuring the change to statement coverage does not capture the nuances of code evolution, thus, fine-grained analysis (*i.e.,* changed statements in commits) is needed to better capture coverage changes over time. Zhai *et al.* [45] assess the state of code coverage in five Python systems. They find that coverage depends on control flow structures and that exception handling statements are less frequently covered. Our study focus on code deliberately excluded from coverage, however, we concur with those findings in the sense that both conditional statements and exception handling tend to be less covered.

Chen *et al.* [6] propose an approach to estimate code coverage measures using execution logs. Kochhar *et al.* [29] analyze 100 open-source Java projects and detect that coverage has an insignificant correlation with the number of bugs that are detected after the release of the software at the project level. Ivanković *et al.* [27] investigate the usage of code coverage at Google. The authors analyze five years of historical data and 512 responses from developers. Overall, developers at Google are positive regarding code coverage and they view it as a valuable addition to their workflow. Google also presents a solution to generate test cases for uncovered code paths for increasing code coverage [15]. In this study, we contribute to the coverage research landscape by analyzing code coverage exclusion practices.

10 Conclusion

In this paper, we provided an empirical study to understand code that is excluded from coverage reports. We mined popular Python projects, assessed commit messages, analyzed coverage documentation, and computed test coverage by running test suites. We found that:

1. Over one-third of the analyzed projects performed code coverage exclusion.
2. Most code is excluded from coverage analysis since its creation, while in 1/4 the exclusion feature is added over time (24 days, on the median).
3. Developers tend to exclude non-runnable, debug-only, and defensive code, but also platform-specific and conditional importing.
4. Most code is excluded because it is already untested, low-level, or complex.

5. Distinct test coverage recommendations are available for developers, such as covering the change, exploring coverage reports, and increasing coverage.
6. Code exclusion may impact test coverage, decreasing the number of statements that should be covered by tests and and the missing statements
7. Test coverage can be refined by following code exclusion recommendations. Most of systems decreased the number of statements that should be covered by tests when their exclusions were properly configured, no matter if the project already had exclusions or not.

Based on our findings, we discussed several implications for both practitioners and researchers to improve coverage tools, testing guidelines, and coverage analysis and inspire novel research on test coverage. For example, we discussed the enhancement of coverage tools with mandatory explanations for the exclusion features; the proposal of project guidelines to enforce explanations when using the exclusion feature; the detection of trivial/safe candidates for coverage exclusion to produce more accurate test coverage reports; the focus on test quality and careful on increasing test coverage at any cost; and the proposal of techniques to spot biased coverage reports and project-specific coverage exclusions by the research community.

As future work, we plan to extend this research to other programming languages, such as Java and JavaScript. Specifically, we aim to investigate popular tools like JaCoCo [28] and Cobertura [8] for Java and Istanbul [26] for JavaScript. We aim to propose a technique to identify flawed coverage reports and bring to light potential harmful coverage analysis. We also plan to perform a survey with expert developers that exclude code from coverage to find novel rationales behind their exclusions. Finally, we observed that the removal of coverage exclusion is common in some projects. In this case, the developers seem to deal with coverage exclusion as a kind of code (or test) smell. This is an interesting assessment that we also plan to further investigate.

Acknowledgment

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

References

1. Anand, S., Burke, E.K., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., McMinn, P., Bertolino, A., et al.: An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software **86**(8), 1978–2001 (2013)
2. Arcuri, A.: Restful api automated test case generation with evomaster. ACM Transactions on Software Engineering and Methodology **28**(1), 1–37 (2019)
3. 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)
4. Brito, A., Valente, M.T., Xavier, L., Hora, A.: You broke my code: understanding the motivations for breaking changes in APIs. Empirical Software Engineering **25**(2), 1458–1492 (2020)

5. Brunetto, M., Denaro, G., Mariani, L., Pezzé, M.: On introducing automatic test case generation in practice: A success story and lessons learned. Journal of Systems and Software
6. Chen, B., Song, J., Xu, P., Hu, X., Jiang, Z.M.: An automated approach to estimating code coverage measures via execution logs. In: International Conference on Automated Software Engineering, pp. 305–316 (2018)
7. Coala: hhttps://github.com/coala/coala (November, 2021)
8. Cobertura: https://cobertura.github.io/cobertura (November, 2021)
9. Code Coverage Best Practices: https://testing.googleblog.com/2020/08/code-coverage-best-practices.html (November, 2021)
10. Codecov: https://codecov.io (November, 2021)
11. Coles, H., Laurent, T., Henard, C., Papadakis, M., Ventresque, A.: Pit: a practical mutation testing tool for java. In: International Symposium on Software Testing and Analysis, pp. 449–452 (2016)
12. Contributing to Cookiecutter: https://github.com/cookiecutter/cookiecutter/blob/master/CONTRIBUTING.md#Testing-with-tox (November, 2021)
13. Contributing to DateUtil: https://github.com/dateutil/dateutil/blob/master/CONTRIBUTING.md#testing (November, 2021)
14. Contributing to FastAPI: https://fastapi.tiangolo.com/contributing/#tests (November, 2021)
15. Cooper, S., Fulton, M.S.: Test case generation for uncovered code paths (2018). US Patent 9,990,272
16. Coverage.py: https://coverage.readthedocs.io (November, 2021)
17. Coveralls: https://coveralls.io (November, 2021)
18. 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)
19. Episkopos, D.C., Li, J.J., Yee, H.S., Weiss, D.M.: Prioritize code for testing to improve code coverage of complex software (2011). US Patent 7,886,272
20. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Professional (2018)
21. Hilton, M., Bell, J., Marinov, D.: A large-scale study of test coverage evolution. In: International Conference on Automated Software Engineering, pp. 53–63 (2018)
22. Hora, A.: What code is deliberately excluded from test coverage and why? In: International Conference on Mining Software Repositories, pp. 392–402 (2021)
23. https://jestjs.io: https://jestjs.io (November, 2021)
24. Increase Test Coverage (Apache Superset): https://github.com/apache/superset/blob/master/CONTRIBUTING.md#testing (November, 2021)
25. Increase Test Coverage (CPython): https://devguide.python.org/coverage (November, 2021)
26. Istanbul: https://istanbul.js.org (November, 2021)
27. Ivanković, M., Petrović, G., Just, R., Fraser, G.: Code coverage at Google. In: ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 955–963 (2019)
28. JaCoCo Java Code Coverage Library: https://www.eclemma.org/jacoco (November, 2021)
29. Kochhar, P.S., Lo, D., Lawall, J., Nagappan, N.: Code coverage and postrelease defects: A large-scale study on open source projects. IEEE Transactions on Reliability **66**(4), 1213–1228 (2017)
30. Marick, B., et al.: How to misuse code coverage. In: Interational Conference on Testing Computer Software, pp. 16–18 (1999)
31. Marinescu, P., Hosek, P., Cadar, C.: Covrig: A framework for the analysis of code, test, and coverage evolution in real software. In: International Symposium on Software Testing and Analysis, pp. 93–104 (2014)
32. Meszaros, G.: xUnit test patterns: Refactoring test code. Pearson Education (2007)
33. Palomba, F., Panichella, A., Zaidman, A., Oliveto, R., De Lucia, A.: Automatic test case generation: What if test code quality matters? In: International Symposium on Software Testing and Analysis, pp. 130–141 (2016)

34. Panichella, A., Kifetew, F.M., Tonella, P.: Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. IEEE Transactions on Software Engineering **44**(2), 122–158 (2017)
35. PIT Mutation Testing: https://pitest.org (November, 2021)
36. Python Package Index (PyPI): https://pypi.org (November, 2021)
37. Rapps, S., Weyuker, E.J.: Selecting software test data using data flow information. IEEE Transactions on Software Engineering (4), 367–375 (1985)
38. Sinha, S., Harrold, M.J.: Analysis and testing of programs with exception handling constructs. IEEE Transactions on Software Engineering **26**(9), 849–871 (2000)
39. Spadini, D., Aniche, M., Bacchelli, A.: Pydriller: Python framework for mining software repositories. In: Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 908–911 (2018)
40. Testing and improving test coverage (scikit-learn): https://scikit-learn.org/dev/developers/contributing.html#testing-and-improving-test-coverage (November, 2021)
41. Thomas, D., Hunt, A.: The Pragmatic Programmer: your journey to mastery. Addison-Wesley Professional (2019)
42. Winters, T., Wright, H., Manshreck, T.: Software Engineering at Google: Lessons Learned from Programming over Time (2020)
43. Zaidman, A., Van Rompaey, B., van Deursen, A., Demeyer, S.: Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. Empirical Software Engineering **16**(3), 325–364 (2011)
44. Zeller, A., Gopinath, R., Böhme, M., Fraser, G., Holler, C.: The fuzzing book. In: The Fuzzing Book. Saarland University (2019). URL https://www.fuzzingbook.org
45. Zhai, H., Casalnuovo, C., Devanbu, P.: Test Coverage in Python Programs. In: International Conference on Mining Software Repositories, pp. 116–120 (2019)