

What Code Is Deliberately Excluded from Test Coverage and Why?

Andre Hora

Department of Computer Science
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte, Brazil
andrehora@dcc.ufmg.br

Abstract—Test coverage is largely used to assess test effectiveness. In practice, not all code is equally important for coverage analysis, for instance, code that will not be executed during tests is irrelevant and can actually harm the analysis. Some coverage tools provide support for code exclusion from coverage reports, however, we are not yet aware of what code tends to be excluded nor the reasons behind it. This can support the creation of more accurate coverage reports and reveal novel and harmful usage cases. In this paper, we provide the first empirical study to understand code exclusion practices in test coverage. We mine 55 Python projects and assess commit messages and code comments to detect rationales for exclusions. We find that (1) over 1/3 of the projects perform deliberate coverage exclusion; (2) 75% of the code are already created using the exclusion feature, while 25% add it over time; (3) developers exclude non-runnable, debug-only, and defensive code, but also platform-specific and conditional importing; and (4) most code is excluded because it is already untested, low-level, or complex. Finally, we discuss implications to improve coverage analysis and shed light on the existence of biased coverage reports.

Index Terms—Software Testing, Test Coverage, Software Evolution, Software Maintenance, Software Repository Mining

I. INTRODUCTION

Test coverage measures the percentage of code that is covered (and uncovered) by tests, that is, which parts of a program are actually executed during a test run [1]. Coverage measurement is used to assess the test effectiveness [2]–[4] and provides benefits to the developer workflow by offering an objective, industry-standard metric with actionable data [3], [5]. 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 [5], [6]. Indeed, many coverage tools are available nowadays for most languages, for example, Coverage.py [4] for Python, JaCoCo [7] and Cobertura [8] for Java, Jest [9] and Istanbul [10] 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 [4], [5]. Consequently, this type of code can actually harm coverage reports [4], [5]. 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.py¹ and Istanbul², 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*. This knowledge can be used to understand code coverage exclusion practices, 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 [5].

In this paper, we provide the first empirical study to better understand code exclusion practices in coverage reports. We focus on assessing *what* code is excluded from coverage reports and *why*. For this purpose, we first mine 55 popular Python projects that adopt test coverage and assess their usage of code coverage exclusion. We then assess commit messages and code comments to detect rationales behind those exclusions. Specifically, we propose four research questions to assess the frequency, time, code, and reasons, as follows:

- **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.
- **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.

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 foment novel research on test coverage. In short, we discuss the enhancement of coverage tools with mandatory explanations

¹<https://coverage.readthedocs.io/en/coverage-5.3/excluding.html>


²<https://github.com/gotwarlost/istanbul#ignoring-code-for-coverage>

for the exclusion features; the proposal of project guidelines to enforce explanations when using the exclusion feature; the improvement of test coverage tools' documentation with novel exclusion examples; the detection of trivial/safe candidates for coverage exclusion to produce more accurate test coverage reports; and techniques to spot biased coverage reports as well as to detect project-specific test coverage exclusion.

Contributions. The contributions of this paper are threefold: (i) we provide the first empirical study to assess code exclusion practices in coverage analysis; (ii) we present what code is excluded and the reasons for exclusions; and (iii) we propose implications for practitioners and researchers.

Organization. Section II motivates the study. Section III presents the study design, while Section IV details the results. Section V discusses the implications. Lastly, Section VI details the threats to validity, Section VII discusses the related work, and Section VIII concludes the paper.

II. MOTIVATION

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 [4]. 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 on CI and facilitating code review workflow. For example, industrial-scale tools are available to generate detailed coverage analysis for most programming languages, such as Codecov [11] and Coveralls [12]. This way, many open-source projects hosted on GitHub expose their coverage reports to the public via badges, e.g., . For instance, 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 or worse [2]. 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.

Test coverage is widespread in the software industry. Thus, better understanding coverage exclusion practices can reveal novel usage cases that should be fomented 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.

Another benefit of coverage analysis is to support code review. For example, developers at Google state that coverage analysis can facilitate the code review process: “[...] *embedding code coverage into your code review process makes code*

reviews faster and easier” [5]. 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 [5]. 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*” [5].

Test coverage can be used to support code review. Therefore, assessing and detecting code coverage exclusion practices can improve code review workflow by eliminating possible noisy code.

III. STUDY DESIGN

A. Coverage Assessment

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 GitHub⁵ and TIOBE⁶ 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 [4], 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 the feature “pragma: no cover” 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 [4], 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 not need to flag the source code directly, but only indicate the patterns to be excluded.

| | |
|--|--|
| <pre>function1() if debug: # pragma: no cover msg = "log message" log_message(msg) function2()</pre> | <pre>[report] exclude_lines = pragma: no cover def __repr__ if debug: raise NotImplementedError if 0: if __name__ == '__main__':</pre> |
| (a) Code comment | (b) Configuration file |

Fig. 1: Examples of coverage exclusion in Coverage.py.

³<https://github.com/scikit-learn/scikit-learn>

⁴<https://codecov.io/github/scikit-learn/scikit-learn>

⁵GitHub ranking: <https://bit.ly/2XHn2PY>

⁶TIOBE ranking: <https://www.tiobe.com/tiobe-index>

⁷<https://docs.python.org/3/library/trace.html>

B. Case Study Selection

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 [13], 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) [14]; 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 the 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 I: 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. Our dataset is publicly available at <https://doi.org/10.5281/zenodo.4425671>.

TABLE I: Selected software systems.

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

C. Research Questions Assessment

1) *RQ1 (frequency)*: We first assess the frequency of code exclusion in 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.

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`

⁸<https://pypistats.org/top>

cover, we verify the commit and the date that added it (we rely on the PyDriller [15] mining tool and 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 3¹⁰ and the comment was added 4 days later, in December 7¹¹, as presented in 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: 0fdd2f2).

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) *RQ3 (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) [4], 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.

4) *RQ4 (reasons)*: Lastly, 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

⁹Commit URL: <https://bit.ly/3m5PDtf>

¹⁰Commit URL: <https://bit.ly/2V4orPI>

¹¹Commit URL: <https://bit.ly/2V0IkYb>

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 and results from toy and irrelevant projects, 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 [16], [17]: (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 foment, 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.

IV. RESULTS

A. RQ1: How frequent is code excluded from test coverage?

In this first research question, we assess the frequency of code exclusion in test coverage. Table II 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 II: Frequency of code exclusion in 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 III). We find that 70% (377 out of 534) of the exclusion are located in external code.¹² Interestingly, this suggests that exclusion is even broader in the Python

¹²We 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.

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 exclusion 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 that exclusion is broader in the Python ecosystem.

TABLE III: Frequency of code exclusion in 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.

B. 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.¹³ 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 IV. 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.

TABLE IV: 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 |

Considering the 79 occurrences in which developers added the exclusion comment later, the delay to update is 24 days, on the median. Table V 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 notice that most update (57%) is fast, in a period up to 30

¹³Notice 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.

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 V: Delay to add the exclusion feature in test coverage.

| Delay | # | % | |
|---|----|-----|--|
| Fast ($0 \leq \text{days} \leq 30$) | 45 | 57 | |
| Medium ($31 \leq \text{days} \leq 180$) | 18 | 23 | |
| Slow ($\text{days} > 181$) | 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.

C. 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 VI 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, 4%), pass statement (31, 3%), method definition (36, 3%), and pass statement (31, 3%).

TABLE VI: 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 VII 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 `pass` (31). Next, we present more details about each category.

TABLE VII: Most excluded code snippets.

| Pos | Code | # | % |
|-----|--|-----|-----|
| 1 | <code>except ImportError:</code> | 131 | 14 |
| 2 | <code>else:</code> | 102 | 11 |
| 3 | <code>if __name__ == "__main__"</code> | 52 | 5 |
| 4 | <code>if TYPE_CHECKING</code> | 33 | 3 |
| 5 | <code>pass</code> | 31 | 3 |
| | All | 934 | 100 |

Conditional Statement. Control flow structures like `if` statements are less likely to be covered by tests [18] and in deliberately excluded code this is not different. Table VIII presents the most excluded `if` statements. The top one is `if __name__ == "__main__"`, which happens 52 times and typically represents non-runnable code.¹⁴ Unsurprisingly, this is among the suggested code to be excluded by the Coverage.py documentation [4]. 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.¹⁵ 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.¹⁶ 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 in 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 VIII: Most excluded `if` statement.

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

We now explore the most excluded platforms, OSs, and versions. Here, we focus on the native APIs `sys`¹⁷ and `os`,¹⁸ 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 IX: Most excluded versions and platforms.

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

A commonly mentioned code snippet that should be excluded from test coverage is *debugging-only code*. For example, the Coverage.py documentation [4] illustrates this as a possible scenario to aid developers. This way, we looked

¹⁴https://docs.python.org/3/library/__main__.html

¹⁵<https://docs.python.org/3/library/typing.html#constant>

¹⁶*e.g.*, <https://bit.ly/36QmUDA>

¹⁷<https://docs.python.org/3/library/sys.html>

¹⁸<https://docs.python.org/3/library/os.html>

for debug code in our dataset, however, we could find only two code snippets: `if use_debug` in project `home-assistant/core`¹⁹ and `if self.app.debug` in project `hugoboss/sanic`.²⁰ Another related flag is the verbose one, which is typically used to produce detailed logging information, as in project `scikit-learn`.²¹ Those detailed outputs may make the execution slower, thus, developers may not be concerned about testing (and covering) them.

TABLE X: Most excluded debug and verbose code.

| Pos | Code | # |
|-----|--------------------------------|---|
| 1 | <code>if use_debug</code> | 1 |
| 2 | <code>if self.app.debug</code> | 1 |
| 1 | <code>if verbose</code> | 5 |
| 2 | <code>if self.verbose</code> | 1 |

Exception Handling. Exception handling is known to be hard to test [19]. We also find that developers tend to omit them from coverage analysis. Table XI presents the most excluded exceptions in 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*.²² For example, in project `FastAPI`,²³ `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 XI: Most excluded exceptions.

| Pos | Code | # | % |
|-----|--|-----|-----|
| 1 | <code>except ImportError</code> | 131 | 48 |
| 2 | <code>except Exception</code> | 35 | 13 |
| 3 | <code>except AttributeError</code> | 18 | 6 |
| 4 | <code>except SQLAlchemyError</code> | 14 | 5 |
| 5 | <code>except UnicodeDecodeError</code> | 13 | 4 |
| All | | 275 | 100 |

Method Call. We could not find any meaningful method or function call in our analysis. Indeed, most of the 38 detected calls are local and refer to specific methods and functions. Thus, like some of the exception handling aforementioned, this reinforces that developers may exclude code they are not willing to test (e.g., due to complexity or performance issues, etc.). This is clearly not a best practice, as developers seem to be excluding code snippets from test coverage without any plausible explanation. To better assess this problem, we analyze the reasons behind exclusions in RQ4.

¹⁹Commit URL: <https://bit.ly/2JHpO4M>

²⁰Commit URL: <https://bit.ly/33O5rd6>

²¹Commit URL: <https://bit.ly/31XQ48a>

²²<https://docs.python.org/3/library/exceptions.html>

²³Commit URL: <https://bit.ly/3oA0UTK>

Method Definition. As presented in Table XII, 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.²⁴ Coverage.py also suggests it to be excluded from test coverage [4]. Notice that the remaining method definitions are project-specific.

TABLE XII: Most excluded method definition.

| Pos | Code | # |
|-----|---|---|
| 1 | <code>def __repr__(self):</code> | 6 |
| 2 | <code>def _cygwin_patch(filename):</code> | 3 |
| 3 | <code>def test():</code> | 2 |
| 4 | <code>def dummy_get_response(request):</code> | 2 |

Other Statements. Lastly, we present other infrequent statements that are excluded from test coverage. As a first example, we show `raise NotImplementedError`: abstract methods should raise this exception when they require derived classes to override the method or to indicate that the real implementation still needs to be added while the class is being developed.²⁵ For instance, it is used in project `Dash` in the abstract methods `start` and `stop`.²⁶ 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” [4].

TABLE XIII: Other excluded code.

| Pos | Code | # |
|-----|--|----|
| 1 | <code>raise NotImplementedError</code> | 10 |
| 2 | <code>raise AssertionError</code> | 2 |

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

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

²⁴https://docs.python.org/3/reference/datamodel.html#object.__repr__

²⁵<https://docs.python.org/3/library/exceptions.html#NotImplementedError>

²⁶Commit URL: <https://bit.ly/37KJ0XB>

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.

D. RQ4: Why is code excluded from test coverage?

In the previous RQ, we explored what code is typically excluded from test analysis. To better understand the rationales behind the exclusions, we now assess the explanations provided by the developers themselves. The rationales are inferred from both commit messages and code comments and are summarized in Table XV. 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 XV: 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%”.²⁷ Similarly, in project dateutil, the developer comments: “[...] uses nocover pragmas for known-uncovered parts of the tests, so that the baseline is 100%”.²⁸ Another developer says in project singularity: “Add a ‘no cover’ pragma to a untested case”.²⁹

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”³⁰ (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”.³¹

Complexity. Developers also tend to exclude complex code from test coverage. In project 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”.³² Likewise, in project thewalrus, a recursive function is excluded: “Adds pragma: no cover to the recursive functions”.³³ In project isort, the developer is very direct: “Improve test coverage [...] Setuptools commands would be hard to test”.³⁴ Notice that, clearly, 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”.³⁵ Similarly, in project isort, a deprecated flag is excluded: “Add pragma no cover to deprecated flags check”.³⁶ 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”.³⁷

Parallelism. In some cases, parallelism may appear as an issue for developers. In project 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”.³⁸ Likewise, in project datacube-core, the developer comments when adding the exclusion feature: “very rare multi-thread only event [...] Disable test cover”.³⁹ 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”.⁴⁰ Similarly, in project borgmatic, the developer states that some trivial functions (with no code) should not be tested: “Add some no-cover pragmas on functions that don’t need tests”.⁴¹ 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.

³⁰Commit URL: <https://bit.ly/33ZzM8K>

³¹Commit URL: <https://bit.ly/372NCjp>

³²Commit URL: <https://bit.ly/3oGB0xG>

³³Commit URL: <https://bit.ly/2W4m6EV>

³⁴Commit URL: <https://bit.ly/2Kfl2vb>

³⁵Commit URL: <https://bit.ly/2W42EIG>

³⁶Commit URL: <https://bit.ly/373or9q>

³⁷Commit URL: <https://bit.ly/2W45L32>

³⁸Commit URL: <https://bit.ly/3naFKeC>

³⁹Commit URL: <https://bit.ly/3oDVR56>

⁴⁰Commit URL: <https://bit.ly/3qKhaDs>

⁴¹Commit URL: <https://bit.ly/2W4BhOp>

²⁷Commit URL: <https://bit.ly/3mXXREI>

²⁸Commit URL: <https://bit.ly/370DRv3>

²⁹Commit URL: <https://bit.ly/36YjPS6>

Non-determinism. Developers may apply coverage exclusion in non-deterministic and flaky code. In project *coala*, a code comment states in the excluded statement: “those branches are only non-deterministically covered.”⁴² Likewise, in project *datalad*, the developer mentions: “Mark skips of flaky assertions as ‘pragma: no cover’.”⁴³ 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”.⁴⁴

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.

V. DISCUSSION AND IMPLICATIONS

Based on the findings provided by RQs1-4, we discuss several implications for both practitioners and researchers.

A. 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 [18]–[20], 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 increasing coverage, which is not a best practice. This way, we shed light on this harmful practice 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.

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 [21]. 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”,⁴⁵ as presented in Figure 3.

⁴²Commit URL: <https://bit.ly/3m1Wxii>

⁴³Commit URL: <https://bit.ly/39YdWpZ>

⁴⁴Commit URL: <https://bit.ly/2W286LX>

⁴⁵Testing guideline of *coala*: <https://bit.ly/3gAGCGR>

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

Fig. 3: *coala* guideline for excluding code [21].

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-usage of coverage exclusion: “pragma: no cover is being over used to avoid writing test cases, and slipping through review. It should be prevented [...]”.⁴⁶ Thus, we suggest that projects facing a similar dilemma should propose guidelines to enforce explanations when using the exclusion feature. That is, developers would need to clearly explain the reasons they are excluding the code via commit messages or code comments.

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 [4]), 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, better 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. 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 XVI. Each line of the table presents a trivial/safe case detected in our study; column “#” presents the total

⁴⁶Issue URL: <https://bit.ly/3gAQCjr>

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 exclusion 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.

TABLE XVI: Trivial/safe candidates for exclusion.

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

It is important to recall that despite being largely adopted in the software industry [3], [5], [11], [12], developers should not strive to achieve “magic” coverage numbers and this should not be a project requirement [5], [6], [22]. Indeed, one can have great coverage without checking correctness result [1], this way, a high code coverage percentage does not ensure high quality in the tests [5], [6].

B. 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. This may produce misleading coverage reports with biased coverage numbers. 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 [5], [6]. 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. 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 XVII 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 XVII: Potential candidates for exclusion (depending on the project practices and policies).

| Code | Total |
|-----------------------------|-------|
| <code>if .*verbose.*</code> | 740 |
| <code>if .*debug.*</code> | 492 |
| <code>if .*legacy.*</code> | 140 |
| <code>if .*deprec.*</code> | 43 |
| <code>if .*dummy.*</code> | 21 |

Novel studies on test coverage and automated test case generation. This is the first empirical study to assess code exclusion practices in 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 their rationales. We focused on Python because it is a stable ecosystem regarding coverage analysis, in which a single coverage tool, Coverage.py [4], is the *de facto* one. This is not true for other programming languages in which several tools are available, for example, JaCoCo [7] and Cobertura [8] for Java, whereas Jest [9] and Istanbul [10] 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. Furthermore, our results are also relevant for researchers working in the area of automated test case generation [23]–[27]. 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 area of the code.

VI. THREATS TO VALIDITY

Mining code history. In RQ1 we analyze only the last version of the studied projects (*i.e.*, single version), while in RQ2 and RQ3 we analyze their code history (*i.e.*, multiple versions). This is the reason the number of code exclusions is distinct among the research questions. For example, in RQ1 we find a total of 534 exclusions in the last version of the studied systems, whereas in RQ2 and RQ3 we find 934 exclusions. The latter is larger because the search space is broader.

Local code analysis. RQ2 (time analysis) only assesses 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 classification of the rationales. In RQ4 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 [16] to reduce the subjectiveness.

Generalization. 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.

VII. RELATED WORK

Test coverage is a topic largely explored in technical books (e.g., [6], [28]–[30]) and research papers (e.g., [2], [3], [18], [31]–[34]). Many coverage criteria have been proposed, such as statement, branch, and data-flow [35]. 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 over, and facilitating code review [5], [6]. On the other hand, it also has some well-known limitations: a software project can have high coverage without checking correctness result [1], that is, a high code coverage rate does not ensure high quality in the tests [5], [6], [22]. Fowler suggests that coverage analysis should be used “*for identifying untested areas of the code, not for assessing the quality of a test suite*” [6]. Moreover, like any other metric, if a magic number is pre-defined, 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 [5], [6], [22]. Another solution to assess test quality (and overcome coverage limitations) is mutation testing [1], [5], [36], [37].

Coverage has long been the focus of various software testing research. Some studies assess code coverage evolution [2], [31], [32]. For example, recently, Hilton *et al.* [2] study the evolution of test coverage with the support of data provided by Coveralls [12]. 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.* [18] 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.* [34] propose an approach to estimate code coverage measures using execution logs. Kochhar *et al.* [33] 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.* [3] 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 [38]. In this study, we contribute to the coverage research landscape by analyzing code coverage exclusion practices.

VIII. CONCLUSION

In this paper, we provided an empirical study to understand code that is deliberately excluded from coverage reports. We mined 55 popular Python projects that adopt test coverage and assessed commit messages and code comments to detect rationales behind exclusions. We found that:

- Over one-third of the analyzed projects performed deliberate code coverage exclusion.
- 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).
- Developers tend to exclude non-runnable, debug-only, and defensive code, but also platform-specific and conditional importing.
- Most code is excluded because it is already untested, low-level, or complex.

Based on our findings, we discussed several implications for both practitioners and researchers to improve coverage tools, testing guidelines, and coverage analysis and foment 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; 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 [7] and Cobertura [8] for Java, Jest [9] and Istanbul [10] for JavaScript. We also plan to propose a technique to identify flawed coverage reports and bring to light potential harmful coverage analysis. Finally, we observed that the removal of coverage exclusion is common in some projects. In this case, the developers seems 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.

REFERENCES

- [1] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, “The fuzzing book,” in *The Fuzzing Book*. Saarland University, 2019. [Online]. Available: <https://www.fuzzingbook.org/>

- [2] M. Hilton, J. Bell, and D. Marinov, "A large-scale study of test coverage evolution," in *ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 53–63.
- [3] M. Ivanković, G. Petrović, R. Just, and G. Fraser, "Code coverage at google," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 955–963.
- [4] Coverage.py, <https://coverage.readthedocs.io>, November, 2020.
- [5] Code Coverage Best Practices, <https://testing.googleblog.com/2020/08/code-coverage-best-practices.html>, November, 2020.
- [6] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.
- [7] JaCoCo Java Code Coverage Library, <https://www.eclemma.org/jacoco>, November, 2020.
- [8] Cobertura, <https://cobertura.github.io/cobertura>, November, 2020.
- [9] <https://jestjs.io>, <https://jestjs.io>, November, 2020.
- [10] Istanbul, <https://istanbul.js.org>, November, 2020.
- [11] Codecov, <https://codecov.io>, November, 2020.
- [12] Coveralls, <https://coveralls.io>, November, 2020.
- [13] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of GitHub repositories," in *International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 334–344.
- [14] Python Package Index (PyPI), <https://pypi.org>, November, 2020.
- [15] D. Spadini, M. Aniche, and A. Bacchelli, "Pydriller: Python framework for mining software repositories," in *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 908–911.
- [16] D. S. Cruzes and T. Dyba, "Recommended steps for thematic synthesis in software engineering," in *International Symposium on Empirical Software Engineering and Measurement*, 2011, pp. 275–284.
- [17] A. Brito, M. T. Valente, L. Xavier, and A. Hora, "You broke my code: understanding the motivations for breaking changes in apis," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1458–1492, 2020.
- [18] H. Zhai, C. Casalnuovo, and P. Devanbu, "Test Coverage in Python Programs," in *International Conference on Mining Software Repositories (MSR)*, 2019, pp. 116–120.
- [19] S. Sinha and M. J. Harrold, "Analysis and testing of programs with exception handling constructs," *IEEE Transactions on Software Engineering*, vol. 26, no. 9, pp. 849–871, 2000.
- [20] D. C. Episkopos, J. J. Li, H. S. Yee, and D. M. Weiss, "Prioritize code for testing to improve code coverage of complex software," Feb. 8 2011, US Patent 7,886,272.
- [21] coala, <https://github.com/coala/coala>, November, 2020.
- [22] B. Marick *et al.*, "How to misuse code coverage," in *International Conference on Testing Computer Software*, 1999, pp. 16–18.
- [23] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.
- [24] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "Automatic test case generation: What if test code quality matters?" in *International Symposium on Software Testing and Analysis*, 2016, pp. 130–141.
- [25] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2017.
- [26] A. Arcuri, "Restful api automated test case generation with evomaster," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 1, pp. 1–37, 2019.
- [27] M. Brunetto, G. Denaro, L. Mariani, and M. Pezzé, "On introducing automatic test case generation in practice: A success story and lessons learned," *Journal of Systems and Software*.
- [28] G. Meszaros, *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.
- [29] D. Thomas and A. Hunt, *The Pragmatic Programmer: your journey to mastery*. Addison-Wesley Professional, 2019.
- [30] T. Winters, H. Wright, and T. Manshreck, "Software engineering at google: Lessons learned from programming over time," 2020.
- [31] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Software Engineering*, vol. 16, no. 3, pp. 325–364, 2011.
- [32] P. Marinescu, P. Hosek, and C. Cadar, "Covrig: A framework for the analysis of code, test, and coverage evolution in real software," in *International Symposium on Software Testing and Analysis*, 2014, pp. 93–104.
- [33] P. S. Kochhar, D. Lo, J. Lawall, and N. Nagappan, "Code coverage and postrelease defects: A large-scale study on open source projects," *IEEE Transactions on Reliability*, vol. 66, no. 4, pp. 1213–1228, 2017.
- [34] B. Chen, J. Song, P. Xu, X. Hu, and Z. M. Jiang, "An automated approach to estimating code coverage measures via execution logs," in *International Conference on Automated Software Engineering*, 2018, pp. 305–316.
- [35] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Transactions on Software Engineering*, no. 4, pp. 367–375, 1985.
- [36] PIT Mutation Testing, <https://pitest.org>, November, 2020.
- [37] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for java," in *International Symposium on Software Testing and Analysis*, 2016, pp. 449–452.
- [38] S. Cooper and M. S. Fulton, "Test case generation for uncovered code paths," Jun. 5 2018, US Patent 9,990,272.