Running a Red Light: An Investigation into Why Software Engineers (Occasionally) Ignore Coverage Checks

Alexander Sterk
ajhsterk@gmail.com
Delft University of Technology
The Netherlands

Mairieli Wessel
mairieli.wessel@ru.nl
Radboud University
The Netherlands

Eli Hooten
eli.hooten@sentry.io
Sentry.io
United States of America

Andy Zaidman
a.e.zaidman@tudelft.nl
Delft University of Technology
The Netherlands

ABSTRACT

Many modern code coverage tools track and report code coverage data generated from running tests during continuous integration. They report code coverage data through a variety of channels, including email, Slack, Mattermost, or through the web interface of social coding platforms such as GitHub. In fact, this ensemble of tools can be configured in such a way that the software engineer gets a *failing status check* when code coverage drops below a certain threshold. In this study, we broadly investigate the opinions and experience with code coverage tools through a survey among 279 software engineers whose projects use the Codecov coverage tool and bot. In particular, we are investigating why software engineers would ignore a failing status check caused by drop in code coverage. We observe that >80% of software engineers — at least sometimes — ignore these failing status checks, and we get insights into the main reasons why software engineers ignore these checks.

KEYWORDS

software testing, code coverage, coverage checks

ACM Reference Format:
Alexander Sterk, Mairieli Wessel, Eli Hooten, and Andy Zaidman. 2024. Running a Red Light: An Investigation into Why Software Engineers (Occasionally) Ignore Coverage Checks. In *5th ACM/IEEE International Conference on Automation of Software Test (AST 2024) (AST '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3644032.3644444

1 INTRODUCTION

Software has become a critical aspect of modern society as we are more and more relying on software for everyday tasks. Because of our reliance on software, its quality and reliability is indispensable [4, 24]. The repercussions of unreliable and incorrect software can be severe, ranging from users getting frustrated, over huge financial loses [31, 32], or years of lost research [33], to causing injury or even death [26]. As Aniche et al. state "Making sure software works is maybe the greatest responsibility of a software developer" [2].

Software testing is essential to ensure the quality of the software systems that we as a society rely on [5, 6]. However, writing tests is a tedious and time consuming task [1, 3, 5]. Code coverage, which refers to the percentage of code that is executed by a test suite, is both an important metric in assessing the reach of a test suite, but also an indicator of areas of code that may need further testing [47].

One of the more popular ways to measure code coverage in the development process is using code coverage tools or bots, which are integrated into popular open-source development platforms, such as GitHub, GitLab, or Bitbucket. These tools collect coverage data from running tests, calculate various coverage metrics, and report the results back to the developers [12, 44].

In this study, we focus on the Codecov[1] code coverage tool. Codecov integrates with many different open-source development platforms and supports a large number of programming languages. It generates coverage reports for each commit and in the case of a pull request, posts a comment to the pull request that provides a summary of the full report. It can provide a *"status check"* for pull requests, which can either be a pass or a failure, if the coverage results of the pull request do not pass the standards of the project. It can also report the coverage results through email or Slack. An example of a comment and the status checks is given in Figure 1. Important for our particular study, is that Codecov also calculates coverage levels for individual pull requests, which Codecov call *patch coverage*. This only measures coverage for the lines that were actually changed in the pull request[2].

Due to Codecov being free for students and public projects, and because it is used by over a million software developers, they have a large amount of available coverage data, which they made available to us, to use for research purposes. Specifically, that data made us wonder why software engineers would deliberately ignore indicators by the coverage tool. As such, our guiding research question is the following:

> **RQ:** Why are coverage checks ignored by software engineers?

In this paper, we explore the developers' rationale when considering code coverage and working with a code coverage tool, specifically the Codecov tool. We do so through a survey among

AST '24, April 15–16, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0588-5/24/04.
https://doi.org/10.1145/3644032.3644444

[1] https://about.codecov.io/
[2] See https://docs.codecov.com/docs, last visited October 4th, 2023.

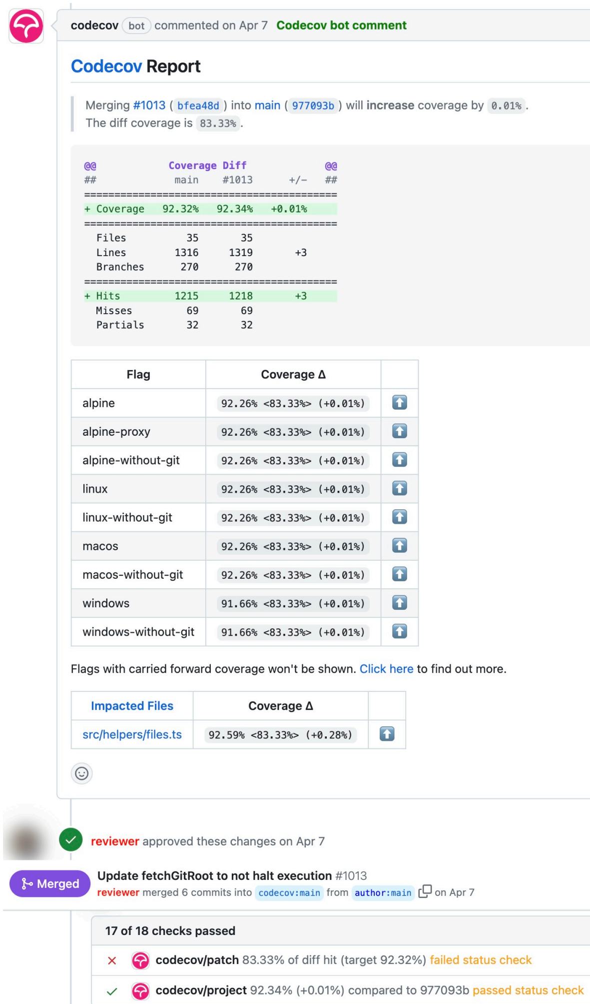

Figure 1: Example of a Codecov comment and status checks

279 software engineers of whom the projects make use of Codecov. Our findings indicate that >80% of developers sometimes purposefully ignore a coverage check, for a variety of reasons.

2 STUDY SETUP

The goal of our study is to explore how developers use and look towards code coverage in open-source development projects. Specifically, we aim to come up with reasons why coverage checks would get ignored, as well as categorize what developers would consider good coverage practices. To address a sizeable population, we have chosen to conduct a survey to gauge the aforementioned elements. When preparing and executing the survey, we have followed the guidelines set by the Delft University of Technology's Research Ethics Council and received approval from that council to conduct our study. We used a GDPR compliant version of Qualtrics for our survey.

2.1 The Survey

Mindful of the different roles that exist within open-source software development [16, 17], we have opted to use two different paths in the survey based on whether a respondent identifies themselves as a *maintainer*, a software engineer that manages, reviews, and integrates contributions (pull requests), or a *contributor*, a software engineer that makes contributions and opens a pull request. Both groups mostly received the same questions, although some were written from the perspective that better aligns with the chosen group.

During a pilot run with 4 PhD students in the research group, we have received feedback on the survey, which can be summarised as follows: (1) the survey was considered lengthy, (2) some ordinal scale options were questionable or difficult to choose between, (3) a lack of images and clarity in the survey when describing different functionalites of code coverage tools. This feedback was incorporated in the final survey design. A summary of the final survey is shown in Table 1.

2.2 Recruitment of Participants

The target audience for our survey were developers familiar with using code coverage tools on open-source development platforms. We have been fortunate enough to be able to rely on Codecov for obtaining users which had contributed to GitHub projects that use Codecov. The information we collected was the number of contributions, the time of the last contribution, usernames and repository names. The entire dataset consisted of around 260,000 entries (including duplicates).

We then used GitHub's API to look up these usernames and find user profiles with public email addresses. This was very important since we did not want to breach any terms of service or be a nuisance to people. Subsequently, we were left with roughly 90,000 email addresses. From these we filtered users based on the number of commits they have made overall, and when their last commit took place. Only users with over 100 commits, and a last commit between the start of 2019 and the end of 2021 were kept. This was done to (1) prevent absolute newcomers from taking the survey, and (2) assure that participants had recent experience with the tool. This filtering resulted in a list of 11,000 people, of which we randomly selected 2,000 to send our survey to, by email. We opted for 2,000, to account for a potentially low response rate of 6%, which was a rough estimate based on previous studies in software engineering [37, 43].

2.3 Deployment of the Survey

In December 2021, our survey was sent out to 2,000 email addresses. During this time, the survey was open and able to receive submissions for roughly a month. However, most of the responses were submitted in the first two weeks. We obtained 379 response attempts in total, and 278 complete responses of respondents who finished the entire survey, i.e., a response rate of ~14%.

3 RESULTS

This section describes the results from the survey. We only report on the 278 surveys that were completed. Our replication package contains all anonymized data [39].

#	Question	Type	A/M/C
Demographic questions			
1	For how many years have you been developing software? You can consider all hobby, study and/or work experience.	Open	A
2	For how many years have you been active on open-source development platforms, such as GitHub, Gitlab, etc?	Open	A
3	How often do you contribute to an open source project?	Closed	A
3a	On average, I make a contribution (e.g. a commit, a pull request, etc) to somebody else's project(s) ...		
3b	On average, I make a contribution to my own project(s) ...		
3c	On average, I review other people's contributions ...		
4	When contributing to open source projects, I primarily act as a: (Code contributor OR Maintainer/Project Manager)	Closed	A
5	Do you work in software development in a professional capacity? (Yes OR No)	Closed	A
6	What is your job title?	Open	Q5="Yes"
7	How long have you been performing the following tasks, in either a professional or hobby capacity?	Closed	All
7a	Automatic software testing tasks, such as writing unit tests or integration tests		
7b	Manual software testing tasks or performing any sort of manual quality assurance functions		
7c	Performing code review of others' contributions to any project, either open or closed source		
8	When it comes to automated software testing (e.g. unit testing, integration testing, etc) and its relationship to overall code quality, do you believe that automated software testing is: (Likert scale of importance)	Closed	A
Main questions			
9	How often do you use code coverage tools outside of GitHub? For example, on your own machine.	Closed	A
10	How often do you utilise the information from code coverage tools on GitHub?	Closed	A
10a	I use code coverage tools while developing/contributing ...		
10b	I use code coverage tools while reviewing ...		
11	In the last question you answered you never utilise the information from code coverage tools on GitHub. Do you have any particular reason why you do not use code coverage tools on GitHub? *After this question, the survey ends.*	Open	Q10="Never"
12	In your experience, what is a good coverage goal for a project? For example, is there a certain set of rules you'd like to follow, or a certain target you'd like to reach? If it's possible, please also give us your reasoning.	Open	A
13	Please give your opinions on the following statements:	Closed	
13a	Code coverage is a good metric to consider as part of overall code quality		A
13b	Code coverage tools on open- source platforms provide an incentive to improve coverage and/or write tests.		A
13c	I am more likely to approve a pull request that improves code coverage than ones that lower it.		M
13d	If my pull request improves coverage, it is accepted more quickly, in my experience.		C
14	How often do you write tests for projects you are contributing to?	Closed	A
15	How often do you write a test or multiple tests with (just) the intent to improve the code coverage?	Closed	A
16	How often are you asked/encouraged to better test your contributions, in the comments of a pull request you opened?	Closed	C
16a	People ask me this ...		
16b	A coverage tool asks me this ...		
17	Do you remember a particularly interesting instance where this (see Q16) happened? How did the situation get resolved?	Open	C
18	How often do you have to tell a contributor that their tests need to be improved, based on the results of a coverage tool?	Closed	M
19	Do you remember a particularly interesting instance where this (see Q18) happened? How did that situation get resolved?	Open	M
20	The following actions constitute incentives to improve coverage and/or write tests, and can also be done by code coverage tools. Please rank them on how much incentive you think they provide, from most incentive to least incentive.	Ranking	A
20a	Leaving a comment on a pull request, summarising the coverage changes		
20b	Giving a failing status check for a commit or pull request, preventing automatic merging		
20c	Annotating uncovered lines in the "Files changed" overview of a pull request		
20d	Notifying users through messaging applications or email, if coverage is lowered		
20e	Reminding users of contributing guidelines, when opening a pull request		
21	In your experience, what is the best way to provide incentive for improving code coverage?	Open	A
22	Can you come up with situations where you would ignore a failing coverage check? What are your reasons?	Open	A
23	How often do you neglect or ignore a failing coverage check on a commit or pull request?	Closed	A
23a	I ignore a failing coverage check when contributing to a project ...		
23b	I ignore a failing coverage check when reviewing a pull request ...		
24	Can you give us 2 things you like about using a code coverage tool on an open-source platform?	Open	A
25	Can you give us 2 things you dislike about using a code coverage tool on an open-source platform?	Open	A

Table 1: Summary of the survey questions. The final column indicates whether this question was dependent on a previous question, or asked to All respondents, or only Maintainers, or Contributors.

3.1 Demographic

General demographics. The demographics of our survey population can be seen in Figure 2. We observe that a majority of the participants have several years of experience with software development. Furthermore, while most participants actively work in a software development field, there are a substantial number of participants that do not. These participants come from a number of different fields, such as biology, mathematics, ecology, mobile development, etc. Some participants work in research, while others work in a more practical setting. Furthermore, some participants work in a high position, such as CEO, while others have listed themselves as interns. We clustered the different occupations into larger categories, which are shown in Table 2.

Overall, the participants to our study seem quite diverse, and coupled with the fact that our participants are from a set of diverse organisations, this should ensure the generalizability of our insights.

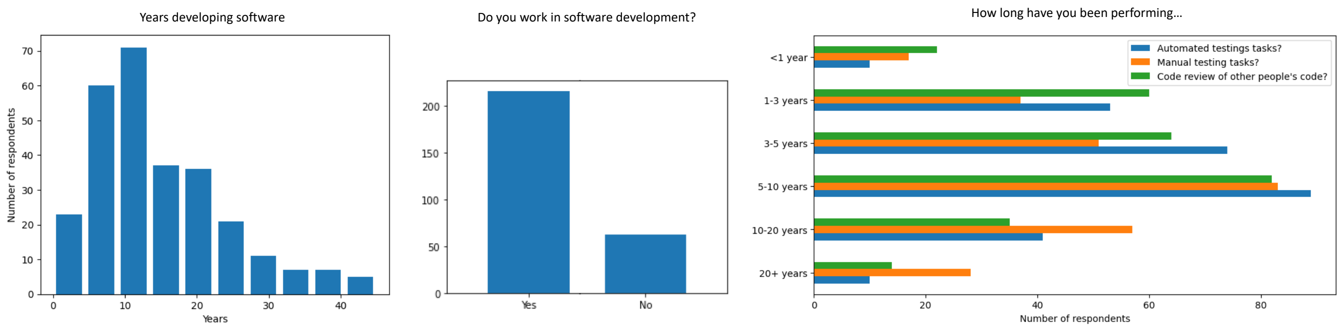

Figure 2: General experience with software development

Field	Count
Software Development/Engineering	109
Management/Leadership	41
Research/Science/PhD	38
IT/Infrastructure/DevOps	15
Data Science/Analysis	11
Software architect	10
Other	7

Table 2: Distribution of participants' occupations

Open-source experience. Figure 3 shows that most participants have experience with open-source development platforms. We also observe the frequency at which they perform certain tasks related to open-source software development on these platforms. However, we also noticed some inconsistencies, with some participants claiming to have over 20 years of experience, while platforms such as GitHub have not been around as long. Nonetheless, it is entirely possible that participants have experience with older version control systems. Furthermore, we find that some participants rarely or never perform one or more of these tasks.

When gauging whether a participant to the survey considers themselves more of a code contributor, or more of a project maintainer, we found the groups to be of almost equal size (143 maintainers vs. 135 contributors). Depending on the role that they assigned to themselves (maintainers vs. contributor), some of the follow-up questions in the survey will differ.

3.2 Quantitative Results

Since the quantitative questions are all closed or Likert scale questions, their results can all be interpreted using graphs.

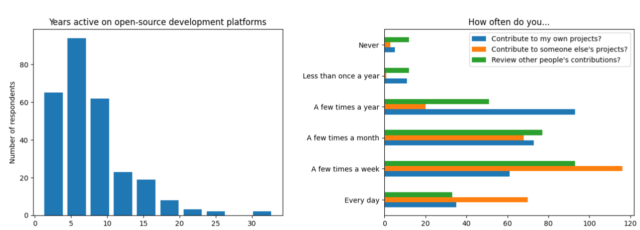

Figure 3: Experience with and frequency of open-source development

Code coverage tool usage. The first set of questions is regarding how often the participants use code coverage tools. From Figure 4 we find that using code coverage tools for each pull request is the most popular answer, regardless of whether one is contributing to a pull request, or reviewing one. In the case that a participant responded *"Never"* to both questions, this would mean that they lack the experience that is required for the remainder of the survey. Therefore, we would have them skip the remainder of the survey. This occured 9 times, causing the number of responses for the remaining questions to go down to 269.

The right-most graph of Figure 4 shows a quite diverse answer to whether participants use code coverage tools outside of GitHub, for example on their local device while developing.

General attitude towards coverage and testing. When asked about the general necessity of automated software testing and code coverage, most participants responded that they consider testing and code coverage as important, as can be seen in Figure 5. However, not all respondents feel this way, as one person indicated that automated software testing is not at all important.

Frequency of writing tests. Figure 6 shows the results for two questions asking how often the participants write tests. An interesting observation is that 0 respondents answered that they never write tests. Even the one participant who answered that automated software testing is not important at all still writes tests. Furthermore, we also find that *"For each pull request"* is the most popular answer for the first question. Interestingly, the second graph indicates that respondents frequently write tests with the purpose of improving coverage.

Coverage and incentive. Keeping in mind that the middle graph of Figure 6 indicates that most participants write tests with the sole purpose of improving coverage, we wonder about the potential incentive that code coverage tools provide for writing tests and/or improving the coverage metric(s) of a code base. When we consider the right-most graph in Figure 6, we see that almost all participants agree, either somewhat or strongly, that code coverage tools provide an incentive to improve coverage, with no big difference of opinion between the contributors and the maintainers.

Secondly, we asked contributors and maintainers near identical questions (Q13c and Q13d in Table 1) regarding the acceptance rate of pull requests, based on their coverage levels. In Figure 7 we observe a difference of opinion between what contributors believe,

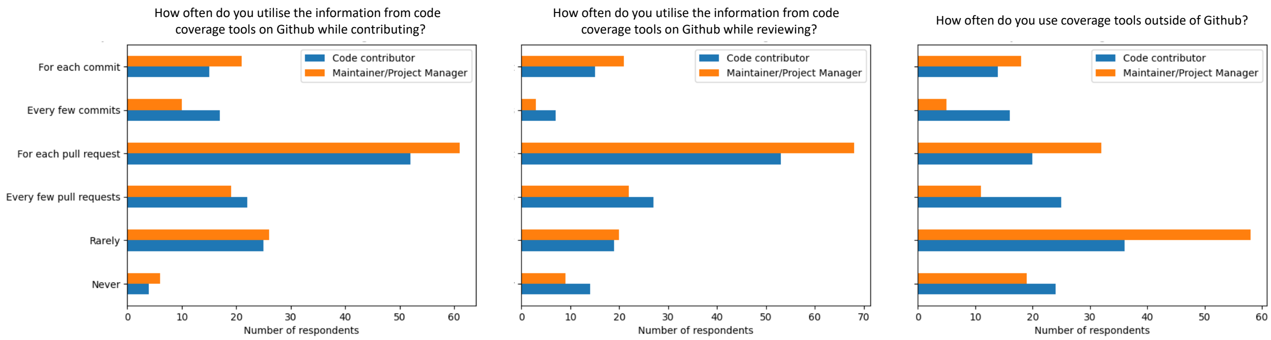

Figure 4: Code coverage tool frequency, and coverage tool usage outside of GitHub

versus what maintainers claim: maintainers indicate they are more likely to accept a pull request which improves coverage, while contributors indicate to think that a coverage-improving pull request will not get accepted more quickly compared to other pull requests.

We also asked participants to rank different features of code coverage tools, from what they consider providing the most to the least incentive to improve coverage. In Figure 8 we graph both the number of times a feature has been put in first position, as well as its average rank, with 1 being the highest, and 5 being the lowest. The *status check* is by far ranked as the most incentivising feature. Nevertheless, every other feature has also been ranked first by one or more participants at some point. We observe that contributors and maintainer share the same opinion.

Neglecting the coverage check. Figure 9 shows the frequency with which the participants ignore a failing coverage check when both contributing and reviewing. Of interest to observe is that >80% of the respondents across categories indicate to at least sometimes ignore failing coverage checks. Additionally, we see that maintainers tend to ignore failing coverage checks slightly more frequently.

Pointing out decreasing coverage. Figure 10 provides insight into how often the respondents are asked to improve coverage, or need to ask other contributors to improve coverage.

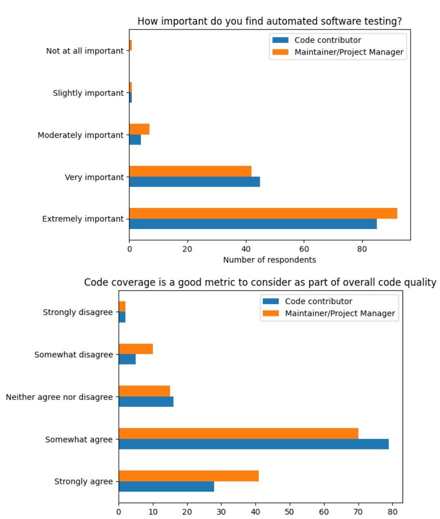

Figure 5: Importance of testing and code coverage

A first observation is that a large group of contributors indicate that they have never been asked to improve their coverage by a human, and an even larger group says they have never been asked by a coverage tool. Secondly, it appears that overall coverage tools are perceived to ask to improve coverage more frequently than humans. Thirdly, when we compare maintainers to contributors, we see very similar numbers, especially for the *"By a human"* bars, with the exception for the *"A few times a month"* option. A possible explanation could be that maintainers review many pull requests from different contributors per month, but contributors might not have their pull requests reviewed by many different reviewers per month, i.e., maintainers overall review more varied pull requests.

3.3 Qualitative Insights

For our analysis of the qualitative (open) questions, we applied open and axial coding [40] throughout multiple rounds of analysis. We started by manually assigning each answer one or more codes; if a response was unclear and could not be given a code, it was discarded. To check for consistency and agreement in our coding process, the first and second authors performed the open coding of the first 25 participants' answers separately. Then, these two researchers discussed the emergent codes and reached a negotiated agreement in a hands-on meeting. After reaching an agreement, the first author coded the remaining answers.

Afterwards, we performed axial coding by grouping the codes into larger categories of responses. For each question, we listed the codes that were generated from the open coding process and assigned them into groups based on how closely the codes were related to one another or a particular topic. In the case a code could not be properly categorized, it was added to a *"Miscellaneous"* category. We then retrofitted the categories back to the original responses, based on which codes they had. Finally, we counted how many responses belonged to each category. The axial coding was performed by the first author, and discussed with the second and third authors until reaching consensus. The coding process was conducted using a continuous comparison technique [15], wherein we continuously compared emerging codes with existing findings to validate our interpretation. The full list of responses and codes can be found in our online replication package [39].

As mentioned above, we counted the number of responses per category, to find how many participants mentioned a specific category in their responses to each question. It is possible for a response to mention multiple categories, and therefore count for multiple

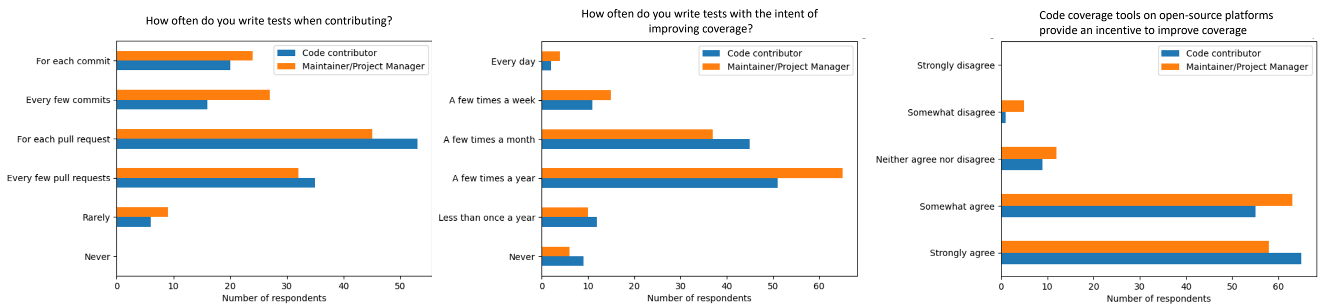

Figure 6: Frequency of writing tests and code coverage improvement

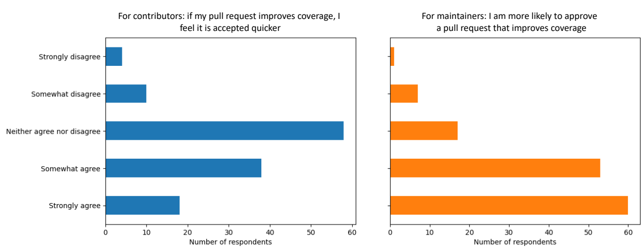

Figure 7: Beliefs about pull request acceptance and code coverage

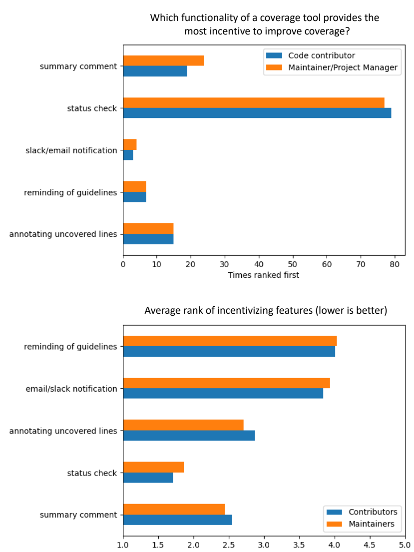

Figure 8: Ranking the most incentivising feature of code coverage tools

categories. Furthermore, the results are grouped per participants' roles as either a contributor or a maintainer again. These results can be found in Tables 3 through 5. We will now go through them, one question at a time.

Good coverage goal. The first open question we asked to the participants, was for them to describe their own idea of a good coverage goal to set for a project. The first thing that piques our

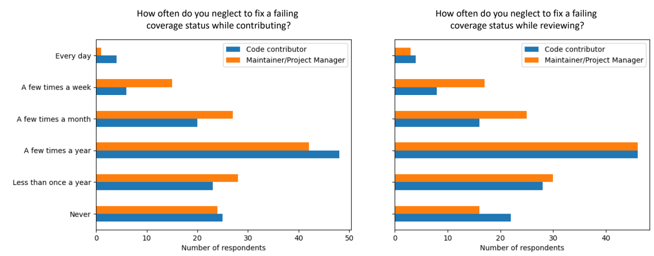

Figure 9: Frequency of ignoring a coverage check

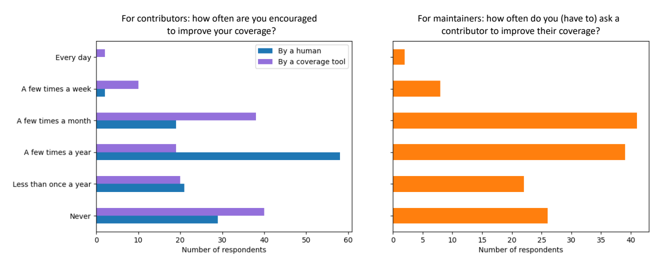

Figure 10: Frequency of asking to improve coverage

interest when looking at the first question in Table 3, is that for each category of responses the ratio of participants that mention it is (almost) the same for both contributors and maintainers. Secondly, nearly 50% of all respondents mention some kind of high coverage goal in their responses. However, for both groups at least 20% of the participants believe that striving to achieve some arbitrary target is not the right goal to set.

Best way to incentivize for coverage. Previously, we asked respondents to rank different features of coverage tools, based on how much incentive they provide. For this question however, participants were able to give their own input on the best way to encourage writing tests and improving coverage. The results can be found in the 2nd part of Table 3.

We observe that participants most often list negative impacts as the best way to provide incentive to improve coverage. This typically comes down to blocking the merge of the pull request, or closing it all together. One caveat with these responses is that the more popular answers (negative impacts, using PR comments, getting a coverage report) are features that coverage tools already

Category	Summary	Example	Contributor mentioned	Maintainer mentioned
Q20: What is a good coverage goal?				
High number	Goal is to have high coverage	"Total coverage should be high, e.g. > 80% Each PR should not have diff coverage < 85%"	71 (53%)	79 (55%)
Important code only	Goal is to cover important parts of the code. Trivial code can be uncovered	"Tests should cover the important code paths, beyond the "happy path". However, 100% coverage is neither necessary nor particularly desirable"	25 (19%)	25 (17%)
It depends per project	Coverage goal depends on multiple factors. Lot of different variables	"Every project would have a different goal: UI and integration testing is much harder to test than unit testing core business logic, and different projects have different proportions of each"	24 (18%)	23 (16%)
Cover sensibly	Goal is to have everything covered in a sensible matter, where uncovered items are justified properly	"In general, coverage should not be allowed to fall without justification. Coverage of, and benefits from, unit and integration tests should generally be considered separately"	4 (3%)	3 (2%)
Quality more important	Not a lot of focus on coverage, since it distracts from actually writing high quality code/tests	"A few years ago I spent a lot of time writing coverage tests. Then I gave up: the tooling got too much in the way, it took too much time, and I decided to focus on real problems instead."	33 (24%)	34 (24%)
Q31: What is the best way to incentivize improving coverage?				
Interaction w/ comments	Interaction through PR comments	"Generated comment on PR"	16 (12%)	19 (13%)
Negative impacts	Negative impacts, such as blocking merge	"Not merging when a PR has no tests"	36 (27%)	37 (26%)
Clear expectations	Setting expectations or contributing guidelines	"Make it normative and expected for a project."	15 (11%)	14 (10%)
Coverage tool	Generic mention of using a coverage tool	"Providing an adequate report of test coverage that includes coverage status, changes, and uncovered lines."	27 (20%)	30 (21%)
Notifications	Notifications	"A failing status with notification about what is unit tests and how to add them"	3 (2%)	3 (2%)
Focused testing	Clear/Focussed testing efforts. E.g. making it easy to write tests, or providing a proper guide to testing	"Make it easy to test your code. If it is difficult to write tests, people will not write them."	4 (3%)	9 (6%)
Positive reinforcement	Positive reinforcement, such as using gamification. Or general politeness	"Gamification on getting a high coverage score"	1 (1%)	6 (4%)
Good feeling	Getting feelings of safety, security and trust in the code	"My primary incentive for good test coverage is more confidence to deploy changes"	3 (2%)	0 (0%)
Deeper understanding of necessity	Understanding of the necessity of well-tested code provides the incentive	"Demonstrating the improvements to reliability of tested vs untested software :) Vanity metrics like % code coverage don't convince people that don't care about testing; not getting paged due to bugs does."	6 (4%)	8 (6%)

Table 3: Summarising what is a good coverage goal, and how to incentivize improving coverage.

provide, and as such they easily come to mind. Especially since we already mentioned them previously in the survey. Therefore, it might be worthwhile to look at other categories. For example, 10% of participants believe that setting clear expectations, or writing contribution guidelines, could be a way to incentivize for coverage.

Likes and dislikes. We asked the participants to give us two reasons why they like using coverage tools, and two things they dislike about them. The first part of Table 4 shows the likes, the later part lists the dislikes.

When it comes to *likes*, we observe that a couple of categories are mentioned quite frequently by both groups. For example, collecting all coverage information into a single report or online place, or the feeling of safety it gives, to know that the code is properly tested. One interesting result to see, is the big difference for the *"encourages testing"* category: contributors mention this twice as often as the maintainers.

However, there is a similar big difference when it comes to *dislikes*. Namely in the *"poor feedback"* category. Some examples given by the participants are that the bigger picture is unclear, and that the results are hard to interpret, or offer no guidance on what to do next. Maintainers mention this dislike twice as often. We would expect this, since maintainers might be more involved with and interested in the overall state of a project's coverage throughout its development. Similarly, this would also hold true for the *"third-party host concerns"*.

Another dislike is that a decent percentage of contributors mention the difficult setup as something they dislike. This is something we would expect from maintainers, but not from contributors, since the whole point of code coverage tools is to have them on something like GitHub, running during the CI pipeline. This means that contributors would not have to set them up themselves. Moreover, in the list of *likes* we find that 10% of both groups enjoy the low entry barrier and easy set-up for coverage tools, while in the dislikes table we find that there is another set of respondents that claim the exact opposite.

The largest overall expressed dislike by both groups, is that code coverage tools lead to people treating code coverage as a form of code quality. This is not necessarily the case, since it is entirely possible to cover all the lines of code in a project, without actually verifying whether the output is correct [23]. The second largest

Category	Summary	Example	Contributor mentioned	Maintainer mentioned
Two things you like about coverage tools?				
Honesty	Honesty of the developer	"The public nature of open-source software helps keep people honest."	4 (3%)	2 (1%)
Automation	Automated tool	"I like the automation and the ease of adding it to new product."	22 (16%)	21 (15%)
Awareness	Awareness of other people's contributions	"Safe teamwork and knowledge of other people's code"	0 (0%)	1 (1%)
Finding improvements	Finding weaknesses or code you can improve	"Helps you find dead code and untested code."	25 (19%)	23 (16%)
Community	Community Support	"the community support available when things don't work."	0 (0%)	1 (1%)
Easier work	Easier/faster contributing, reviewing, etc.	"Quickly discovering if new code has tests is really helpful, especially as a maintainer."	9 (7%)	10 (7%)
Low entry barrier	Low entry barrier to get started	"Easy to set up. Easy to maintain."	13 (10%)	15 (10%)
Encourages testing	Encourages testing	"Code coverage tools incentivise developers to write test."	21 (16%)	12 (8%)
Guarantees	Safety/security/trust feelings	"It definitely adds a bit of credibility to a project."	30 (22%)	34 (24%)
Gamification	Gamification	"Gameify's software testing so that it is easier to approach and incentivizes having more thoroughly tested software."	1 (1%)	1 (1%)
UI/UX Design	Design	"GUI for exploring historical code coverage"	2 (1%)	3 (2%)
Collected information	Collection of coverage information	"No more "but the coverage is different on my machine"."	36 (27%)	29 (20%)
Prevents coverage decrease	Prevents coverage from going down	"It prevents to add code that lowers the code coverage"	0 (0%)	1 (1%)
Public	Looks good to the outside world	"The results can be viewed by anyone."	6 (4%)	8 (6%)
Sets guidelines	Guidelines and/or protocols	"Creates a culture of test driven development"	6 (4%)	4 (3%)
Quality control	It provides some kind of quality control	"It helps keeping project code clean and fresh although outside contributes codes are merged."	10 (7%)	9 (6%)
Educational	Can be used as a learning tool	"Can be a good bridge to introduce contributors to automated tests and QA."	1 (1%)	1 (1%)
Useful functions	It has useful features (generic)	"free, useful"	1 (1%)	2 (1%)
Two things you dislike about coverage tools?				
Complacency	People become complacent	"Most devs stop if they hit the quality goals of the repo"	4 (3%)	2 (1%)
Set-up	Complex Set-up	"setting up & maintaining code coverage CI can be tedious"	20 (15%)	19 (13%)
Mistaken for quality	Coverage metric treated as quality	"It can encourage developers to write shoddy tests that don't properly cover use cases in order to increase the coverage metric"	26 (19%)	27 (19%)
Reluctance to use	Some people are reluctant to use them	"Many developers are still not used to deal with such tools: overhead to enforce their use"	2 (1%)	4 (3%)
Unclear/wrong results	Unclear, wrong or insignificant results	"It is flaky and often gives incorrect results / misleading information."	26 (19%)	22 (15%)
Annoying	Considered frustrating or interrupt workflow	"Can be annoying during initial development activities"	3 (2%)	3 (2%)
Not always applicable	The tool is hard to use depending on the project	"Hard to introduce after a while (low coverage), if not introduced early."	2 (1%)	1 (1%)
Third-party host	Third-party host concerns	"Cloud services can disappear or get monetized at any moment; you have to have an offline local way to measure coverage"	7 (5%)	14 (10%)
Lacking features	Respondent mentioned features they miss	"not yet enough AI to provide good actionable advice"	8 (6%)	10 (7%)
Poor feedback	Tool provides mediocre feedback	"Doesn't offer suggestions on how to write _good_ tests."	12 (9%)	24 (17%)
Noisy	Noisy, redundant or verbose	"Adds PR comment / commit status noise"	10 (7%)	14 (10%)
Takes time	Generating coverage results takes a long time	"Steals valuable time from contributors."	5 (4%)	8 (6%)
Strict	Too strict	"Communities that use the tool too rigidly are unpleasant to contribute to."	10 (7%)	13 (9%)
Crutch	Used as a crutch	"It can be a crutch to write bad tests that just increase coverage"	0 (0%)	1 (1%)

Table 4: Likes and dislikes of coverage tools according to the respondents.

dislike is that the results of the coverage tool can be incorrect, irrelevant, or too small to really matter [18].

Why are coverage checks ignored. Finally, we move on to one of the most important questions of the survey. We would like to find out in what situations the participants would ignore a failing coverage check. We asked this to the participants in the form of an open question, and the results can be found in Table 5.

Firstly, it seems coverage failures tend to be ignored when they are minimal, e.g., less than 1% coverage delta. Something like this can happen when only a few lines of code are added, or perhaps deleted, or by introducing an extra branch, or removing a test case, which happens on occasion [36]. Out of all the reasons, this one is given most often by the maintainers.

Secondly, the coverage failure can be a result of another failure, for example in the CI pipeline. Another reason that is given is that the coverage information is simply incorrect, or unrelated to the PR in question.

The third reason is priority. Some issues need to be merged quickly, as they contain fixes for critical bugs, or very desirable features. For contributors this is the most frequently cited reason. Here we see a small difference of opinion between contributors and maintainers.

Category	Summary	Example	Contributor mentioned	Maintainer mentioned
Trivial change	Trivial/negligible coverage change	"if the code coverage failures are due to insignificant (trivial) branches not being covered."	32 (24%)	40 (28%)
Failure elsewhere	Failure unrelated to PR	"Unfortunately some of the results can be odd, not explained. Like a commit that just touches documentation (without example code) can change the coverage."	28 (21%)	34 (24%)
Future fix	Assurance that tests are added/coverage will be fixed later	"The code is good, it is better to have it in the codebase earlier despite not having the coverage. Coverage can be developed later."	22 (16%)	20 (14%)
Not worth fixing	Not worth time/effort to improve the coverage	"Because coverage is not important. And big coverage is expensive."	16 (12%)	8 (6%)
Priority	Different priorities such as hotfixes are desirable features	"If the PR clearly solves a primary problem, I am going to merge it. Lower coverage is a secondary problem."	36 (27%)	28 (20%)
Complexity	Complexity of the code	"If the test is very difficult to properly implement, like having to simulate external dependencies."	32 (24%)	31 (22%)
Other means	Ensuring quality is done through other means than coverage	"Code should be well tested, but code coverage isn't synonymous to how well code is tested, it's just correlated"	12 (9%)	10 (7%)
Not production code	Code not meant for production (yet)	"The feature added is experimental and the tests are failing"	9 (7%)	9 (6%)
Not scaring contributors	To avoid scaring away contributors	"It shouldn't become prohibitive of pushing a PR to not discourage people to contribute"	1 (1%)	8 (6%)
Change justified	Change justified	"instead of merging the PR with a failing check, we would instruct the user on how to ignore the line or the file in question if it was suitably justified."	0 (0%)	2 (1%)

Table 5: When would you ignore a failing coverage check?

The fourth reason is simply the complexity it takes to write tests and improve the coverage. Not all code is equal and can be tested as easily, e.g., code for a GUI is not the same as some business logic [10].

A final interesting finding is that contributors mention that they do not find fixing coverage worth their efforts, at twice the rate than maintainers do. Overall, it seems that contributors are slightly more concerned with just getting their code merged, and less with testing. This can also be said for the results of the *"priority"* category. And on the other hand, we see that for example maintainers are more likely to mention that they do not want to scare away contributors.

4 DISCUSSION & IMPLICATIONS

4.1 Revisiting the Research Question

Through the responses to our survey, we have found indications that failing coverage checks are sometimes ignored. Overall, we have observed that >80% of the respondents have indicated that they ignore failing coverage checks at some point. We also uncovered four key reasons for ignoring coverage checks, namely: (1) a small coverage delta [10], (2) coverage calculation failures or mistakes, (3) priority lies with production code [30, 45, 46], and finally, (4) the difficulty of testing complex code [14, 41].

In what follows, we try to formulate recommendations for open-source developers and developers of code coverage tools to potentially alleviate some of the aforementioned issues.

4.2 Recommendations for Open-Source Developers

One of the main reasons for ignoring a failing coverage check is that the failure is due to a minimal decrease in coverage. One way to alleviate this problem is by configuring their coverage tools with a certain threshold, so only larger decreases trigger an actual failure[3]. Furthermore, developers can also configure the coverage tools to decrease the noise they create. A practical approach would be to group multiple bot comment reports into a single report, which was demonstrated as useful in a paper by Wessel and Steinmacher [44].

Another big reason for neglecting fixing the coverage is simply the high complexity of testing (certain parts of) the code. This is a call to arms to developers to ensure that the code is testable [11], and potentially refactored in a test-driven way [34].

Another recommendation is writing clear guidelines for contributing, but with a clear goal for testing and/or coverage targets. The answers to the question *"What is a good coverage goal?"* in Table 3 can provide some inspiration for contribution guidelines.

4.3 Recommendations for Code Coverage Tool Developers

The *dislikes* listed in Table 4 provide developers of code coverage tools with a list of opportunities to improve their tools. Some of the problems that are mentioned by both maintainers and contributors are: unclear or wrong results, a difficult setup process, and noise.

Zooming in specifically on the matter of robustness of results, it might be worthwhile to investigate ways to ignore individual methods or lines that cause problems, e.g., through annotations in source code. Another angle here is to annotate flaky tests, so that developers better understand where fluctuations in code coverage is coming from (confer [42]).

A large reason for why failing coverage checks are ignored is because the contributors and/or maintainers of a project have other priorities, do not have enough time to immediately address the failure, or because they are working on experimental code, which might change a lot in the future. Therefore, we think it would

[3]For example: https://docs.codecov.com/docs/codecov-yaml, last visited October 4th, 2023.

be beneficial if code coverage tools provided an optional integration with issue tracking systems (i.e., GitHub issues, Jira, etc.), to (automatically) open new issues for uncovered, but merged, code changes. From a developer's perspective this would make it easier to track the testing technical debt [29] with a system they are likely already familiar with. Similarly, an integration with a tool like TestKnight, that creates boilerplate code for tests yet to be written, could prove interesting [9]. Finally, a tool like TestAxis could be extended to report coverage information from cloud-based continuous integration services directly into the IDE [8].

4.4 Threats to Validity

Internal validity. One of the main threats to validity is the possible bias we introduced while categorising the qualitative responses. To mitigate this threat, we used a process in which the first responses for each open question were analysed by the first and second author, who then compared their categories. During this process, no major differences were found. Subsequently, the other responses were categorised by the first author, and later verified by the second author.

In order for participants to get acquainted with the idea of code coverage tools, we used some screenshots of functionality provided by Codecov. This could have unintentionally steered participants to think of Codecov in particular, when we asked them about their experiences with any tool. We tried to mitigate this threat by also making references to other possible code coverage tools, and consistently using the term "code coverage tool" in our questioning.

External validity. We were limited by both time constraints and GitHub's API to invite and include all potential users of coverage bots for our survey. As such, we had to narrow our search by first querying people from Codecov's database. This introduces a bias in our results, as while all our participants have used Codecov before, it does not mean that they have used other tools before. In future work, we intend to widen the scope of our investigation to other code coverage tools.

We randomly sampled 2,000 users of Codecov to invite them for our survey. We ended up with an almost even split of 135 contributors and 143 maintainers. This means that we could gather opinions and experiences from different perspectives and developer profiles. In future work, we intend to study more developers, who also use a variety of code coverage tools.

5 RELATED WORK

Ivanković et al. have studied code coverage challenges and experiences at Google [22]. Through a survey among 512 employees of Google they observe that *"a substantial number of developers do use code coverage on a regular basis and find value in it"*. Their study also highlights how developers find it useful to *obtain coverage information within the tools developers commonly use*, something that relates to how the Codecov bot works on GitHub. Also similar to our study, they mention that errors in code coverage numbers are consuming time and energy from software engineers.

Elbaum et al. have established that even small changes in production code, can cause big fluctuations in code coverage values [13].

The respondents to our survey have indicated to regularly write additional test cases to improve the code coverage. The study by Kochar et al. puts this into perspective [27]. In particular, their findings are that a relationship between code coverage and post-release defects is non-existent or statistically insignificant. As such, they warn that *designing test cases with the sole purpose of increasing coverage may or may not translate to higher bug finding rate*. Our findings highlight that although coverage is commonly used as a yardstick for test adequacy, its impact should not be overestimated.

Beller et al. have investigated the reasons for failing continuous integration builds [7]. They have observed that the key reason for build failures are test failures. Instead, our work looks at failing status checks caused by missed code coverage targets.

Khatami et al. have looked into the awareness of software engineers about the quality assurance practices in vogue in the GitHub projects that engineers contribute to [25]. Through a large-scale survey they have established that ~80% of the respondents indicate that the coverage their project reaches is easy to retrieve, and that ~37% is aware of coverage targets.

Hilton et al. have studied code coverage evolution [20]. They found that measuring the patch coverage, like the Codecov tool that we study enables, provides visibility into the impact of the changes that software engineers make.

There are a number of other studies in the field of code coverage, but they are typically based on a few projects, based in a single language [19, 21, 35], or at a single corporate entity [22, 28]. The results of these studies might therefore not necessarily generalize well to open-source development. Furthermore, most of these studies measure the impact of code coverage on the development or testing processes, but not the impact on the developers themselves.

6 CONCLUSION & FUTURE WORK

In this study we have broadly investigated the perceptions and opinions of open-source software engineers on code coverage tools in general, and Codecov specifically. More specifically, we tried to establish *why code coverage checks are ignored?*. Through an online survey among 279 open-source software engineers of which the projects make use of Codecov, we find that >80% of the software engineers have at some point ignored code coverage checks. We also find four key reasons for ignoring these checks, namely: (1) a small coverage delta for the particular commit or pull request, (2) a failure to compute code coverage, or wrong code coverage results, (3) a difference in priorities as production code is deemed more important, and finally, (4) the difficulty of testing complex code.

In future research, we aim to better understand the role of contribution guidelines on the process of working with code coverage tools. We will also look into the relationship between test flakiness and fluctuations in code coverage reports. In particular, we see opportunities to (1) better understand the influence of flaky tests on code coverage results, and (2) investigate whether excluding flaky tests from coverage calculations, e.g., through annotations in code, would help software engineers work better with coverage tools.

ACKNOWLEDGMENTS

This research was partially funded by the Dutch science foundation NWO through the Vici "TestShift" grant (No. VI.C.182.032) and conducted as part of Alexander Sterk's master thesis [38].

REFERENCES

[1] Maurício Aniche. 2022. *Effective Software Testing: A Developer's Guide.* Manning Publications.

[2] Maurício Aniche, Felienne Hermans, and Arie van Deursen. 2019. Pragmatic software testing education. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education.* 414–420.

[3] Maurício Aniche, Christoph Treude, and Andy Zaidman. 2022. How Developers Engineer Test Cases: An Observational Study. *IEEE Trans. on Software Eng.* 48, 12 (2022), 4925–4946.

[4] Baris Ardiç and Andy Zaidman. 2023. Hey Teachers, Teach Those Kids Some Software Testing. In *5th IEEE/ACM International Workshop on Software Engineering Education for the Next Generation (SEENG@ICSE).* IEEE, 9–16.

[5] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. 2019. Developer Testing in the IDE: Patterns, Beliefs, and Behavior. *IEEE Trans. Software Eng.* 45, 3 (2019).

[6] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE).* ACM, 179–190.

[7] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, my tests broke the build: an explorative analysis of Travis CI with GitHub. In *Proceedings of the International Conference on Mining Software Repositories (MSR).* IEEE, 356–367.

[8] Casper Boone, Carolin Brandt, and Andy Zaidman. 2022. Fixing Continuous Integration Tests From Within the IDE With Contextual Information. In *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC).* ACM, 287–297.

[9] Cristian-Alexandru Botocan, Piyush Deshmukh, Pavlos Makridis, Jorge Romeu Huidobro, Mathanrajan Sundarrajan, Maurício Aniche, and Andy Zaidman. 2022. TestKnight: An Interactive Assistant to Stimulate Test Engineering. In *IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE).* ACM/IEEE, 222–226.

[10] Carolin Brandt, Marco Castelluccio, Christian Holler, Jason Kratzer, Andy Zaidman, and Alberto Bacchelli. 2024. Mind the Gap: What Working With Developers on Fuzz Tests Taught Us About Coverage Gaps. In *Proceedings of the International Conference on Software Engineering - Software Engineering In Practice (ICSE-SEIP).* ACM.

[11] Magiel Bruntink and Arie van Deursen. 2006. An empirical study into class testability. *J. Syst. Softw.* 79, 9 (2006), 1219–1232. https://doi.org/10.1016/j.jss.2006.02.036

[12] Codecov. 2022. The leading code coverage solution. https://about.codecov.io/

[13] S. Elbaum, D. Gable, and G. Rothermel. 2001. The impact of software evolution on code coverage information. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001.* IEEE, 170–179.

[14] Michael Feathers. 2004. *Working Effectively with Legacy Code.* Pearson.

[15] Barney G Glaser and Anselm L Strauss. 2017. *Discovery of grounded theory: Strategies for qualitative research.* Routledge.

[16] Georgios Gousios, Margaret-Anne D. Storey, and Alberto Bacchelli. 2016. Work practices and challenges in pull-based development: the contributor's perspective. In *Proceedings of the 38th International Conference on Software Engineering (ICSE).* ACM, 285–296.

[17] Georgios Gousios, Andy Zaidman, Margaret-Anne D. Storey, and Arie van Deursen. 2015. Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective. In *37th IEEE/ACM International Conference on Software Engineering (ICSE).* IEEE, 358–368.

[18] Z. Guo, T. Tan, S. Liu, X. Liu, W. Lai, Y. Yang, Y. Li, L. Chen, W. Dong, and Y. Zhou. 2023. Mitigating False Positive Static Analysis Warnings: Progress, Challenges, and Opportunities. *IEEE Transactions on Software Engineering* 49, 12 (2023), 5154–5188.

[19] Hadi Hemmati. 2015. How Effective Are Code Coverage Criteria?. In *2015 IEEE International Conference on Software Quality, Reliability and Security (QRS).* IEEE, 151–156.

[20] Michael Hilton, Jonathan Bell, and Darko Marinov. 2018. A Large-Scale Study of Test Coverage Evolution. In *Proceedings of the International Conference on Automated Software Engineering (ASE).* ACM, 53—-63.

[21] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. 1994. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In *Proceedings of 16th International Conference on Software Engineering (ICSE).* IEEE, 191–200.

[22] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 2019. Code Coverage at Google. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).* ACM, 955—-963.

[23] Kush Jain, Goutamkumar Tulajappa Kalburgi, Claire Le Goues, and Alex Groce. 2023. Mind the Gap: The Difference Between Coverage and Mutation Score Can Guide Testing Efforts. *CoRR* abs/2309.02395 (2023). https://doi.org/10.48550/arXiv.2309.02395 arXiv:2309.02395

[24] Mehdi Jazayeri. 2004. The Education of a Software Engineer. In *Proc. International Conference on Automated Software Engineering (ASE).* IEEE, xviii–xxvii.

[25] Ali Khatami and Andy Zaidman. 2023. Quality Assurance Awareness in Open Source Software Projects on GitHub. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM).* IEEE, 174–185.

[26] Amy J. Ko, Bryan Dosono, and Neeraja Duriseti. 2014. Thirty years of software problems in the news. In *Int'l Workshop on Cooperative and Human Aspects of Software Engineering (CHASE).* ACM, 32–39.

[27] Pavneet Singh Kochhar, David Lo, Julia Lawall, and Nachiappan Nagappan. 2017. Code Coverage and Postrelease Defects: A Large-Scale Study on Open Source Projects. *IEEE Transactions on Reliability* 66, 4 (2017), 1213–1228.

[28] Chandra Maddila, Chetan Bansal, and Nachiappan Nagappan. 2019. Predicting Pull Request Completion Time: A Case Study on Large Scale Cloud Services. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).* ACM, 874—-882.

[29] Everton da S. Maldonado and Emad Shihab. 2015. Detecting and quantifying different types of self-admitted technical Debt. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD).* 9–15. https://doi.org/10.1109/MTD.2015.7332619

[30] Cosmin Marsavina, Daniele Romano, and Andy Zaidman. 2014. Studying Fine-Grained Co-evolution Patterns of Production and Test Code. In *14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM).* IEEE, 195–204.

[31] Scott Matteson. 2018. Report: Software failure caused 1.7 trillion in financial losses in 2017. https://www.techrepublic.com/article/report-software-failure-caused-1-7-trillion-in-inancial-losses-in-2017/

[32] Harry McCracken. 2017. The Year That Software Bugs Ate The World. https://web.archive.org/web/20230307155438/https://www.fastcompany.com/40505226/the-year-that-software-bugs-ate-the-world

[33] Greg Miller. 2006. A Scientist's Nightmare: Software Problem Leads to Five Retractions. *Science* 314, 5807 (2006), 1856–1857.

[34] Leon Moonen, Arie van Deursen, Andy Zaidman, and Magiel Bruntink. 2008. On the Interplay Between Software Testing and Evolution and its Effect on Program Comprehension. In *Software Evolution*, Tom Mens and Serge Demeyer (Eds.). Springer, 173–202.

[35] Akbar Siami Namin and James H. Andrews. 2009. The Influence of Size and Coverage on Test Suite Effectiveness. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA).* ACM, 57—-68.

[36] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. 2012. Understanding Myths and Realities of Test-Suite Evolution. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE).* ACM, Article 33, 11 pages.

[37] Igor Steinmacher, Gustavo Pinto, Igor Scaliante Wiese, and Marco A. Gerosa. 2018. Almost There: A Study on Quasi-Contributors in Open Source Software Projects. In *Proceedings of the International Conference on Software Engineering (ICSE).* ACM, 256—-266.

[38] Alexander Sterk. 2023. *Exploring Code Coverage in Open-Source Development.* Master's thesis. Delft University of Technology.

[39] Alexander Sterk, Mairieli Wessel, Ali Hooten, and Andy Zaidman. 2023. Running a Red Light: An Investigation into Why Software Engineers (Occasionally) Ignore Coverage Checks — Appendix. https://doi.org/10.5281/zenodo.10119287

[40] Anselm L Strauss and JM Corbin. 1998. *Basics of qualitative research: Techniques and procedures for developing grounded theory.*

[41] Mark Swillus and Andy Zaidman. 2023. Sentiment overflow in the testing stack: Analyzing software testing posts on Stack Overflow. *J. Syst. Softw.* 205 (2023), 111804.

[42] Shivashree Vysali. 2020. *Enriching Code Coverage with Test Characteristics.* Master's thesis. McGill University, 3480 Rue University, Montréal, QC, Canada.

[43] Mairieli Wessel, Alexander Serebrenik, Igor Wiese, Igor Steinmacher, and Marco A. Gerosa. 2020. What to Expect from Code Review Bots on GitHub? A Survey with OSS Maintainers. In *Proceedings of the Brazilian Symposium on Software Engineering (SBES).* ACM, 457–462.

[44] Mairieli Wessel and Igor Steinmacher. 2020. The Inconvenient Side of Software Bots on Pull Requests. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSE Workshops).* ACM, 51–55.

[45] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie van Deursen. 2008. Mining Software Repositories to Study Co-Evolution of Production & Test Code. In *Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST).* IEEE, 220–229.

[46] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. 2011. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empir. Softw. Eng.* 16, 3 (2011), 325–364.

[47] Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software Unit Test Coverage and Adequacy. *ACM Comput. Surv.* 29, 4 (Dec. 1997), 366–427.