Exploring Code Coverage in Open-Source Development

Master's Thesis

Alexander Sterk

Exploring Code Coverage in Open-Source Development

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Alexander Sterk
born in Rijswijk, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Codecov
45 Fremont St San Francisco, CA 94105
https://about.codecov.io/

Exploring Code Coverage in Open-Source Development

Author: Alexander Sterk
Student id: 4542800

Abstract

Software development has increasingly become an activity that is (partially) done online on open-source platforms such as GitHub, and with it, so have the tools developers typically use. One such category of tools is that of code coverage tools. These tools track and report coverage data generated during CI tests. As the adoption of these tools has grown, so does the amount of available coverage data. In this thesis we explore a large database of coverage data from Codecov, a popular coverage tool. What sets our work apart from existing research is that it spans a large number of projects which vary in size, language, and domain. Furthermore, we conduct a survey, which was disseminated among a wide variety of open-source developers, instead of at a single company or in an enterprise setting. Our research consists of three parts. Firstly, we assess whether there is a relationship between the time to merge a PR and its coverage levels. We find that such a relationship does exist in certain projects. Secondly, we look at the impact of PR comments mentioning coverage on the odds of said coverage improving. Using the odds ratio test, we conclude that there are greater odds of coverage improving when it is mentioned than when it is not. Thirdly, we conduct a survey to ask developers their reasons for ignoring a failing status check related to code coverage. Some reasons they give are the complexity of testing, the triviality of the proposed changes, or the pull request being too important to wait for proper testing. Furthermore, respondents who identify as code contributors find themselves twice more likely to find fixing coverage a waste of their time than those who identify as code maintainers, while code maintainers are more concerned with not scaring away new contributors with strict coverage guidelines.

Thesis Committee:

Chair:	Prof. Dr. A. Zaidman, Faculty EEMCS, TU Delft
Daily supervisor:	Dr. M. Wessel, Radboud University
Company supervisor:	E. Hooten, Codecov
Committee Member:	Dr. R. Hai, Faculty EEMCS, TU Delft

Preface

As I am sitting now, writing this preface, I realize this is the last part of my thesis that I still have to write. Like most things, I thought writing it would be easy and come naturally to me. However, I once again find myself struggling with finding the right words.

Over the last two years of working on my thesis, I have grown more accustomed to this feeling. But it still feels strange. I can honestly say there has never been a project I have worked as hard on, or worried about as much, as this one.

Whilst writing this thesis, I've learnt a lot about myself. I procrastinate, I worry, and when I procrastinate, I worry, and when I worry, I procrastinate. Breaking this cycle has been difficult. It took me two years... Needless to say, I am very thankful that I am nearly done, and proud of the work I have done so far.

In particular, I am thankful to a few people. Mainly, I'd like to thank my friends and family for their support. They've had to endure a lot of (needless) worrying from my end, and I am grateful for the listening ears they provided. I am also grateful for my colleagues, who have been patient with me while I finished up this project.

I'd also like to thank Mauricio Aniche, for introducing me to Codecov and this thesis topic, and for supervising me for the first research questions. I'd like to thank Andy Zaidman, for taking over as my supervisor, and for all his feedback in the final stages. I'd like to thank Codecov, for providing me with this opportunity and allowing me access to their database. I'd especially like to thank Eli Hooten from Codecov, for his supervision, his insight into academic writing, and our biweekly[1] meetings. And last but not least, I'd like to thank Mairieli Wessel for being my daily supervisor, answering each of my questions, verifying all my work, and for helping me, even after moving to another university.

And finally, I'd like to thank you, dear reader, for taking an interest in my work. May you find it useful, and of course, enjoyable to read.

Alexander Sterk
Delft, the Netherlands
June 19, 2023

[1] every two weeks

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software is a critical aspect of modern society and ensuring that code is well-tested and reliable is a crucial part of the development process. One way to make this process easier is by using code coverage.

Code coverage refers to the percentage of code that is executed by a test suite, and it is an important metric in assessing the quality of tests and identifying areas of code that may need further testing [31]. One of the most 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 [27, 4].

There are a number of existing studies in the field of code coverage, but they typically are based on a few projects, based in a single language [20, 10, 8] or at a single corporate entity [17, 12]. The results of these kinds of studies therefore do not necessarily generalize well to open-source development. Furthermore, most existing studies measure the impact of code coverage on the development or testing processes, but not the impact on the developers themselves.

In this thesis, we aim to address both these concerns by conducting several consecutive studies, starting off with empirical research based on a large set of existing coverage data, and ending with a survey disseminated among open-source software developers.

To better understand the working relationship of software engineers and their code coverage tools, we set out to investigate the following research questions:

RQ1. Is there a relationship between code coverage and the time to merge a pull request?

RQ2. To what extent do comments and status checks lead to improved coverage?

RQ3. Why are coverage checks ignored?

By doing so, we aim to contribute to a better understanding of the challenges and best practices in using code coverage on open-source development platforms.

Our empirical research is based on a large-scale data set sourced from a wide variety of projects independent of programming languages, frameworks, and project types. The survey is also conducted among a large and highly varied sample of open-source developers.

The data for our research as well as the list of potential survey candidates, are a combination of public data from GitHub, and data that was made available to us by Codecov, a popular code coverage tool. Overall, 287 projects were initially queried, and 278 developers completed our survey.

The structure of this thesis is as follows. First, we discuss the related works and relevant background information in Chapter 2. Then, we discuss each of our research questions individually in Chapters 3, 4 and 5, respectively. For each research question we conduct a separate study, and we report on our methodology and results in their chapters.

Finally, we will discuss our collective findings in Chapter 6, and a conclusion is given in Chapter 7.

Chapter 2

Background

In this thesis, we explore the use of code coverage tools on open-source development platforms and seek to answer our three research questions. However, before we start, we need to address the fact that not every reader is aware and up-to-date on these topics.

Therefore, in this chapter we will present the background information that is necessary to understand the research questions addressed in this thesis. We will start by explaining the pull-based development model, which is frequently used on open-source development platforms, such as GitHub. Then we move on to introducing the concept of code coverage, which is a metric for measuring the lines of source code in an application that are executed by automated tests. Third, we will give an example of a code coverage tool, namely Codecov. And finally, there will be a discussion of the related work in this field.

2.1 Background information

2.1.1 Pull-Based Development

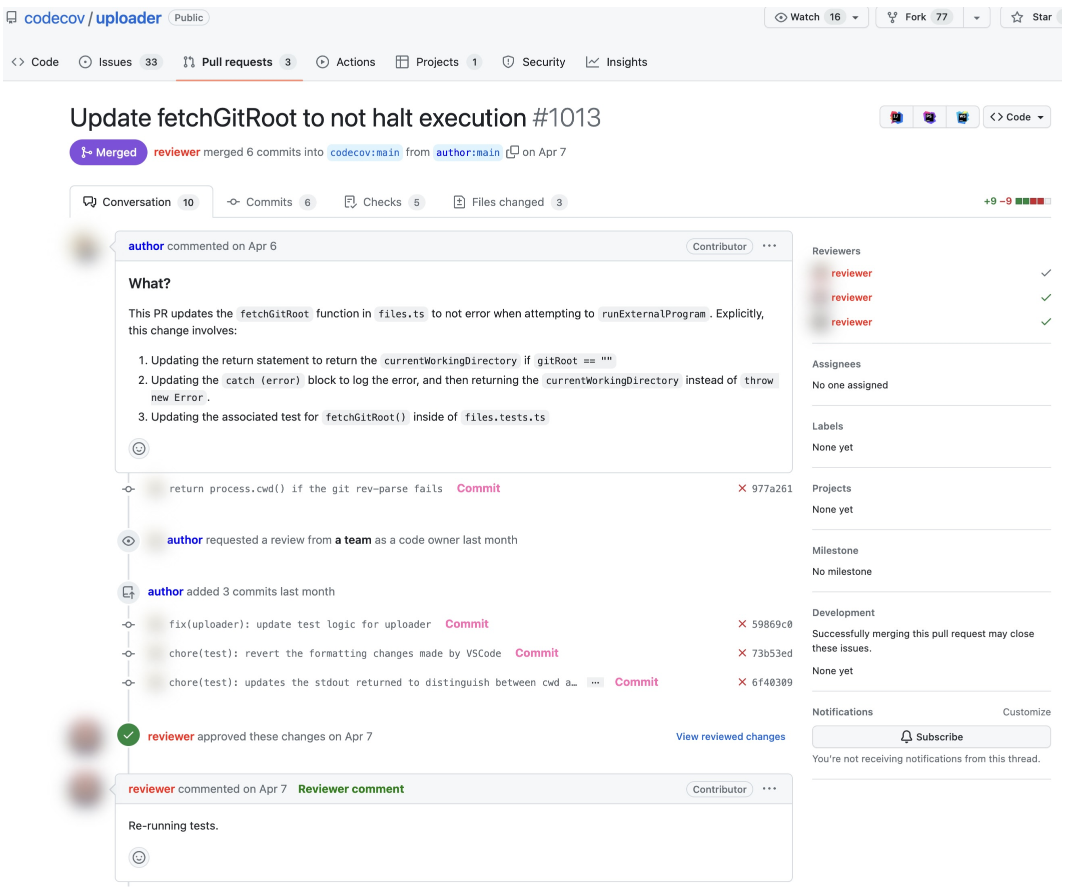
Code is often written by more than one person. For multiple people to collaborate on code, there needs to be some kind of method to keep track of, and combine, different versions of the code. In open-source development, platforms such as GitHub[1] exist to facilitate this collaborative software development. GitHub implements what is known as the pull-based development model [6]. This practice uses essentially two roles: the contributor and the integrator[2]. Nonetheless, it is possible to take up both roles, depending on the project or situation.

The contributor is a person who creates a new feature or bug fix and wants to have it included in the code. They do this by creating a 'fork' or 'branch' of the main code repository, making their changes, and submitting a 'pull request' to the integrator. A pull request consists of one or more 'commits', which are essentially records of the changes made to the code. Each commit can be seen as its own version of the codebase. Once the pull request is opened, the integrator is responsible for reviewing the proposed changes,

[1]But also other platforms such as GitLab, BitBucket, Azure DevOps, etc.

[2]or maintainer, code owner, etc. In our research, all these names are used interchangeably for this role

ensuring they meet the project's standards and merging them into the main codebase. An example of a pull request on GitHub is given in Figure 2.1. In this screenshot, relevant details are marked with coloured text, and personal details have been blurred.



Update fetchGitRoot to not halt execution #1013

reviewer merged 6 commits into `codecov:main` from `author:main` on Apr 7

author commented on Apr 6 — Contributor

What?

This PR updates the `fetchGitRoot` function in `files.ts` to not error when attempting to `runExternalProgram`. Explicitly, this change involves:

1. Updating the return statement to return the `currentWorkingDirectory` if `gitRoot == ""`
2. Updating the `catch (error)` block to log the error, and then returning the `currentWorkingDirectory` instead of `throw new Error`.
3. Updating the associated test for `fetchGitRoot()` inside of `files.tests.ts`

`return process.cwd() if the git rev-parse fails` Commit ✕ 977a261

author requested a review from **a team** as a code owner last month

author added 3 commits last month

`fix(uploader): update test logic for uploader` Commit ✕ 59869c0

`chore(test): revert the formatting changes made by VSCode` Commit ✕ 73b53ed

`chore(test): updates the stdout returned to distinguish between cwd a…` … Commit ✕ 6f40309

reviewer approved these changes on Apr 7 — View reviewed changes

reviewer commented on Apr 7 **Reviewer comment** — Contributor

Re-running tests.

Reviewers: reviewer ✓, reviewer ✓, reviewer ✓

Assignees: No one assigned

Labels: None yet

Projects: None yet

Milestone: No milestone

Development: Successfully merging this pull request may close these issues. None yet

Notifications: You're not receiving notifications from this thread.

Figure 2.1: Example of a pull request

To assess whether the standards are met, the integrator can use a variety of tools. For example: reading the code, manual and/or automatic testing, static analysis tools, code quality tools, code readability tools and code coverage. This is a process that we refer to as 'code review'.

2.1.2 What is code coverage?

Code coverage is a metric used to measure the amount of code that is executed during testing. It is used to assess the quality of a test suite [31], and to identify parts of the code that are not sufficiently tested. A commonly held belief among development teams is that, simply put, the more code that is covered by tests, the more likely it is that the code is correct and free of bugs. Although as we will see later, this belief does not always hold up.

Code coverage is usually expressed as a percentage, and there are several types of coverage metrics, such as line or statement coverage, branch coverage, and path coverage. Statement coverage measures the percentage of statements that are executed during testing. Branch coverage measures the percentage of branches in the code that are executed. Path coverage measures the percentage of all possible execution paths through the code that are executed.

Code coverage can be measured by running the code using a code coverage tool. Typically, this is done while running tests, and keeping track of which parts of the code are executed, and which are not. The tools can generate a full report, which provides detailed information on exactly which lines or branches are covered by tests. Furthermore, these tools provide a summary for the entire codebase, or individual files or modules, which consists of a coverage percentage, and absolute numbers of covered ('hits') and uncovered ('misses') code. It is these numbers that we use in our upcoming research.

Code coverage can be calculated locally on a developer's computer, However, running all the tests, as well as calculating coverage, can be time-consuming and/or computationally expensive. As such, tools have been developed to perform these tasks in the cloud instead. This is called Continuous Integration (CI)[30]. Besides the saved resources, there are a number of other upsides to this method. For example, developers do not need to install or set up these tools on their own machines, which saves them time and effort. Furthermore, the results are collected online, which means they are available to everyone working on the project, and easy to keep track of over a longer period.

When a contributor opens a pull request to a project that uses CI, this serves as a trigger to run the CI pipeline, which executes the tests and the code coverage tool. Once the tests are finished and all the results are collected, they are reported to the integrator, who will use them to evaluate whether the pull request should be accepted.

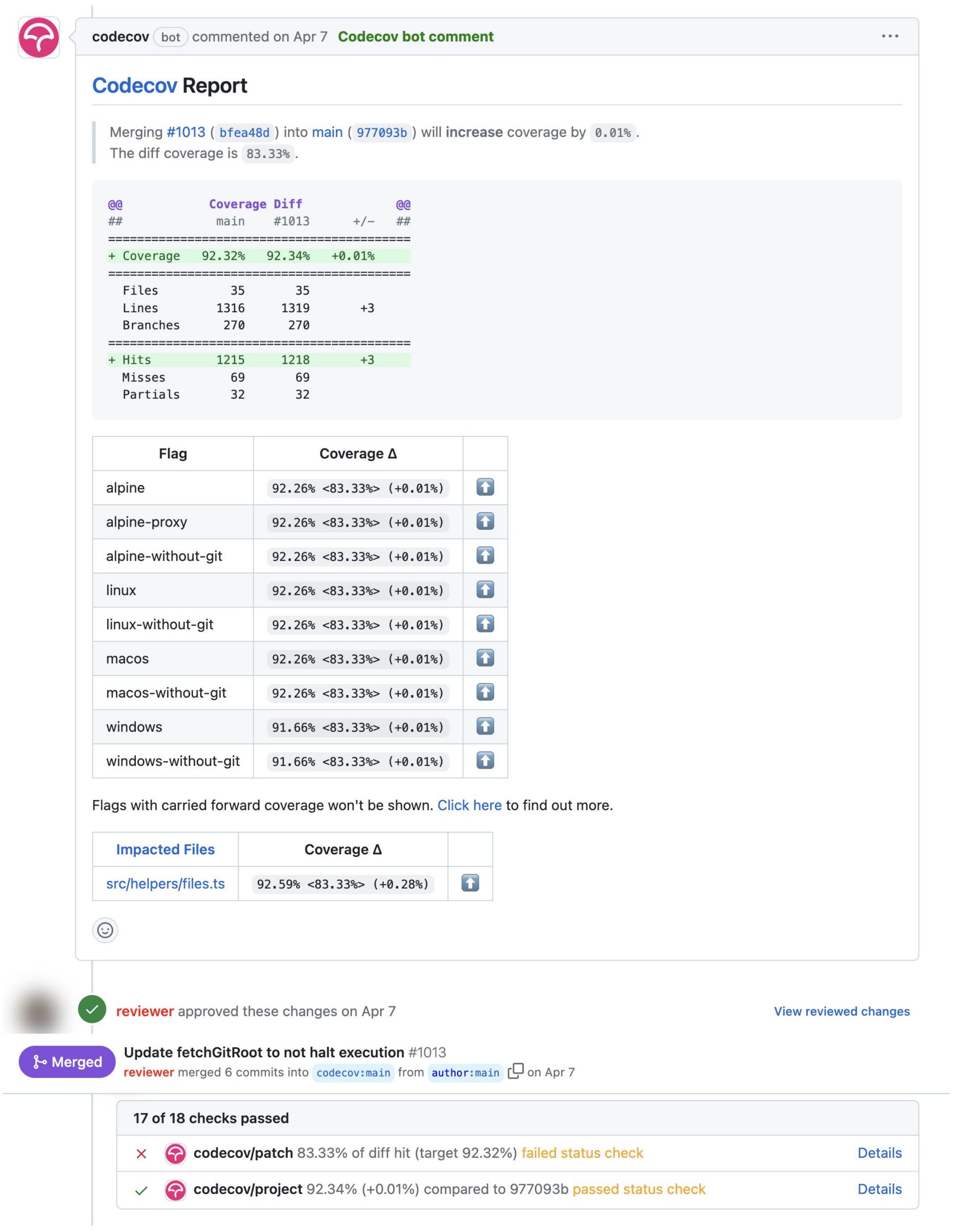
2.1.3 What is Codecov?

Codecov is a code coverage tool that we will use as an example throughout our research. It integrates with many different open-source development platforms and supports a large number of programming languages.

Codecov 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 2.2. Furthermore, it stores the reports it generates, so project maintainers can track the coverage levels over time. Additionally, it not only calculates the coverage levels for a project, but also for individual pull requests, which they call 'patch coverage'. This only measures coverage for the lines that were actually changed in the pull request[3].

Of course, it is not necessary to use a third party to collect and provide coverage information, but using a code coverage tool can make it significantly easier, since they are easy

[3]`https://docs.codecov.com/docs`



codecov bot commented on Apr 7 **Codecov bot comment** ···

Codecov Report

Merging #1013 (`bfea48d`) into main (`977093b`) will **increase** coverage by `0.01%`.
The diff coverage is `83.33%`.

```
@@            Coverage Diff            @@
##              main    #1013    +/-   ##
==========================================
+ Coverage   92.32%   92.34%   +0.01%     
==========================================
  Files          35       35              
  Lines        1316     1319       +3     
  Branches      270      270              
==========================================
+ Hits         1215     1218       +3     
  Misses         69       69              
  Partials       32       32              
```

Flag	Coverage Δ	
alpine	`92.26% <83.33%> (+0.01%)`	⬆
alpine-proxy	`92.26% <83.33%> (+0.01%)`	⬆
alpine-without-git	`92.26% <83.33%> (+0.01%)`	⬆
linux	`92.26% <83.33%> (+0.01%)`	⬆
linux-without-git	`92.26% <83.33%> (+0.01%)`	⬆
macos	`92.26% <83.33%> (+0.01%)`	⬆
macos-without-git	`92.26% <83.33%> (+0.01%)`	⬆
windows	`91.66% <83.33%> (+0.01%)`	⬆
windows-without-git	`91.66% <83.33%> (+0.01%)`	⬆

Flags with carried forward coverage won't be shown. Click here to find out more.

Impacted Files	Coverage Δ	
src/helpers/files.ts	`92.59% <83.33%> (+0.28%)`	⬆

reviewer approved these changes on Apr 7 View reviewed changes

Merged **Update fetchGitRoot to not halt execution** #1013
reviewer merged 6 commits into `codecov:main` from `author:main` on Apr 7

17 of 18 checks passed

✗	**codecov/patch** 83.33% of diff hit (target 92.32%) failed status check	Details
✓	**codecov/project** 92.34% (+0.01%) compared to 977093b passed status check	Details

Figure 2.2: Example of a Codecov comment and status checks

to set up and configure. 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. Therefore, we mainly focus on Codecov in our research. Nevertheless, there exist alternative tools, such as Coveralls or SonarQube.

2.2 Related work

Previous work on code coverage can largely be categorized into three groups. Namely, the effects of code coverage on code review, the effects on software testing, and the effects on code quality. In this section we will highlight these categories, and the related works for each of them.

2.2.1 Code review practices and tools

A field of study which is closely related to our research is that of code review, and particularly the use of tools or bots that assist with code review. This section explores the related work in this field. The studies discussed provide insights into the use of tools that assist with code review and quality control, and the nature of open-source software development. These studies deal with how contributors and maintainers approach pull requests, and which methods they use to determine whether they should be accepted.

For example, a study by Gousios et al. aims to predict whether a PR will be merged using machine learning. Projects were selected based on specific features, and their findings reveal that most pull requests affect only a few lines of code, and 60% of them are decided upon (e.g., merged or rejected) within a day. The decision to merge depends mainly on whether the PR affects code that was modified recently already, and the time to merge depends on code coverage and developer experience/track record. Furthermore, only 13% of rejected PRs are rejected for technical reasons [6].

In a study conducted at Microsoft, the authors created a model to predict the processing time of a pull request. The subsequent results of the model were then used to provide feedback to developers. As a result, pull requests were merged much more quickly, saving significant developer time. [17].

In another large-scale study, Yu et al. investigate the effects of adopting Continuous Integration (CI) on the quality of the developed software. The study's authors collected a large number of metrics and deployed various regression models on them to answer their research questions. These metrics are of various categories, namely: Product, Process, Churn, Social and CI metrics. Their findings reveal that only a few files typically cause CI to fail and therefore should be under more scrutiny. Additionally, PRs that at one point failed the CI, and were subsequently fixed and merged, have a higher chance of introducing bugs regardless. However, using CI alone does not necessarily indicate high-quality code [30]. The same can typically be said for other metrics, including code coverage. These studies served as inspiration for Chapter 3, in which we also deploy linear regression to model pull request completion times, but using code coverage metrics.

Another study, conducted by Mcintosh et al., examines the impact of code review on software quality. Authors collected a large amount of data on code changes and their reviews, and found that code review participation is significantly linked to post-release defects, meaning that review participation and discussion is encouraged to lower the chances of a defect. There is also a link between reviewer expertise and post-release defects, meaning that experts should be included in the review process[18].

Finally, another study by Wessel et al. compares activity indicators from before and after the adoption of certain code review bots. The study found that the number of merged pull requests increased, and the number of non-merged pull requests decreased monthly after adopting these bots. Furthermore, there was less communication on merged pull requests, and less time taken to reject pull requests [28]. Their study included multiple code review bots, including Codecov, but also other code coverage tools. In a survey conducted by the same authors, they found that developers perceived some other positive effects as well, such as less time to close a PR, and a reduction of workload. But also, negative effects such as noise and scaring away new contributors [29]. Furthermore, in a survey conducted at Google, key results were that developers feel positive towards code coverage, and will opt to use it, even if that is optional [12]. Because of these surveys, we were inspired to conduct our own survey in Chapter 5. However, in our survey, we do not focus on a single company, but rather on open-source developers on GitHub.

Overall, these studies suggest that code review is a critical part of software development, and that the use of tools can significantly improve its efficiency and effectiveness. The findings also highlight the importance of review participation and discussion, reviewer expertise, and the need for ongoing improvement of code review processes. They highlight that there are multiple factors to how quickly a pull request is processed. In our research we aim to further explore the specific impact of code coverage.

2.2.2 Software testing

In the field of software testing, several studies have been conducted to understand the factors that influence the effectiveness of test suites. In this section we will discuss some of these studies, and how they include code coverage in their methodology.

Firstly, Namin and Andrews explored the separate contributions of test suite size and code coverage to test suite effectiveness. The study found that both size and coverage independently influence effectiveness in a nonlinear relationship. This indicates that a combination of size and coverage is better at predicting effectiveness than either of them alone [20]. In another study, Mondal et al. explored the goal of selecting tests from a larger test suite to maximize both code coverage and test diversity while minimizing test execution time. The study found that optimizing using all three variables was significantly more effective than optimizing two variables separately [19].

An empirical study was conducted by Kochhar et al. to examine the relationship between different project metrics and the code coverage achieved in open-source projects. The study found a negative correlation between project size and coverage, as well as a negative correlation between cyclomatic complexity and coverage. However, on a file-level, these correlations are positive, showing that developers care about ensuring that large and/or complex files are tested [14].

This is backed up by a case study from Kim that was conducted on code coverage in a large piece of software. Instrumenting all the code to report code coverage takes too many resources, so the study aimed to use the available code coverage more efficiently. The study found that defects follow a Pareto-like distribution, and therefore, a more efficient approach to code coverage analysis was developed. The approach focused on error-prone modules and used coarse coverage approaches for error-free modules. The study also found a strong correlation between block and branch coverage and between complexity and size. The results showed that coarse coverage methods are a cost-efficient replacement for detailed methods [13].

Furthermore, in another empirical study by Derezińska, computer science students examined 50 C/C++ programs to find useful information regarding the coverage of these programs. The study found that basic tests based on program requirements achieved a high amount of code coverage on average. Additional tests were written to increase the code coverage further, and the test suites were optimized by removing tests that did not increase the coverage. The study concluded that additional tests for coverage were not difficult to achieve, and that when test suites were optimized, many of them included both the additional tests and original tests. Lastly, the uncovered code was examined and found to be mainly related to error handling, inaccessible code, or too costly to cover [5]. Something we explore further in Chapters 4 and 5.

Finally, in a past study by Schwartz et al., the authors use a number of projects with high code coverage levels and introduce faults into them. They found that the existing tests were only able to detect some of the faults, while others went undetected [23]. Therefore, the authors concluded that high code coverage alone is not necessarily a good indication of the quality of a test suite.

A similar conclusion was drawn in a study by Inozemtseva and Holmes. In this study, the authors used large Java projects to determine the relationship between code coverage and the tests' fault detection effectiveness, and found that there is a low correlation between them, and varies between different software. [11] This goes to show that, while low coverage is useful for determining parts of the code that need extra testing, high coverage does not mean everything is properly tested. For this reason, it is important to not only write, but also review test code, to ensure it is of high quality. Unfortunately, Spadini et al. concluded in their study that test code is reviewed significantly less than production code [24].

From these papers it shows that code coverage and software testing are two closely related topics. And while coverage is a good starting point for writing tests, it does not imply the tests will be of high quality. Which is a sentiment that is shared among developers, as we have concluded in Chapter 5.

2.2.3 Code coverage

In the previous subsection we mainly discussed the relationship between code coverage and the testing effectiveness. In this subsection we will discuss different studies relating to the direct relationship between code coverage and code quality.

In a past study, Hutchins et al. aimed to investigate the value of setting up code coverage monitoring to increase coverage and detect more faulty behaviour in code [10]. The experiment concluded that even 100% coverage does not guarantee fault detection, but low coverage levels are a good indicator of test inadequacy. This study, however, is nearly 30 years old, but arguably the results are still relevant today.

In a more recent study, Hemmati examined whether code coverage is an effective way of finding bugs. The study found that control-flow forms of code coverage were largely ineffective in finding bugs and recommended using a combination of different control- and data-flow coverage, as well as some tests based on the specification, and not coverage alone [8].

A similar conclusion was reached in another study. Here the researchers investigated the correlation between code coverage and the number of bugs found in a system once it is released. The study found insignificant or no correlation between the two factors [15].

Finally, in a large-scale study Hilton et al. analyze patch coverage for commits to investigate how code coverage evolves over time. Their key findings are that patch coverage is not correlated with project coverage, patches often affect coverage of code not in the patch, and patches which appear to not change the coverage can contain a large number of changed statements. Especially for very large projects, patch coverage is a more useful metric than project coverage, but it is not the sole metric you should focus on during a code review [9].

These papers present a somewhat negative view of code coverage. However, in Section 2.2.1, developers seem to be mainly positive about code coverage. This therefore serves as a great opportunity to explore both of these facets further in our own research, by conducting empirical research in Chapters 3 and 4, and by conducting a survey with developers in Chapter 5.

Chapter 3

RQ1: Is there a relationship between code coverage and the time to merge a pull request?

As previously mentioned, tools like Codecov generate coverage reports, that inform developers of the coverage levels of their projects or contributions. In this chapter we aim to determine in what way having these coverage reports affects how the PR is merged. For example, do pull requests with fully covered changes get merged quicker than others? And do pull requests which lower the overall coverage get merged slower? Such a relationship could motivate a developer to write better and more tests for their pull requests, and improve its coverage.

In our study, we therefore focus on establishing a relationship between the coverage levels reported on a pull request, and the time it takes to merge that pull request. We do this by combining the coverage data from open-source repositories on GitHub from Codecov's database, and public information about the repository gathered by GHTorrent [7], to build individual linear regression models for a selection of popular and/or big open-source projects. Using these models, we can evaluate the impact of different code coverage measures on the pull requests' merge time. We use coverage levels from before the pull request was merged, the coverage levels achieved in the pull request, the difference between these, and Codecov's own measure of ”patch coverage” as various coverage metrics in our models.

The full methodology of our study is described further down, in Section 3.1, while the results can be found in Section 3.2.

3.1 Methodology

In this section we explain how we performed this research. We will discuss how we collected and processed our data, used it to create our models, and how we then analysed those models.

Data collection We gathered data on public, merged pull requests on GitHub from Codecov's own database of coverage information and pull requests, and appended it with additional information from GitHub using the March 2021 database dump of the GHTorrent project [7]. We originally searched for PRs from 287 popular open-source projects on GitHub, which also use Codecov in their repositories. These open-source projects were provided by Codecov as a starting point for our research. The period during which the PRs were merged range from 2017 to 2021, and we included only projects with over 100 merged PRs. The full list of projects can be found in Appendix A.

The information we gathered from Codecov includes the coverage information from base and head of the merged PR. This information adheres to the same format as the coverage information returned by Codecov's API [1], and this API could therefore, in combination with GHTorrent, be used to replicate this experiment. The data from GitHub/GHTorrent includes some basic information of the PR and the repository at the time of opening the PR. The coverage information is provided in two ways. One is a number between 0 and 100, which represents the coverage percentage. The other consists of absolute values for the number of coverage hits, misses and partials[1].

The data was queried from a Google BigQuery Database and subsequently around 11GB of JSON files were downloaded to a machine for further processing. Ultimately only 123 repositories with over 100 PRs were found from the original list of 287.

Data processing After collecting the data, we processed it by calculating the various metrics shown in Table 3.1. We also removed duplicate entries which are a result from duplicate rows in the databases. The data was downloaded as nested JSON files, which were then processed into flat JSON objects. During this process, many of the metrics were calculated.

- Not only were the raw coverage metrics used, the difference between the coverage metrics of the head and base commit of the PR were also calculated and used as separate features.
- The difference between the time the PR was opened and the time it was merged was used to calculate the time to merge, given in hours.
- The authors of the commits in the PR were used to get the number of commits in the PR, as well as the unique authors in the PR.
- The authors of the comments left on the PR were used to get the number of comments, as well as the number of reviewers (excluding the authors of the PR, who might have left comments as well).
- The authors of the previous commits in the repository were used to calculate the number of previous commits, as well as the number of past contributors in the repository.
- Whether the PR originated from a fork or not was used as-is.

After this step, one large flat JSON file remained, which was converted into a CSV file using `json2csv`. Subsequently, duplicate rows, which were the result of duplicate rows in the original databases, were removed.

[1]branches for which only some parts are covered

Metric	**Gathered from**	**Name in model**
Independent variables		
Time to merge PR, given in hours	GHTorrent	timeToMerge
Variables we are interested in		
Project coverage before PR	Codecov	coverage_before_ {hits,misses,partials,coverage}
Project coverage achieved in PR		coverage_after_*
Patch coverage achieved in PR		patch_coverage_*
ΔCoverage before and after the PR		diff_coverage_*
Control variables		
#Commits in PR	GHTorrent	commitsInPr
#Authors in PR		authorsInPr
#Commits in project before PR		commitsInProject
#Contributors before PR		contributorsInProject
#Comments left on PR		comments
#Reviewers		reviewers
PR opened from a fork or not		intraBranch
#Files included in the patch coverage of the PR	Codecov	patch_coverage_files
#Lines of code included in the patch coverage of the PR		patch_coverage_lines
#Files included in the coverage report before PR		coverage_before_files
#Files included in the coverage report after PR		coverage_after_files
ΔFiles in the before and after reports		diff_coverage_files
#Lines of code in the coverage report before PR		coverage_before_lines
#Lines of code in the coverage report after PR		coverage_after_lines
ΔLines of code in the before and after reports		diff_coverage_lines

Table 3.1: Metrics used in the linear regressions to answer RQ1.

Modelling In this step, we group all the gathered PRs by the repository they belong to. Using the `statsmodels` module for Python, we create an Ordinary Least Squares (OLS) model for each of the projects. These models include all the metrics given in table 3.1, as well as a constant to serve as an intercept. For our models, we've set the dependent variable to the time to merge the pull request. The full results from these models can be found on GitHub: `https://doi.org/10.5281/zenodo.8044949`.

The point of doing this, is to model our dependent variable as the outcome of a function with the independent variables as parameters. In this case it would be a linear function, where the outcome is the sum of the independent variables, and each independent variable has a weight associated with it. By using the OLS algorithm, we can adjust those weights to get the function that fits the weights the best.

Analysis Here we rank the models from the previous step based on their R-squared score as reported by `statsmodels`. In this case, this score indicates how well the model explains the given data for a project. This suits our use-case, since we are not looking to predict the

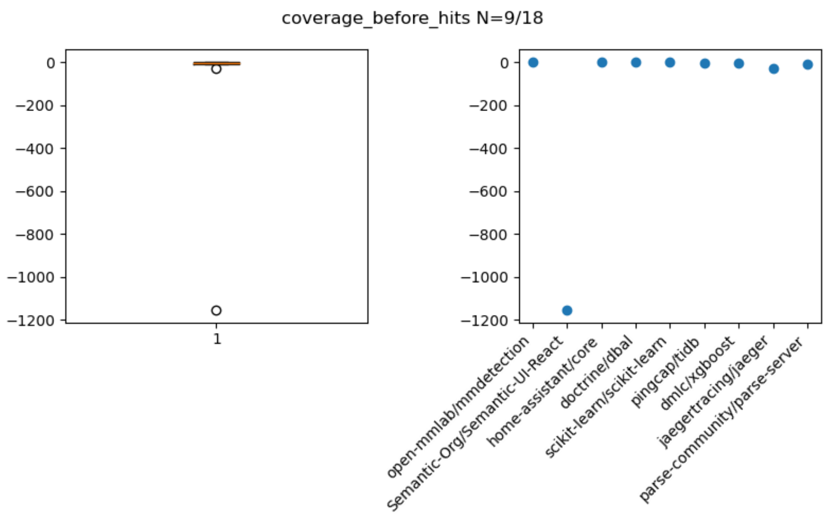


Figure 3.1: Plots for the coefficients for coverage_before_hits

time it takes to merge a PR, but simply to explain whether there is a significant relationship between this variable, and the coverage variables.

We filter out the models with an R-squared less than 0.7, since we believe them to not be performing well enough. A list of the projects belonging to these models can be found in A. For the remaining 18 models, we collect each metric for which `statsmodels` indicates it has a p-value of less than 0.05. These metrics are then considered to have a significant relationship with the dependent variable (the time to merge) in that particular model.

The results of this analysis can be found in the next section.

3.2 Results

For each metric, we collect the corresponding coefficients from only the models where that metric is considered significant (i.e., its p-value is less than 0.05). A quick summary of these coefficients per metric is given in Table 3.2. From these results, we see the following interesting things:

Firstly, out of all the coverage metrics, **the project's coverage before the PR appears to be the most often significant of all the coverage metrics.** In 9 out of the 18 projects (50%), this variable has a significant relationship with the time to merge a PR. Interestingly, this metric says something about the state of the repository before the merge of the PR. This could imply that the coverage levels at the time a PR was opened are important more often than the coverage levels achieved by the PR itself. Especially since coverage levels after the PR are significant slightly less often. Namely, only in 7 out of 18 projects. Admittedly this is not a huge difference, but still noteworthy enough to deserve a mention.

	#Times	**coefficient**			
metric	**significant**	**mean**	**std**	**min**	**max**
commitsInProject	13	-0,55	11,80	-32,90	22,68
contributorsInProject	12	-46,20	213,35	-257,81	566,91
comments	10	29,13	32,38	-16,49	74,99
const	10	-2027653,74	6043836,16	-18798268,25	2351660,60
coverage_before_lines	9	131,28	381,17	-1,53	1147,50
coverage_before_files	9	10,94	99,06	-102,62	235,70
coverage_before_hits	9	-131,72	383,47	-1154,03	2,59
coverage_after_files	8	105,10	225,60	-59,63	552,50
commitsInPr	7	27,47	34,07	4,64	102,28
coverage_after_hits	7	-20,64	49,22	-131,41	4,00
coverage_after_lines	7	16,00	37,13	-1,66	99,36
coverage_before_missed	7	334,47	867,48	-2,86	2301,53
coverage_before_coverage	7	14783,16	35481,25	-259,55	95096,54
diff_coverage_coverage	6	-1572,07	5105,39	-11748,33	2541,43
coverage_before_partials	6	2,35	5,49	-4,41	9,69
reviewers	6	114,19	164,04	-161,08	291,99
coverage_after_missed	6	16,72	55,66	-41,41	120,71
coverage_after_partials	5	22,56	48,92	0,00	110,06
coverage_after_coverage	5	4617,87	11772,01	-6037,16	24453,06
authorsInPr	5	138,66	431,76	-314,93	640,26
diff_coverage_files	4	98,51	268,25	-86,27	494,97
diff_coverage_hits	4	8,98	25,47	-8,08	46,83
diff_coverage_missed	4	-14,53	38,43	-71,25	14,08
patch_coverage_files	4	16,00	27,41	-16,43	48,71
patch_coverage_partials	4	-2,03	4,06	-8,11	0,00
diff_coverage_lines	3	-12,66	27,15	-43,81	6,00
intraBranch	3	2458,59	4500,31	-563,99	7630,59
patch_coverage_coverage	2	-5,64	3,07	-7,81	-3,47
patch_coverage_lines	2	0,02	3,81	-2,68	2,71
patch_coverage_misses	2	4,02	2,08	2,55	5,49
patch_coverage_hits	1	-2,77		-2,77	-2,77

Table 3.2: A table summarizing the results

From the data we can tell that the coefficient for this metric is typically negative or very slightly positive, with a maximum of 2.59 according to Table 3.2. One theory for why most coefficients are negative is that hits are a positive coverage metric. The more hits you have, the better your code is tested. This in turn could lower the time to review (and therefore merge) a pull request.

When we look at the coefficients for this metric (coverage_before_hits), which are given in Figure 3.1, we see one outlier, with a very negative value. This outlier belongs to a single project that already has a very high level of coverage (roughly 99%) in all of its data points, namely `Semantic-Org/Semantic-UI-React`. This is on average the highest level of coverage we were able to find in our database. While the number of hits does vary a bit, the actual coverage percentage changes almost nothing for each PR. This could cause the number of hits to weigh more in the model than, for example, the percentage-based coverage metrics.

Secondly, when looking at both the coverage_before (Figure 3.1) and coverage_after metrics (Figure 3.2), **hits are significant more often than misses.** However, the difference is not that big: 9/18 vs 7/18 for before [the pull request], and 7/18 vs 6/18 for after. It should be noted that not every project where misses are significant, has hits significant as well and vice versa. In general, we also find that **hits have a more negative relationship/coefficient with the time to merge, and misses a more positive one.**

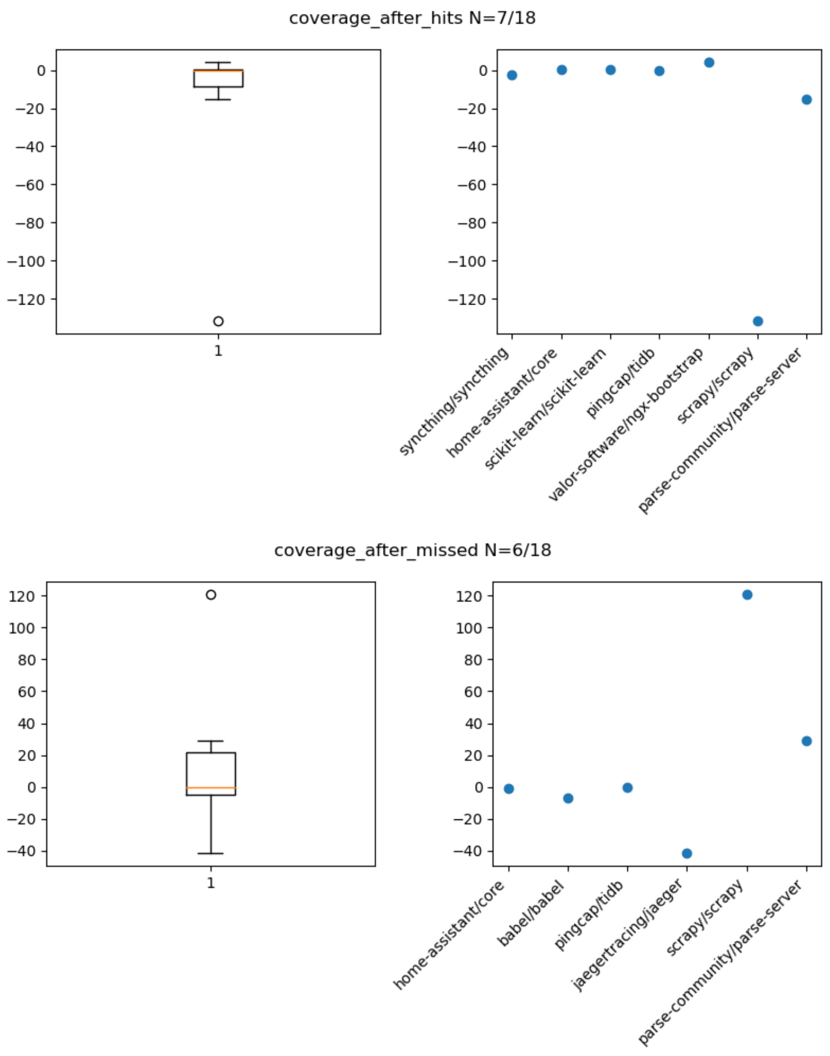


Figure 3.2: The coefficients for hits and misses of coverage_after

One interesting project to look at though, when we look at Figure 3.2, is `scrapy/scrapy`. Here we see that for the hits, there is a very strong negative relationship, while for the misses, there is a very strong positive relationship. There seems to be some kind of balance between the weight of hits and misses. The more hits implies a lower time to merge, while the more misses implies a higher time to merge. This is, of course, not a groundbreaking conclusion, considering that the existence of more misses automatically implies a lack of hits and vice versa, and the fact that hits are favoured over misses when developing and reviewing code. However, it is interesting to see this principle reflected in the data so strongly and so clearly.

The third interesting thing we find, is that **patch coverage metrics are the least significant coverage metrics**. These metrics represent how well the code in the diff of the PR is covered. Just like the other coverage levels, Codecov provides a report of the patch coverage on the PR itself, in the form of a comment. However, according to our results, these metrics rarely have some kind of significant relationship with the time to merge a PR. Looking at Table 3.2, we see that the coverage percentage, number of hits and number of misses are significant only 2, 2, and 1 time, respectively.

One final thing to note here is that even here it holds that for the projects where these metrics *are* significant, misses increase the time to merge, while hits decrease it. This lines up with the same conclusion made in the previous paragraph.

Furthermore, something that may appear odd at first, when looking at Table 3.2, is the high variability of the ‘const’ metric, also known as the intercept value. This is the value the time to merge would take if all other metrics would be 0. While that is a completely unrealistic case, since a PR or project will always have commits or contributors, etc, it is worthwhile to theorise a bit more on why this metric has such extreme values and high variance.

One theory is that certain metrics are simply always very large. Especially in these particularly popular projects. These projects can have hundreds if not thousands commits and/or contributors. Furthermore, hits or misses can occur in almost every line of code. When you think of a model as a formula, with our metrics on one side and time to merge on the other, it could mean that a high intercept value is necessary to offset the sum of all the other metrics, to get a reasonable result.

Of course, each project is also inherently different. They are all set up differently and are managed differently as well. There are a lot of more human factors, that we, in this study, were not able to capture in metrics. As such, the intercept variable could represent all these ‘hidden’ metrics. For example, it is possible for a team to only review PRs on a particular moment every week. In this made-up scenario, PRs will take longer to merge, not due to any of the metrics we collected, but due to the structure set up by the developers.

Lastly, the final thing we see is that **some of the control variables, and not the coverage variables, are the most significant overall.** Specifically, `commitsInProject`, `contributorsInProject` and `comments`. These metrics have a relationship that can be considered significant with the time to merge a PR 13, 12 and 10 times, respectively. Our theory is that if a pull request has a lot of comments, there is a high chance that it was open for a relatively long time. Similarly, a bigger project with a large number of maintainers who merge pull requests, can also have a quicker time to merge, since more maintainers are available. Nevertheless, we did not perform additional research to confirm these theories.

This is also shown in Figure 3.3. With a single exception, comments only see a positive relationship with the time to merge. However, the other two metrics see more of a mix between positive and negative relationships. Looking at the raw data (`https://doi.org/10.5281/zenodo.8044949`) of the projects these coefficients belong to, there is no clear indication of whether a high number of commits or contributors corresponds to either a positive or negative relationship. There seems to be some further hidden complexity that cannot be captured by our models.

Another interesting thing to note is that the coefficients for the number of contributors are more varied than those for the number of commits or the comments. This can be seen in table 3.2 and Figure 3.3.

Additionally, we did consider the possibility that a large number of commits and comments can have an impact on the coverage level of the PR, as well as the time to merge. We look further into this first relationship in our next chapter.

3.3 Threats to Validity

We have come up with a number of threats to the validity of our results.

Firstly, we made a clear effort to only look at a small list of large projects. Not only were we constrained to using projects that use Codecov specifically, but we also believed that for our models to be meaningful, we needed a lot of data per project. We therefore only looked at large, popular projects, because we believed they would have the most data. Also, we only included projects that use Codecov, and not any other coverage tools. Consequently, our research here might not be as generalisable to any open-source project. However, the projects we chose did span multiple fields and programming languages, so we still believe that it is diverse enough. Similarly, in hindsight we could have included more metrics. One that immediately comes to mind is the number of pull requests the project had, as well as how many of those were merged, closed or remained open.

Secondly, we combined data from Codecov with data from GHTorrent. This means we used only a subset of GitHub's data, while we could have used GitHub's API instead, and be closer to the source. The GHTorrent data was already somewhat outdated at the time of the research, spanning up to March 2021, but not going beyond that. If we had found a way to use the GitHub API instead, we might have had more data to work with. Nevertheless, the use of GHTorrent hugely simplified our process of linking Codecov data to GitHub data, since we could both query them together using Google BigQuery. Furthermore, we tried to mitigate any problems with extra efforts on removing duplicates from our final dataset.

Lastly, we used a rather simple model to look at the data. We tried a variety of different models to fit the data, but we found that linear regression simply gave us the best results. You can argue whether that makes it the best model for our purposes, or whether we just picked the one that gave us the nicest looking results. However, we believe the linear regression model we used is suitable, because we are only aiming to explain a certain behaviour and look for possible relations. We are not interested in predicting anything, so therefore a

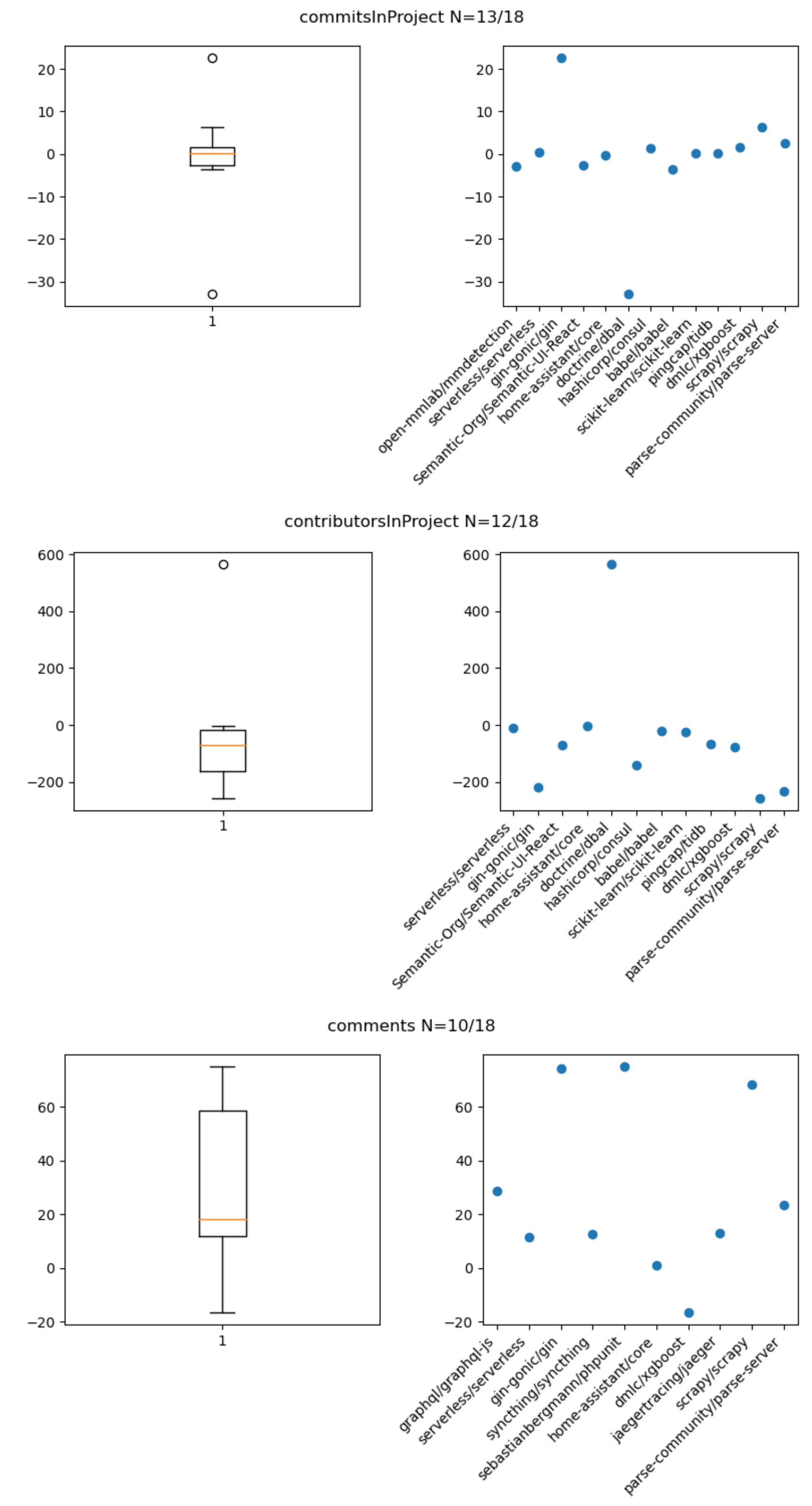


Figure 3.3: The coefficients corresponding with various control variables

perfect fit is not necessary. In any case, the results it gave us led us to come up with many interesting theories regarding the relation between the time to merge and code coverage.

3.4 Conclusion

In this chapter we investigated the relationship between code coverage of a pull request, and the time it takes to get merged. To collect our data, we combined data from Codecov's own database, and GitHub (using GHTorrent). We focused on larger, high-ranking projects on GitHub, with over 100 merged PRs only. The metrics we collected included coverage levels at the base and head of the PRs, as well as how many commits, files, lines, comments, reviewers and contributors the PR had.

For each individual project, we created a linear regression model, where the time to merge the PR was the dependent variable, and the other collected metrics the independent variables. We then filtered these models on how well they performed, and from the remaining ones we collected all the independent metrics that were considered to have a significant relationship with the time to merge.

We then grouped these significant metrics into metrics related to code coverage, and other metrics, to answer the question whether there is a relationship between code coverage and the time to merge of a pull request. In our results we find that in 50% of the models, there is a significant relationship between the project's coverage levels and the time to merge a PR. Furthermore, we also find that coverage hits are considered to have a more negative relationship with the time to merge, while misses have a positive relationship. Meaning that more hits lead to a faster time to merge, and more misses lead to a slower time to merge.

We also find that patch coverage metrics are the metrics that are the least often significant of all the metrics we collected. This implies that project coverage plays a bigger role in determining the time to merge in our models.

In conclusion, in some projects there exists a significant relationship between the time to merge a pull request, and code coverage.

Chapter 4

RQ2: To what extent do comments and status checks lead to improved coverage?

As part of their services Codecov provides developers on GitHub (and other platforms) with updates on their coverage. These updates are typically in the form of a comment left on a pull request, or a status check submitted for individual commits. They also provide other forms of notifications, for example through a Slack bot, which notifies you directly if there are any problems.

In our previous research we found that the number of comments and commits are the metrics that we found to have a significant relationship with the time to merge a pull request the most often. Therefore, in this chapter, we will now focus on the comments and commit messages of PRs, to find out to what extent they are about the coverage levels and/or tests of the PR.

We explore this by looking for commits for which the Codecov status check reported a failure, implying that the PR, at some point, did not adhere to the coverage targets set by the maintainers. We then look at the subsequent comments and commit messages to look for mentions of coverage and/or tests.

4.1 Methodology

Codecov keeps track of all the notifications it sends to its users in a database. Using this data, we find commits for which users were notified of a failing status. Here we only focus on notifications in the form of comments left on the PRs, and status checks left on commits belonging to PRs. We use the GitHub API to check for the failing statuses, as this information is not actually stored by Codecov[1].

We divide our commits in two groups. Each group consists of 200 commits, which were randomly selected from the earlier group of projects we used in chapter 3. The first group

[1]Codecov stores whether setting the status itself was a success or a failure, but not the actual state of the status

contains PRs for which the project coverage at the head of the PR would go down. The second group contains PRs for which the project coverage at the head of the PR would go up or stay the same. Both groups contain PRs with at least one commit which has a failing status due to Codecov **only**. This failure is triggered when a project does not adhere to the coverage target that is configured for the project.

4.1.1 Quantitative analysis

For both groups, we gather the comments and commit messages, and look for certain keywords such as:

- codecov
- coverage
- test
- testing
- tested
- tests
- retest

We then look at how many of these PRs mention coverage or tests in their discussions and commit messages, and how many of these PRs actually see their coverage improve. This is a statistical test that compares the odds of an event happening in one group, to the odds of the same event in another group. We compare this to the other PRs, which do not mention coverage, and run the **odds ratio test** to compare the chances of coverage improving when it is mentioned.

We run two tests, one for mentioning coverage or tests in the comments left on the PR, and one for mentioning coverage or tests in the commit messages for the PR. The results can be found in Section 4.2.

4.1.2 Qualitative analysis

We then manually analyze each PR, to see whether any kind of discussion or mention regarding the failing coverage check or missing tests has taken place, or if there were any commits specifically addressing coverage or adding new testcases. Furthermore, we look at the reports left by Codecov, to get a sense of how much these play a role in the discussions. We then categorize the PRs into the following buckets:

1. No kind of discussion has taken place.
2. A discussion/mention was there, but coverage did not get fixed, and the PR was merged anyway.

	Mentions of coverage or testing	
Coverage goes up	**True**	**False**
True	79	121
False	55	145

Odds ratio	1.7213
P-value	0.0113
Confidence interval	1.1305 to 2.6207

Table 4.1: Results for mentions of coverage in PR comments

	Mentions of coverage or testing	
Coverage goes up	**True**	**False**
True	73	127
False	52	148

Odds ratio	1.6360
P-value	0.0240
Confidence interval	1.0669 to 2.5085

Table 4.2: Results for mentions of coverage in PR commit messages

3. A discussion/mention was there, and coverage got fixed or the PR was closed if it was not.

4. Miscellaneous. A small category for PRs which had some unexpected behaviour. For example, the coverage check being displayed as passing during a manual review, despite the GitHub API listing it as a failure during our data collection.

In the results Section 4.2, we will report the distribution of the PRs among these categories. After this we will also try to dive into each category and see if there are some commonalities between the PRs within the category.

4.2 Results

4.2.1 Quantitative analysis

The results of the quantitative analysis can be found in Tables 4.1 and 4.2. These tables show the division of the PRs into four categories, based on the two axes: “coverage of the PR went up” and “coverage and/or tests were mentioned”. Furthermore, they show the results of the odds ratio test, as well as P-value (the probability that the data we observe is likely to occur, given that there is no relation at all (i.e., the lower the better).

From these results we can see that, according to our data, **the odds of coverage improving when either tests or coverage are mentioned in the discussion are greater than the odds of when these things are not discussed**. And this effect is considered statistically significant, as stated by the P-value being lower than 0.05. However, this does not mean that just mentioning coverage or testing in a discussion will automatically increase the coverage of a project, since we know that correlation does not imply causation.

A similar conclusion can be drawn regarding coverage improving when coverage or tests are mentioned in the commit messages. This might seem obvious: creating commits to address coverage or tests should lead to an increase in coverage. However, from Table 4.2

	Coverage goes up	
Category	**True**	**False**
No mention	133	168
Mention, no fix	7	7
Mention, fixed or closed	57	16
Miscellaneous	3	9

Table 4.3: Results of qualitative analysis

we can see that in our experiment we had 52 PRs where this was not the case. So maybe it is not so obvious after all. A possible explanation could be that commit authors mention that they cannot test or improve coverage in their commit messages. These messages would still be considered as mentioning coverage or tests, and would therefore be false positives in our data.

4.2.2 Qualitative analysis

In our qualitative analysis we looked at all 400 PRs manually, and assigned them to one of four categories mentioned in Section 4.1.2. Table 4.3 shows the distribution of PRs in our two groups among these categories.

Recall from Section 4.1 that all our PRs contain at least one commit for which Codecov has reported a failing coverage status check. One of the first interesting things we see in our table, is that in most PRs we have sampled, there is no mention of this failing check.

Another thing we see when looking at our table, is something that corroborates our findings in Section 4.2.1. Again, we see that mentioning coverage is more often associated with coverage going up. While there are 16 PRs where coverage is mentioned and the coverage does not go up, in our manual analysis we found that most of these (11/16) haven't been merged, but either closed or left open for the author to fix. The remaining 5 had some problems with not achieving 100% patch coverage, but the project coverage was unchanged.

4.3 Threats to Validity

One threat to validity we found is that we used a simple text search to look for mentions of testing or coverage. We looked for mentions of "coverage", "test", and a few other keywords. This means that it is possible that certain mentions of these keywords could be taken out of context, and do not actually refer to the current (state of the) pull request.

Furthermore, our list of keywords was quite short, to avoid very common words from being detected as a mention. In a way, this could also be a threat, because we might miss other relevant comments or mentions. Especially since our keywords are only in English. In the end, we opted to be less flexible in the words we accept as a mention, but have more confidence that the mentions are correct.

We tried to mitigate this problem by going over all the pull requests manually, as part of the qualitative analysis. However, this in itself is also a threat. Due to the nature of qualitative analysis, the results are based on very subjective groupings and categories. Only one person looked at all 400 pull requests. We tried to mitigate this problem, by having a supervisor from the TU Delft look at the first 40, to detect any bias, which was not detected.

Another threat is that we used the same list of projects we used in Chapter 3. Therefore, we have the same threat here as we have there, namely that our results might not be as generalisable. However, we still believe that our initial selection of projects is quite diverse in terms of fields, programming languages and structure.

Lastly, one thing we missed is sampling for PRs where the drop in coverage was more significant. We believe that a lot of the PRs we found had failing coverage checks due to tests or the code coverage tool being flaky. We tried to mitigate this problem using the qualitative analysis as well.

4.4 Conclusion

In this chapter, we investigated commits that failed the Codecov status check, and see if they got fixed. From the same list of popular projects from Chapter 3, we sampled pull requests that each had at least one commit for which Codecov reported a failing status check, using the GitHub API.

We then divided our samples into two groups. One where the coverage status check was fixed, and one where it was not. Afterwards, we randomly selected 200 PRs from both groups, in order to compare the two groups. Specifically, we focused on mentions of coverage and/or tests in the comments and commit messages of the PR, to see if there would be a difference in the frequency of these mentions between these groups.

We used a simple text search strategy and verified each mention by hand in a further qualitative study. We ran an odds ratio test between the two groups and found that the odds of coverage improving when either tests or coverage are mentioned are greater than the odds of when they are not. Furthermore, we found that in a large majority of the PRs in both groups, the failing coverage check is never mentioned, which is something we will discuss in the next chapter.

Chapter 5

RQ3: Why are coverage checks ignored?

During our research for Chapter 4, we found that mentioning tests and/or code coverage is related to the code coverage increasing. Furthermore, we found many instances where both contributors and maintainers opened or merged pull requests, where coverage checks were ignored.

To find out why that is the case, we conducted a survey with 278 participants, who were invited based on previous experience with both GitHub and Codecov, a popular code coverage tool.

This chapter will start off by explaining the goals of the survey, and how the survey was constructed in order to achieve those goals, in Section 5.1. Then, we will dive into the results of the survey in Section 5.2. Finally, a conclusion will take place in Section 5.4.

5.1 Methodology

5.1.1 Goals

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. We have chosen to conduct a survey to achieve these goals, because there exists a wide variety of reasons why pull requests with a failing coverage check could get merged, and because these reasons can be subjective, and based off of people's own opinions of code coverage and open-source software development as a whole.

Another thing that is important for us, is to see if there are any significant differences of opinion between developers who consider themselves 'maintainers', and those who more closely identify with 'contributors'. For this we have opted to use two different paths in the survey, based on the participant's answers.

5.1.2 Constructing the survey

Our survey was built using Qualtrics, with a licence provided by the TU Delft. It consists of a mix of demographic, quantitative (closed) and qualitative (open) questions. The entire survey can be found in Appendix B, and a summary list of questions can be found in Table 5.1.

The survey starts with an introductory message clearly stating the purpose of the survey, and who is conducting it. Furthermore, it provides clear information to the participants on how they were selected and additional contact information in the case of questions. Lastly, it informs participants that the survey is anonymous, that the results will possibly be published as part of academic research, and thanks them for their participation. We followed the guidelines set by TU Delft's Human Research Ethics Council and had our study proposal submitted to them for approval.

The demographic questions consist of open and closed questions regarding the participant's years of experience with (open-source) software development, and familiarity with code review and software testing processes. Furthermore, it includes a binary choice for users to assign themselves either a maintainer (or project manager), or a code contributor. Of course, there are situations where users might feel like they are a little bit of both, or neither. However, for our use-case, we asked them to select the option they most identified with. Using their answer to this question, we are able to give them a different set of questions in the survey. So, it is important that we make a binary distinction here.

For the main part of the survey, i.e., the qualitative and quantitative questions, we first explain our interpretation of code coverage tools which we will use during the survey, using screenshots of Codecov as an example. We figured that since it is the leading coverage solution [4], users will most likely recognize it.

The quantitative questions are closed questions that mainly consist of Likert scale questions. These include questions on how often they use code coverage tools, and how important they would consider them. Should users indicate that they do not use code coverage tools, the survey immediately ends with the option to provide additional feedback on why not. Otherwise, the survey continues with the main questions. These are questions about how often they ignore failing coverage checks, both from the perspective of a contributor, and a reviewer. Lastly, it also includes questions about how well code coverage tools incentivize users to keep up coverage and/or testing practices.

The qualitative questions are open questions about in what situations participants would ignore coverage checks, what their coverage goals/practices are, and what they like and dislike regarding coverage tools. There is also a question where participants can recount a memorable situation regarding the use of code coverage during the review process. Lastly, there is an open question for users to leave additional comments on the survey. We also provided participants with the option to leave their email address, in order to be contacted for potential future studies or interviews, yet no participants have been contacted.

Table 5.1: Summary of the survey questions

Number	Question	Type	Intended for
Demographic questions			
1	For how many years have you been developing software? You can consider all hobby, study and/or work experience.	Open	All
2	For how many years have you been active on open-source development platforms, such as GitHub, Gitlab, etc?	Open	All
3	How often do you contribute to an open source project?	Likert	All
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	All
5	Do you work in software development in a professional capacity? (Yes OR No)	Closed	All
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?	Likert	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)	Likert	All
Main questions			
9	How often do you use code coverage tools outside of GitHub? For example, on your own machine.	Likert	All
10	How often do you utilise the information from code coverage tools on GitHub?	Likert	All
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?[1]	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	All
13	Please give your opinions on the following statements:	Likert	
13a	Code coverage is a good metric to consider as part of overall code quality		All
13b	Code coverage tools on open- source platforms provide an incentive to improve coverage and/or write tests.		All
13c	I am more likely to approve a pull request that improves code coverage than ones that lower it.		Maintainers

[1] After this question, the survey ends

Number	Question	Type	Intended for
13d	If my pull request improves coverage, it is accepted more quickly, in my experience.		Contributors
14	How often do you write tests for projects you are contributing to?	Likert	All
15	How often do you write a test or multiple tests with (just) the intent to improve the code coverage?	Likert	All
16	How often are you asked/encouraged to better test your contributions, in the comments of a pull request you opened?	Likert	Contributors
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	Contributors
18	How often do you have to tell a contributor to a project you maintain that their tests need to be improved, based on the results of a coverage tool?	Likert	Maintainers
19	Do you remember a particularly interesting instance where this (see Q18) happened? How did that situation get resolved?	Open	Maintainers
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	All
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	All
22	Can you come up with situations where you would ignore a failing coverage check? What are your reasons?	Open	All
23	How often do you neglect or ignore a failing coverage check on a commit or pull request?	Likert	All
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	All
25	Can you give us 2 things you dislike about using a code coverage tool on an open-source platform?	Open	All

5.1.3 Adjusting the survey

To verify the survey, we ran four trial rounds with PhD students. These students are part of the Software Engineering Research Group at TU Delft, and therefore were familiar with (open-source) software development, and tools such as GitHub and code coverage tools. During these recorded sessions, the students would take the survey whilst thinking aloud.

Afterwards, there was an extra moment of reflection and some feedback on the survey. Here we were able to figure out what works, and what does not. From these students we have received multiple main points of feedback regarding:

- The length of the survey. Most students would consider the survey to be on the lengthy side. The main cause of the length was due to the high number of open questions. As a result of this, some questions were cut from the survey, merged with others, or changed from an open question to a (slightly rephrased) closed question.
- The occasionally questionable Likert scale options. For example, some questions regarding the frequency of code coverage tool usage, had options that were unrelated to actual development workflow, and more to units of time measurement, or vice versa. To remedy this, we rephrased the questions and options in between trial sessions, and made sure to ask the next students to give feedback on them.
- The lack of images and clarity in the survey when describing different functionalities of code coverage tools. We fixed this by including some clarifying images, along with textual explanations. We also improved the overall design of the survey and added different typographic elements to further distinguish questions from context information or explanations.

Overall, the students were mainly positive towards the survey. After each trial round, we discussed the feedback on the survey before improving it further. Afterwards, we discarded the trial rounds from our survey results, so they do not make up a part of the results section. The trial rounds were only used to improve the survey.

5.1.4 Inviting the participants

The target audience for our survey was developers familiar with using code coverage tools on open-source development platforms. We chose this target audience for several reasons. Firstly, this is a diverse group of people. Open source developers work on different projects, in different languages, performing different tasks, etc. This diversity makes for potentially generalizable results. For example, will their opinions vary significantly, or is there a kind of consensus among such a diverse group? Secondly, there are a lot of open-source developers. With a very large pool of users, there is the potential for a very large sample size, and thus a large set of result data. The third reason was that, due to the nature of open-source development, where developers work publicly and online, finding a large set of potential participants is convenient and relatively simple. Furthermore, since we are trying to explain why status checks are ignored, it makes sense to ask the developers who ignore them.

To find these participants, a good start would be to look on open-source development platforms. Here we mainly focused our efforts on GitHub, because it is the largest platform [7]. Our goal was to search for users on GitHub, which have contributed to projects that use code coverage tools. However, since GitHub provided us with limited resources[2] to

[2]The public GitHub API has a rate limit

find these users quickly, we turned to the large database of coverage information we already had available from Codecov.

In the Codecov database, we looked for 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. This 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. Private accounts were therefore removed from our dataset.

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 prevent absolute newcomers from taking the survey. While it is okay (and even good) for participants to have varying levels of experience, we do expect a certain baseline. Also, we did not want people who have not used these tools in a very long time. Partially because the tools themselves evolve over time, and partially because we did not want to bother them, in the case they were an inactive user on the platform. This filtering resulted in a list of 11 000 people, of which we randomly selected 2000 to send our survey to, by email. We opted for 2000, to account for a potentially low response rate of 6%, which was a rough estimate based on previous studies in software engineering [29, 25].

5.1.5 Analysing the results

In December 2021, our survey was sent out to 2000 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.

In total, there were 379 response attempts in total, and 278 complete responses of people who finished the entire survey. This means our survey had a response rate of $\approx 14\%$.

The responses were downloaded from Qualtrics and processed in Python using Pandas. Responses were divided into groups, based on whether the participant considered his/herself more of a contributor or more of a maintainer. These groups were analyzed individually, and subsequently compared against each other.

The full results can be found in Section 5.2.

5.2 Results

This section describes the results from the survey. We have downloaded the results for each survey question, and processed them accordingly, based on the nature of the question, using Pandas and PyPlot.

The individual subsections will dive deeper into the results for each type of question. But the process of analysis has been largely the same for all of them. Each question was processed separately, and we put the answers in the graphs below. There was one demographic question, which determined which of the two paths the participants would take in the survey. We have therefore divided these participants into two groups, on which we report the results for the questions separately, so we can compare and contrast the two.

The groups are ‘contributors’, which are developers which more align themselves with opening pull requests in projects, and ‘maintainers’, which align themselves more with reviewing and merging pull requests, and overall maintaining the project. Both groups received the same questions, although some were written from the perspective that better aligns with the chosen group.

As previously mentioned, the total number of survey responses was 379, of which 278 were completed entries. We have only used these completed entries for the interpretation of the results.

5.2.1 Demographic

The demographic results describe the background of the participants for the rest of the survey. They provide perspective to the rest of the results, aiding their interpretation.

General experience The first demographic questions are about the participant’s experiences with software development overall. In Figure 5.1 we see that a large majority of the participants have several years of experience with software development. Consequently, we also find that most participants have more than a little experience with software development tasks such as testing and code review.

A small, but interesting find here is that the third graph in the figure shows the rise of automated testing in the past couple of years: When you look at the bars for 5-10, 10-20 and 20+ years, there appears to be more manual testing than automatic, while the opposite happens for 1-3 and 3-5 years. For $<$ 1 year we see that manual testing is greater again, likely due to how much easier it is for a new developer.

Furthermore, while most participants actively work in a software development field, there is also a substantial number of participants that do not. For the participants that do work in software development, we also asked them for their job title. Figure 5.2 shows a word cloud of the different job titles. From this it shows that 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 at a high position, such as CEO, while others have listed themselves as interns. We also (manually) clustered the different occupations into larger categories, which are shown in Figure 5.3.

These figures are meant to show the diversity of the participants, which is important to show how many people will be affected by this study. Furthermore, one could argue that the results would be more generalizable than in the case where this study would be held in a single company or department even.

Open source experience The next set of questions pertain to participants’ experience with open-source software development platforms, and the frequency at which they perform

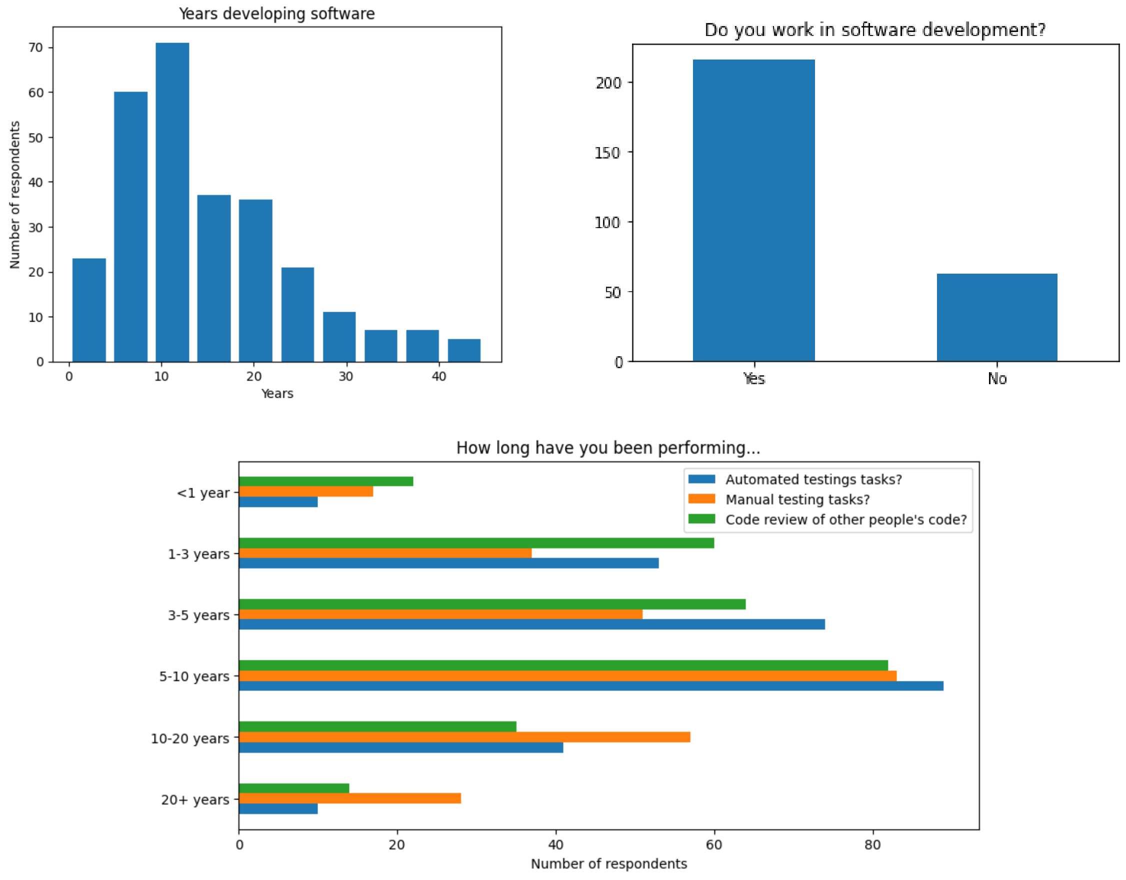


Figure 5.1: General experience with software development

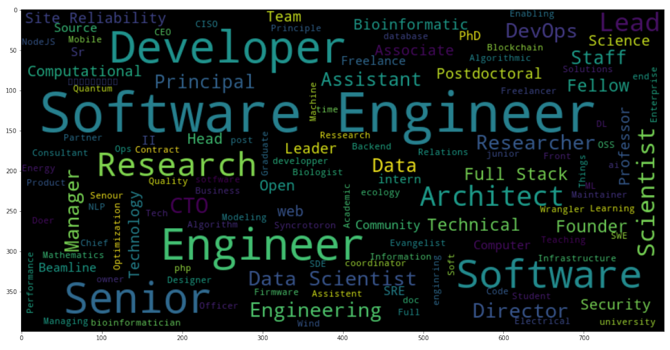


Figure 5.2: Word cloud of job titles of participants

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

Figure 5.3: Distribution of participants' occupations

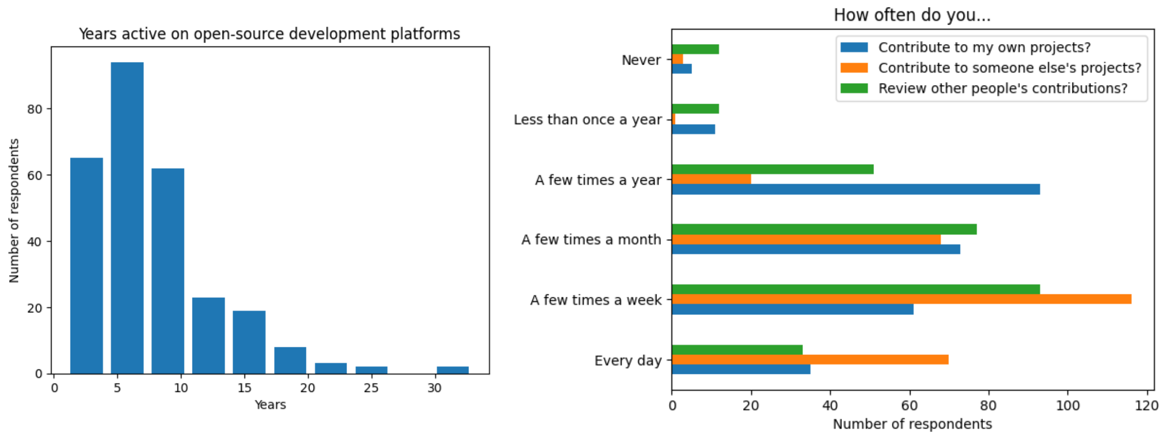


Figure 5.4: Experience with and frequency of open-source development

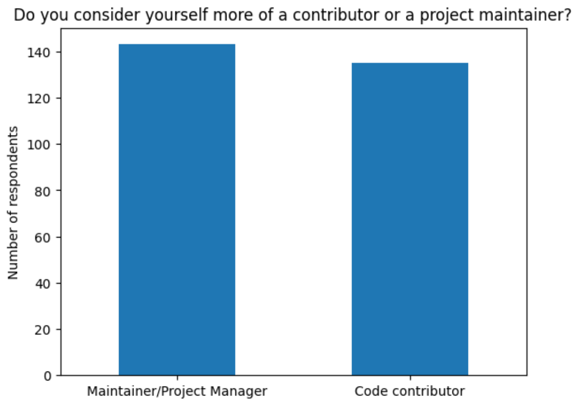


Figure 5.5: Maintainers and Contributors

certain tasks related to those platforms.

In Figure 5.4 we find that most participants have experience with 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 forms of distributed version control or source code management.

In the second graph we see that contributing to someone else's projects is by far the task that is performed the most frequently, whereas contributing to own projects is considerably less frequent. Furthermore, we find that some participants rarely or never perform one or more of these tasks.

Contributor or maintainer The final demographic question we asked was the decision for whether a participant considers themselves more of a code contributor, or a maintainer.

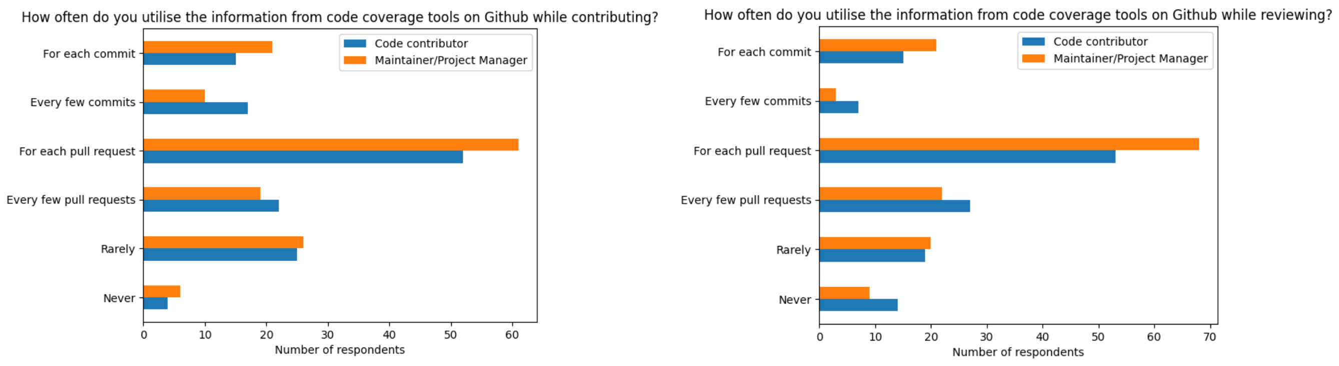


Figure 5.6: Code coverage tool frequency

As you can see in Figure 5.5, the two groups are almost of equal size. For future work it would be interesting to see how these group sizes would change on a much larger scale.

The answer to this question determines some of the questions the participants will see further in the survey.

5.2.2 Quantitative

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

Code coverage tool usage The first set of questions is regarding how often the participants use code coverage tools. From Figure 5.6 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 to both questions with ‘Never’, this would mean that they lack the experience that is required for the remainder of the questions in the survey. Therefore, we would have them skip the rest of the survey, and instead get an alternative ending. In total, this happened 9 times, causing the number of responses for the remaining questions to go down to 269.

We also asked participants how often they used code coverage tools outside of GitHub, for example on their local device while developing. The results can be found in Figure 5.7.

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, which can be seen in Figure 5.8. However, as can be seen in the figure, not all respondents feel that way. In fact, there is even one person who answered that automated software testing is not at all important. We will dive deeper into this in Section 5.2.3.

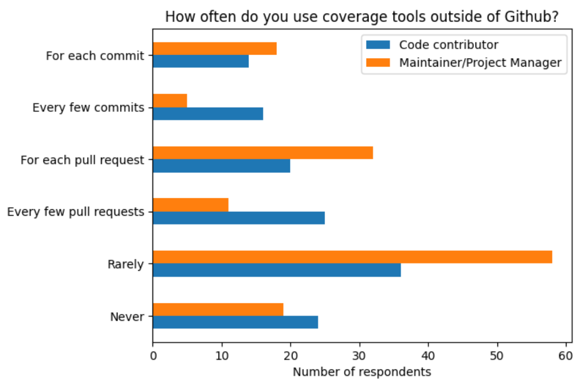


Figure 5.7: Code coverage tool usage outside of GitHub

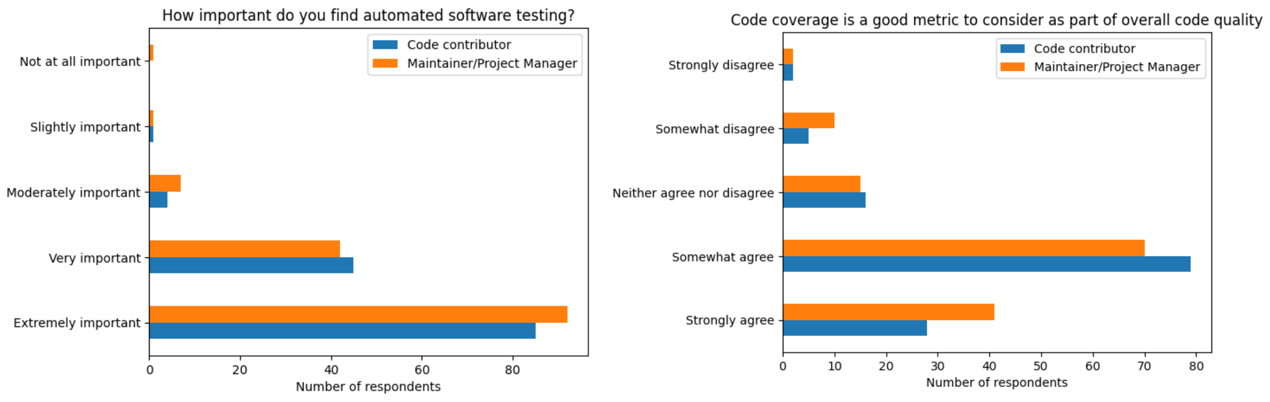


Figure 5.8: Importance of testing and code coverage

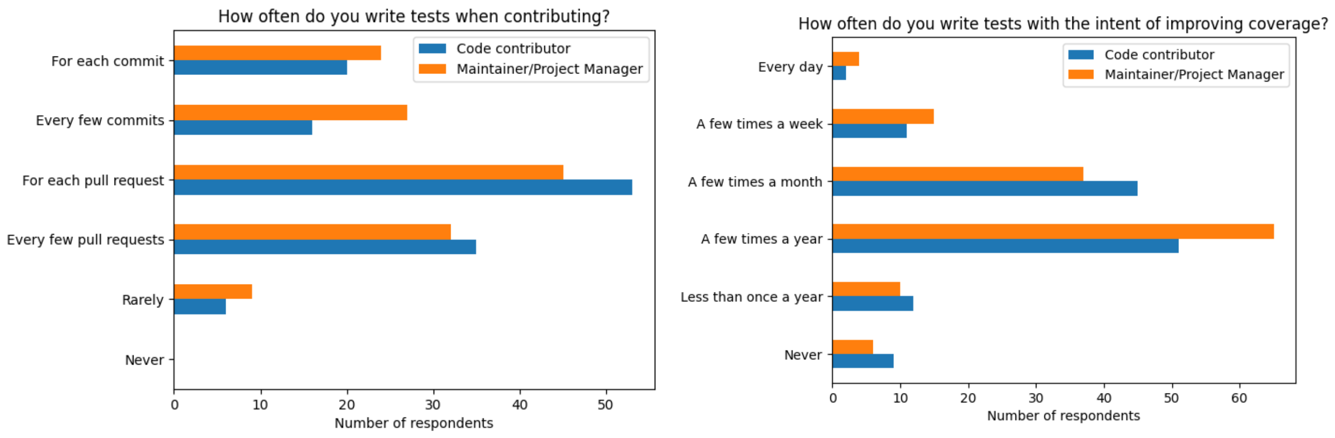


Figure 5.9: Frequency of writing tests

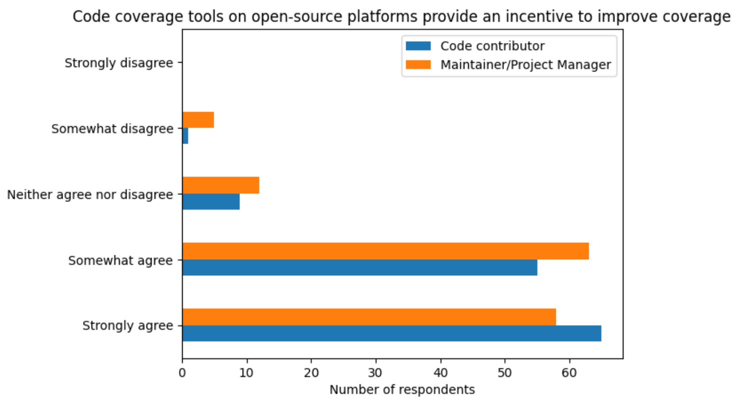


Figure 5.10: Incentive to improve coverage

Frequency of writing tests Figure 5.9 shows the results for two questions asking how often the participants write tests. One finding that immediately strikes us as interesting 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 again "For each pull request" is the most popular answer for the first question.

Additionally, according to the second graph, almost all participants admit that they write tests with the sole purpose of improving coverage, with a frequency greater than "Never". Some even do this every day.

Coverage and incentive We asked the participants a considerable number of questions relating to the incentive that code coverage tools provide for writing tests and/or improving the coverage metric(s) of a code base.

Firstly, when we look at Figure 5.10, we see that almost all participants agree, either somewhat or strongly, that code coverage tools provide an incentive to improve coverage. Furthermore, it appears that there is no big difference of opinion between the contributors and the maintainers. However overall, the contributors lean a bit more towards the "Strongly agree" than the maintainers.

Secondly, we asked contributors and maintainers near identical questions (See Table 5.1) regarding the acceptance rate of pull requests, based on their coverage levels. In Figure 5.11 we actually find a big difference of opinion between what contributors believe, versus what maintainers claim.

Lastly, we also asked participants to rank different features of code coverage tools, from what they consider giving the most incentive to the least incentive to improve coverage. In Figure 5.12 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.

From the graphs it is clear that status checks are the clear winner. Nevertheless, every other feature has also been ranked first by one or more participants at some point. Furthermore, we see that contributors and maintainer very much share the same opinion.

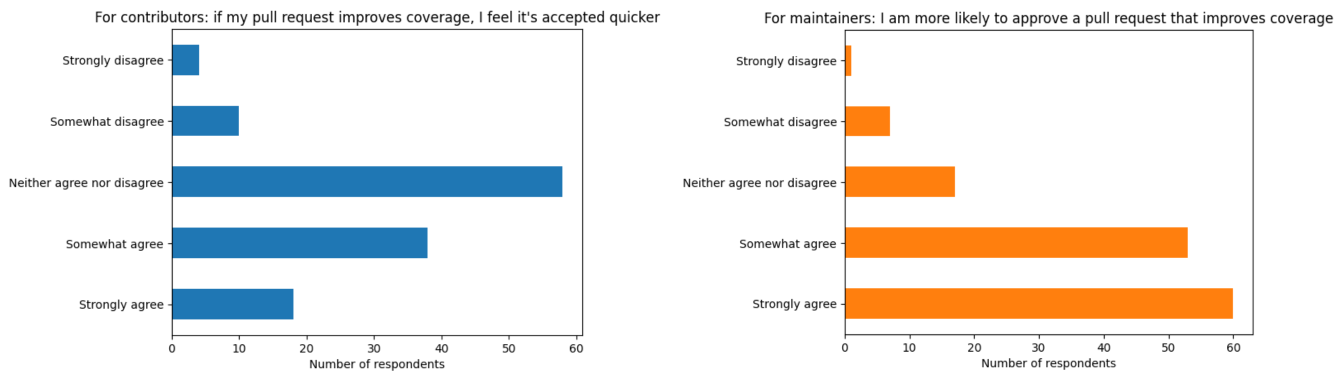


Figure 5.11: Beliefs about pull request acceptance and code coverage

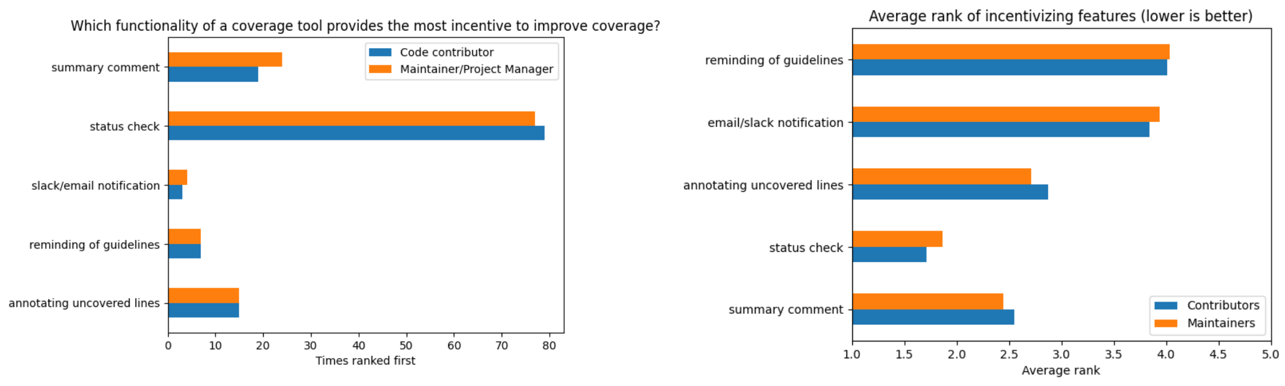


Figure 5.12: Ranking the most incentivising feature of code coverage tools

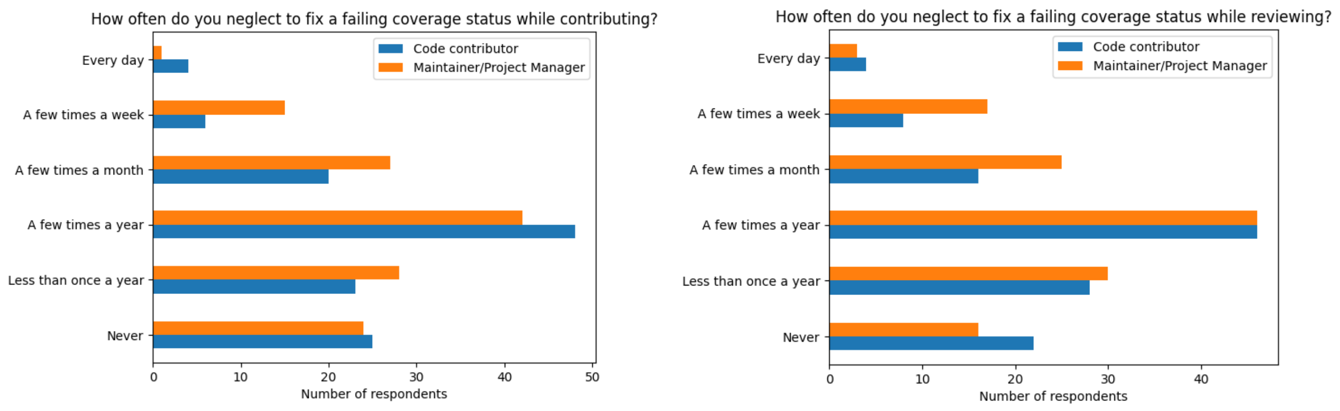


Figure 5.13: Frequency of ignoring a coverage check

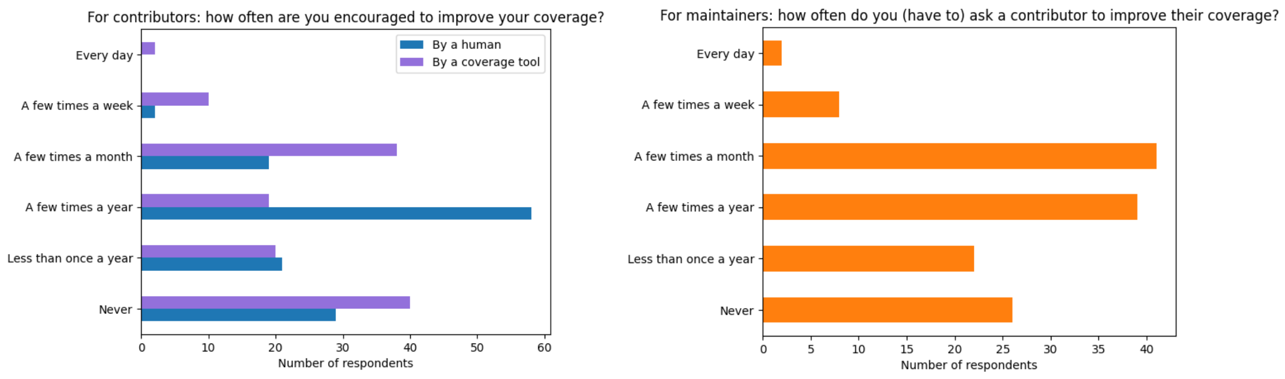


Figure 5.14: Frequency of asking to improve coverage

Neglecting the coverage check Figure 5.13 shows the frequency with which the participants ignore a failing coverage check when both contributing and reviewing. The first thing that strikes us as interesting, is that both graphs display the same trend, for different (but related) tasks. Furthermore, we see that maintainers tend to ignore failing coverage checks slightly more frequently[3].

Also interesting, is that there is a non-negligible group of participants, who claim that they never ignore a failing coverage status.

Pointing out decreasing coverage The final set of quantitative questions consider how often the respondents are asked to improve coverage, or need to ask other contributors to improve coverage. The results can be found in Figure 5.14. Here we find a couple of interesting things.

Firstly, a large group of contributors says 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, as the bars for the more frequent options are larger for the coverage tools.

Thirdly, when we compare the figure for the maintainers, with the figure for the contributors (specifically the "By a human" bars), we again see very similar bars, with an 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. That is, maintainers overall review more varied pull requests, than the contributors open, and therefore must ask to improve more frequently.

[3]taking into account the slightly different group sizes

5.2.3 Qualitative

For our analysis of the qualitative (open) questions, we used Axial coding. We wrote a script that, for each question, allows us to go down the list of responses, and assign them one or more codes. We could re-use previous codes using an autocomplete function, or assign new codes. In the case a response was unclear and could not be given a code, it was discarded. Examples are responses which do not answer the question or answer it with a single word only.

Afterwards, we grouped the codes into larger categories of responses. For each question, we listed the codes that were generated from the previous step 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 categorised, 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 work was also verified by a supervisor from TU Delft, and an advisor from Codecov, to prevent bias and/or incorrect codes and categories. The full list of responses and codes can be found on GitHub: `https://doi.org/10.5281/zenodo.8044949`.

After we labelled each response with its codes and categories, 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 categories. Furthermore, the results are grouped per participants' roles as either a contributor or a maintainer again. These results can be found in Tables 5.2 through 5.7. 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 results and categories of responses can be found in Table 5.2.

The first thing that piques our interest when looking at the table, 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 number as a coverage goal in their responses. This is not unsurprising, given that code coverage can be expressed as a single metric, that we as developers would like to maximise (with minimal effort). However, for both groups at least 20% of the participants believe that striving to achieve some arbitrary goal is not the right goal to set.

Interesting merge stories We asked contributors if they could remember any particularly interesting situations where they were asked to improve the coverage on their pull request. We asked a similar question to maintainers, but whether they remember a situation where they had to ask someone to improve their coverage.

According to Table 5.4, 15% of all maintainers mention that after they brought up the coverage issues, the contributors sorted them out. Furthermore, 6% reports that they occasionally find contributors are (intentionally) avoiding writing tests, which is quite an interesting phenomenon. A possible reason could be that contributors consider some code

Category	Summary	Example	Contributor mentioned	Maintainer mentioned
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 int 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%)

Table 5.2: What is a good coverage goal?

difficult to cover, which is something that 6% of the contributors mention, according to Table 5.3.

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 Table 5.5.

In the table, we see 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 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, can be a good way to provide incentive. No further explanation for this is given, but from our own experience we find that better guidelines lower the barrier for contributing,

Category	Summary	Example	Contributor mentioned
Maintainer interaction	Participants reported they had spoken to a maintainer and fixed the problem	"I just added more tests. The requests from maintainers is typically warranted, and I honor that request out of respect for those maintainers."	13 (10%)
Tool interaction	A coverage tool warned the participant and they fixed the problem	"whenever my PR got blocked due to missing Unit Tests, I resolve it by adding more Tests. But it rarely happens."	12 (9%)
Unclear interaction	The interaction was not clear in the response, but the problem got solved	"I added a commit with the requested unit tests to cover the additional functions I added to the software package. Not really a big deal."	12 (9%)
Difficulty testing	The participant reported that they had difficulty writing tests in that situation	"Some time it's very difficult to cover some corner case code. Then, you have to turn off diff coverage thresholds."	8 (6%)

Table 5.3: For contributors: Do you remember a particularly interesting instance where you were asked to improve coverage?

Likes and dislikes Towards the end of the survey, we asked the participants to give us two reasons why they like using coverage tools, and two things they dislike about them. Table 5.6 shows the likes, and Table 5.7 shows the dislikes.

Firstly, we see in the likes table, 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, in no other category is the difference between the groups this big.

However, there is a similar big difference in the dislikes table. 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 interesting 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 likes table 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. These

Category	Summary	Example	Maintainer mentioned
Blocked merge	Participants mention either ignoring or blocking a pull request until it is fixed	"If the contributor doesn't improve the tests, the PR gets closed or suffer from starvation until I can fix it on my own"	6 (4%)
Problem fixed	The contributor fixed the problem after being asked	"new code is not well tested, then contributor must test all cases, then new code is 95% coverage"	21 (15%)
Coverage bad	The coverage tool was not behaving properly	"In one repository, the code coverage tools were broken. I correctly set it up again, and then made some improvements to make it from 50% to 63% coverage"	4 (3%)
Contributors lacking	Participants report that they found the contributors to be (intentionally) lacking with writing tests	"Some developers consider test coverage a mundane task and use creative ways to get around it."	9 (6%)
Interaction	Maintainers mention longer discussions with the contributors	"Some people are not used to write tests, and tend to disagree and say tests are not important. It is not always easy to deal with this kind of developer, but it is not a rare case."	11 (8%)
Coverage good	The coverage tool helped the maintainer in this situation	"New code had low coverage rate. Turned out that a few important branches were not tested... and had bugs. So code coverage helped."	4 (3%)

Table 5.4: For maintainers: Do you remember a particularly interesting instance where you had to ask to improve coverage?

responses were probably left by different people.

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. Rightfully this is a concern. The second largest dislike is that the results of the coverage tool can be incorrect, irrelevant or too small to really matter.

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 5.8.

The categories of responses that were mentioned most often include a trivial or minimal coverage change, a higher priority to merge the changes that ensure they are properly tested, the difficulty of testing the code changes, and the coverage failure being seemingly incorrect. We provide a further discussion on these reasons in Section 6.3.

Category	**Summary**	**Example**	**Contributor mentioned**	**Maintainer mentioned**
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 5.5: What is the best way to incentivize improving coverage?

Category	Summary	Example	Contributor mentioned	Maintainer mentioned
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	Optics are good for 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%)

Table 5.6: Two things you like about coverage tools?

Category	**Summary**	**Example**	**Contributor mentioned**	**Maintainer mentioned**
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 5.7: Two things you dislike about coverage tools?

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 5.8: When would you ignore a failing coverage check?

5.3 Threats to validity

One of the main threats to validity is the possible bias we introduced while categorising the different qualitative responses. In an attempt to mitigate this threat, we used a process in which the first couple responses for each open question were analysed by the author of this thesis and a daily supervisor from TU Delft, who then compared their categories. In this process, no major differences were found. Afterwards, the rest of the responses were categorised by the author of this thesis, and later verified by the supervisor, as well as by another advisor from Codecov.

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.

Furthermore, we were limited by both time constraints and GitHub’s API to search for all potential users of coverage bots to send the invite of our survey to. As such, we had to narrow our search by first querying people from Codecov’s database, as it stands to reason that all of them have used Codecov before. This in turn introduces a bias in our results. While all our participants have used Codecov before, it does not imply that they have used other tools before. In hindsight, It could have been worthwhile to include a question on which tools they were familiar with.

From our dataset of users, we randomly sampled 2000 to invite. And while not all of them took the survey, we did end up with a pretty even split of 135 contributors and 143 maintainers. This split was also very important for us, because it meant that we could gather opinions and experiences from different perspectives and developer profiles. This means that we mainly get generalisable results, and we care less about the intricacies of individual projects. As such, we will not have identified every possible reason for ignoring the coverage check, or be able to provide detailed explanations for them. However, we believe that our strategy is still a good fit for answering this question.

Lastly, we tried to mitigate any inconsistencies and unclear questions in the survey, by running trial versions with PhD students. These students provided multiple items of feedback and possible suggestions for improvement, which were used to adjust the questionnaire. This process is further described in Section 5.1.

5.4 Conclusion

This chapter set out to answer the question of why developers would ignore a failing coverage check while opening or reviewing a pull request. We answered this question by setting up a survey for developers who are familiar with code coverage tools, and thus have encountered failing coverage checks. The survey consisted of a mix of demographic questions to gather background information on our participants, and both open and closed questions regarding the use of code coverage tools on GitHub. To verify that our survey was clear, we ran trial sessions with PhD students from TU Delft and used their feedback to improve the survey.

We used Codecov's database to find usernames of GitHub users of whom we know that they have used Codecov (a certain code coverage tool). We then filtered these usernames based on how many commits they have made, and when their last commits were. After this, we used the GitHub API to link usernames to public email addresses. From these addresses we ended up sampling 2000 emails and invited those developers to our survey.

Out of 2000 potential participants, we ended up with 278 responses. These responses were processed and analysed. For closed questions we used graphs or tables to represent the data, and for the open questions we used axial coding to assign each response to a question a certain label. To prevent bias these labels were verified by daily supervisors.

From our results, we conclude that participants reported multiple different reasons for ignoring a failing coverage check. The most common given reasons were minimal/trivial coverage decreases, little faith in the correctness of the coverage tool and/or the coverage information, the priority of merging the pull request due to time constraints or high desirability, or the complexity of testing the code changes. These reasons serve as our answer to our proposed research question "Why are coverage checks ignored?". In Chapter 6 we will further dive into these reasons and propose solutions for them.

Chapter 6

Discussion

In this thesis we set out to answer three different research questions regarding the use of code coverage tools on open-source development platforms. We have gathered data by querying Codecov's database, by making use of GitHub's public API, and by distributing a survey among open-source developers.

In this chapter, we will look at our results and discuss them in more detail. We will explain our most interesting findings, relate them to existing scientific work, and talk about their implications for developers and researchers.

6.1 RQ1: Is there a relationship between code coverage and the time to merge a pull request?

In Chapter 3 we set out to answer the question *"Is there a relationship between code coverage and the time to merge a pull request?"*. As we can see from the results in Section 3.2, the short answer to that question is: it depends on the project. While this is not as conclusive of an answer as we would have liked, it is better than a definitive no. In this section we will discuss some of our findings.

The relationship between code coverage and time to merge Table 3.2 shows the aggregated statistics for each of the metrics we explored. From it we can see that each of our code coverage metrics had a significant relationship with the time to merge at least once. Meaning that for each metric, there exists a project for which this metric holds a significant relationship with the time to merge. However, there exists no coverage metric that has a significant relationship in a majority of the models, with the most frequently significant metric only being significant in 50% of the models.

However, when looking from the perspective of the projects, instead of the metrics, we do find that 14/18 projects have at least two significant coverage metrics. This is shown in Figure 6.1. Therefore, it is definitely possible for there to exist a relationship between code coverage and the time to merge in an open-source project.

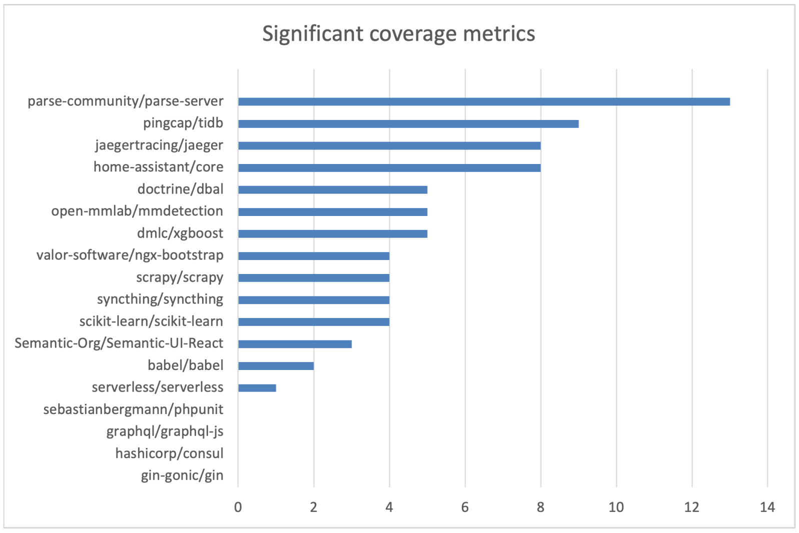


Figure 6.1: Significant coverage metrics per project

The difference between patch coverage and project coverage As shown in Table 3.2, all the patch related metrics tend to be significant the least often, of all the other metrics. While project related metrics are concerned with more 'absolute' numbers, such as the overall coverage percentage of a repository before and after a PR, the patch coverage metrics are about the coverage of only the code that is changed in the pull request.

One possible theory we have for the lack of significant patch coverage metrics, is that patch coverage tends to either be: 0%, i.e., the changed code was not tested, or 100%, meaning that the changes were covered. It is relatively rare to find pull requests where the modified code was partially covered.

Looking at all pull requests of the selected models in our dataset, we find that 3012 out of 5760 pull requests had a patch coverage percentage of 100%, 1095 had a percentage of 0%, and 1653 had some other value. So, with a large majority of pull requests all having the same value, it is possible that the metric itself holds little useful information for the models.

6.2 RQ2: To what extent do comments and status checks lead to improved coverage?

In Chapter 4, we sampled 400 pull requests that, at one point, had a failing coverage status check. These were then divided into two groups. One where the coverage at the most recent commit of the PR goes down, and one where the coverage goes up or remains the same. We then searched the comments and commit messages of these PRs to look for mentions of testing or code coverage.

Relation between coverage and mentioning coverage From our results in Section 4.2.1 we find that there is a statistically significant relation between the mentions of tests and coverage, and which group the pull request belongs to. Therefore, we can say that there is a relation between the mentioning of tests and coverage in a PR, and whether the coverage goes up or down, after an initial failure.

While this may seem intuitive, our investigation now provides empirical insights into this relationship.

Ignoring the coverage check We also looked closer at each pull request and tried to find a specific mention to the failing coverage check. Table 4.3 shows the division of pull requests into separate categories of whether the failing check is mentioned, as well as whether it is properly addressed or fixed. According to this table, a large majority of pull requests have no explicit mention of the failing coverage check. This leads us to believe that it is very easy for a coverage check to be ignored.

We can come up with several reasons for this. First of all, in most of the PRs, the (initial) drop in coverage is minimal, usually $< 1\%$. A possible explanation is that the projects these PRs belong to, have large codebases, and a typical PR only changes relatively few lines of code. With such a small change in coverage, it makes it easier for people to ignore, as long as the overall project coverage is still considered good. Secondly, the projects in our sample happen to be very large, with complicated CI setups. Tests are run across a number of different system architectures and possibly contain flaky tests. Therefore, it can be difficult to attribute a failing coverage check to the actual changes made in the PR. These could all be possible theories as to why the coverage checks are then ignored. We have already listed our sampling as threat to the validity of our results.

6.3 RQ3: Why are coverage checks ignored?

In Chapter 5 we posed the question *"Why are coverage checks ignored?"*. To answer this question, we conducted a survey of various developers and project maintainers on GitHub. In this survey we asked them a number of questions regarding their experiences with using code coverage tools on GitHub and in what scenarios they would ignore a failing coverage check, and what provides the best motivation to improve code coverage. This section will discuss our findings as reported in Section 5.2.

Ignoring the coverage check The main goal of this survey was to find an explanation for the large number of merged pull requests, that also had a failing coverage status. As such, we asked the participants of our survey in what scenarios they would ignore a failing coverage status. Here we summarise our main findings from Table 5.8, which shows the different types of responses we received to that question, as well as how often they were given.

Firstly, coverage failures tend to be ignored when they are minimal. For instance, less than 1%. 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 [22]. 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. This was also briefly mentioned in the previous paragraph. Arguably, this situation makes the most sense to ignore the failing check, as the failure is not anyone's doing.

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 might see a small difference of opinion between contributors and maintainers.

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. For example, code for a GUI is not the same as some business logic.

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. That is much more relevant to maintainers than contributors, of course.

Usage of coverage tools When we look at the results from Figure 5.7, we see that many more respondents claim they rarely or never use code coverage tools outside of GitHub. What is interesting here is that when we compare these results to those from Figure 5.6, most participants claim they do use code coverage tools frequently on GitHub.

Our interpretation is that many participants therefore mostly rely on the coverage results generated on/posted to GitHub, instead of running these tools locally. In turn, this has some consequences for the overall development process, as this means users will have to wait for the remote CI pipelines to finish to get coverage information. At the same time, this saves developers from setting up local coverage tools, which can be difficult to set up correctly.

This lines up with findings from an earlier study, which shows that developers do not rely on running tests in their local IDEs [2]. Consequently, if developers do not run tests in their IDE, they also do not run coverage tools in their IDE.

Perceived effects of code coverage tools One interesting finding is that most maintainers strongly agree that they are more likely to accept a pull request that improves coverage. However, most contributors do not feel like this is actually the case. This is interesting because there seems to be a big difference of opinion between maintainers and contributors.

This difference can be seen in Figure 5.11. It shows that contributors do not necessarily agree with the idea that their pull request is accepted quicker if it improves coverage. However, according to the maintainers, this is in fact the case for a large majority of them. The claim of the maintainers appears to be an entirely new finding, which influences the way pull requests are reviewed. This means that even though maintainers are more likely to accept a certain PR, ultimately this has a low impact from the contributor's perspective.

At the same time, there exists a related study that has found that adopting code review bots leads to less time for maintainers to reject a pull request. However, the time to accept a pull request does not change [28]. What is intriguing about this study, is that it looks at the actual data of projects, and not subjective opinions from developers, which is what we do in our study.

Our theory is that in the case of a PR that lowers coverage, the decision to reject it is made rather quickly, due to the readily available quality information (including coverage), whereas in the case of a PR that improves coverage, there can be other factors at play, that slow down the likely acceptance.

Opinions of code coverage tools Our study highlights many different positive effects of using code coverage tools, such as an easier workflow and/or higher quality code, as reported by the participants. However, more interestingly are the perceived negative effects. The most commonly reported negative effects, for both maintainers and contributors, are that the results from code coverage tools can be incorrect, and that code coverage tools encourage writing tests that cover the code, but do very little to encourage writing high-quality tests.

Some of these mentioned effects were already discovered in an earlier survey performed with maintainers about code review bots. Like ours, this earlier survey also finds that maintainers perceive that there is less effort required, and that the bots lead to a higher code quality [29]. Furthermore, they also found that code review bots can be noisy and scare away newcomers. However, what makes our study different is the fact that we interviewed maintainers as well as contributors. Moreover, we found a lot more negative effects, which can be found in Section 5.2. For example, the setup of these tools is considered complicated, and the results are often found to be incorrect. The tools can also take a while to run, which impacts the developers timewise.

6.4 Implications

Now that we have posed answers to our research questions, we can discuss the implications of our work. This section is divided into practical recommendations for developers, and theoretical recommendations for the developers of code coverage tools.

6.4.1 Open source developers

In Chapters 4 and 5 we found that failing coverage checks tend to be ignored. We determined a number of reasons for this and came up with recommendations for open-source developers to alleviate this problem.

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[1]. This way, the check provided by the coverage tool will become meaningful again. Furthermore, developers can also configure the tools to decrease the noise they create, by changing settings for the comments left on PRs, or disabling the comments altogether. Or

[1]For example: `https://docs.codecov.com/docs/codecov-yaml`

perhaps a more interesting approach to reduce noise could be grouping the comments from different bots into a single report, which was demonstrated in a paper by Wessel and Steinmacher [27].

Another big reason for neglecting fixing the coverage is simply the high complexity of testing (certain parts of) the code. We can think of a number of ways to make this easier. Firstly, by improving the structure of the code, preferably in an early stage of the project, to something that is easily testable[2]. Secondly, by writing clear documentation for (new) contributors on how to properly test the code. Another advantage of this is that potentially less pull requests would be abandoned by newcomers.

Something that ties into this, is writing clear guidelines for contributing, but with a clear goal for testing and/or coverage targets. While the results from Table 5.2 can provide some inspiration, it is also important to be agile, and change the goals to suit the current state of the projects. Tables 4.1 and 4.2 show that a mention of tests or coverage is correlated with a failing PR getting fixed again, so it would also be a good idea to discuss these while reviewing PRs and provide assistance with testing when necessary and/or requested.

6.4.2 Code coverage tool developers

Table 5.7 consists of many different aspects in which code coverage tools are perceived to be lacking. As such, it provides the developers of code coverage tools with many opportunities for improving them.

Some of the biggest problems that are mentioned by both the maintainers and contributors are: unclear or wrong results, a difficult setup process, and noise. Some basic ideas here are: improving the robustness of their tool such that incorrect results occur less frequently, reducing the complexity of the set-up process, and reducing the noise of the tool.

For the setup of code coverage tools, there tends to be some kind of configuration file that is committed to the repository. If this file is not present, either the tool cannot work, or it works with its default settings. To make setting up these tools easier, a nice idea might be to create an interactive tool, either online or through a CLI, that asks developers for their preferences, and outputs a configuration file. Additionally, the tool should be able to explain what each entry in the file means. Another idea is to give users a warning in their console output, if the default configuration is used, and to provide them with a single command or URL to create a configuration file.

For the robustness, it might be worthwhile to investigate adding ways to ignore individual methods or lines that cause problems. Or possibly addressing the problem of flaky tests by detecting them and adding this to the coverage information [26]. This could provide extra information to reviewers that lets them know better where the decrease in coverage is coming from.

Other ideas include providing more actionable information to both contributors and maintainers, judging the actual quality of tests, instead of just how much code they cover, and improve the workflow of the users of the tool. For example, by providing a tool that

[2]While we cannot give specific recommendations for this topic, there exist plenty of good resources on software testability online [21]

can report coverage information from the cloud directly to a developer's IDE, which can already be done with test failures [3].

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 be nice if code coverage tools provided an optional integration with issue tracking systems (i.e., GitHub issues, Jira, Asana, etc.), to (automatically) open new issues for uncovered, but merged, code changes. From a developer's perspective this could reduce the amount of effort needed to keep better track of which parts of the code still need tests, in a system they are already familiar with and frequently use.

6.5 Future work

The research we have conducted for each chapter, and the results we have found, present new opportunities for interesting future work regarding various aspects of code coverage.

Firstly, we think it would be worthwhile to repeat our experiment of Chapter 3, but with a larger group of projects, PRs, and metrics. We simply did not have the time and resources to go above and beyond for this research question. We were limited by the scope of GHTorrent and BigQuery on how much data we could download. We simplified this process by only focusing on popular repositories, but in doing so we also excluded a large number of projects. With more time and physical storage, more data can be collected. We believe there still is a lot that we can uncover. Not only is it possible to find relationships between code coverage and other variables, other than the time to merge, but there are also a lot of metrics that we simply could not include. Like the 'hidden' metrics or human factors we described in Section 3.2 to explain the high variability of the intercept variable. Additionally, different models can be tried, and their results can be compared.

Additionally, since we found that there are some projects that have a significant relationship with code coverage, and some that do not, another idea would be to investigate the similarities and differences between these projects. Furthermore, in previous work we have seen methods to estimate the amount of effort that is required to complete a pull request [17]. This model aims to predict how long a PR stays open based on a number of simple metrics, such as lines of code affected, but also more inventive metrics, such as the day of the week. We suggest adding code coverage metrics to such a model, and seeing whether this makes it perform better or worse, and what kind of correlations these metrics would have.

Another idea we came up with, is to track the progression of code coverage throughout the lifespan of a pull request or project. We can think of a number of questions that we could answer with this data. For example: How long does it take from a failing commit to go to a successful commit? How many commits does it take? Are there conversations in between these commits? And so on.

Lastly, because we have found that patch coverage tends to be either 100% or 0% most of the time (see Section 6.1), we should think about what improvements we can make there. It is believed that software follows a power law distribution [16]. This implies that there are a few certain parts of the code that are executed very often, while most other parts run much more rarely. This means that software often depends on a few critical parts of code. Therefore, changes made to this code should be under more scrutiny and be reflected more strongly in a metric such as patch coverage.

A possible idea to explore is that of a weighted patch coverage, which depends on the importance of the code that is (not) covered. What is unclear yet is how that importance would be measured. For example, it could be the code's cyclomatic complexity, how often it is executed during tests, or, in the case of a function or class, how many lines of code it has, etc. We consider the implementation of such a metric as possible future work.

The results of our survey provide ample possibilities for future studies. Firstly, a more in-depth study of how code coverage (tools) provide an incentive to keep up code coverage and code quality would be beneficial. From our study we found that even though most participants strongly agree that code coverage tools provide an incentive to improve coverage, the contributors do not necessarily believe that it increases the likelihood of their pull requests getting accepted quicker. Therefore, there must be some other driving factor that motivates them to improve the coverage.

Another potential subject of a future study is how to assess the actual quality of test code. As we have seen, a large number of participants mention that they dislike code coverage being mistaken for quality, since it is possible to write tests that cover the entire code, without actually verifying its behaviour. As such, it makes sense to have a tool that can assess the quality of a test, that can also be used during open-source development in the same way a coverage tool can. I.e., easy to set up and readily available. Of course, there already exist tools that provide some form of code quality information, such as static analysis tools. However, the question is whether those tools are also applicable to test code specifically. Furthermore, research for test quality already exists in mutation testing, for example, but these procedures are not suitable to run (swiftly) in a CI pipeline, and perhaps costly to set up for a contributor that just wants to contribute a few lines of code. Therefore, there should be a more intelligent tool, which can potentially use coverage data to discern meaningful information regarding the quality of how this coverage came to be.

Chapter 7

Conclusion

In this thesis, we investigated the use of code coverage tools on open-source development platforms and aimed to answer three research questions. We collected data from Codecov's database, utilized GitHub's public API, and distributed a survey among open-source developers. In this chapter, we will conclude our findings, highlighting their significance and implications for developers and researchers.

Our work stands out from existing work by exploring a large group of highly varied projects, in terms of size, language and domain, instead of focusing on a single language. Furthermore, the participants in our survey belong to different projects and companies, which provides a wide variety of opinions and perspectives. In turn, we believe this makes our work generalisable to a larger group of open-source development efforts.

Our research went as follows. Firstly, in Chapter 3 we aimed to answer **RQ1**: "Is there a relationship between code coverage and the time to merge a pull request?" And in our research, we have found multiple projects for which coverage-related metrics, such as hits and misses, do have a significant relationship with the time to merge using linear regression. However, we also conclude that this finding does not generalise to every project. Further research would be necessary to determine the factors that affect these relationships between code coverage and the time to merge a PR.

The analysis of patch coverage and project coverage metrics revealed that patch-related metrics were the least frequently significant. We attribute this observation to the nature of patch coverage, where the modified code is either untested (0% coverage) or fully covered (100% coverage). Partial coverage of modified code was relatively rare. These findings suggest that patch coverage metrics may not provide substantial information for the models.

Secondly, in Chapter 4 we asked **RQ2**: "To what extent do comments and status checks lead to improved coverage?" By using the odds ratio test on two groups of failing pull requests, one where the failing coverage status was fixed before merging, and one where it was not, we determined that the mentions of tests or coverage in the comments and commit messages of the pull requests are linked to greater odds of the coverage status getting fixed.

Additionally, we found a high number of PRs that were merged despite having a failing coverage status. As such, we investigated this phenomenon further in Chapter 5 where we set out to answer **RQ3**: "Why are coverage checks ignored?". For this research question we conducted a survey among developers, which led to insights into why code contributors or maintainers would disregard fixing coverage. The complexity of writing tests, the perception of insufficient value in fixing coverage, and the desire to avoid scaring away contributors were among the reasons mentioned. Moreover, our study revealed that developers predominantly rely on coverage results generated on GitHub rather than running coverage tools locally, potentially impacting the development process.

The implications of our work are twofold and made concise in Chapter 6. For open-source developers, we recommend configuring coverage tools with appropriate thresholds to make the failure meaningful and reduce noise. Improving code structure, providing clear documentation on testing, setting clear goals for testing and coverage, and discussing tests and coverage during PR reviews can also contribute to addressing the issue of ignoring coverage. For code coverage tool developers, our findings indicate the need for improvements in result accuracy, easier setup processes, and noise reduction. Additional suggestions include providing actionable information, assessing test quality, and enhancing user workflow.

Future research could focus on addressing the limitations identified by our survey, such as improving the robustness and usability of coverage tools, exploring advanced techniques for detecting flaky tests, and further investigating the factors influencing code coverage adoption and perception. Furthermore, we suggest a larger investigation on the effects of code coverage on different project metrics or PR metrics, in a study with more available time and data. Additionally, we introduce the idea of a weighted patch coverage, which augments the existing patch coverage metric with how important the changed code is deemed. Lastly, our survey results suggest that further investigation in developers' perception on code coverage, and code coverage tools, is needed. Specifically, on determining whether the use of code coverage (tools) provides a greater incentive to write tests, and whether those tests would be of high quality.

In conclusion, our study has shed light on the relationship between code coverage and time to merge, offering valuable insights into the impact of mentioning code coverage in a pull request, and the underlying reasons for disregarding coverage checks. The implications of our findings extend not only to developers seeking to optimise their software development processes, but also to code coverage tool developers striving to enhance their tools' effectiveness. Furthermore, our study has generated a wealth of ideas for future research, providing a solid foundation for further exploration in this field. Ultimately, our work contributes to the ongoing pursuit of more efficient and reliable software development practices, fostering innovation and progress in the ever-evolving landscape of open-source software development.

Bibliography

[1] Codecov api. URL https://docs.codecov.com/reference/overview.

[2] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their ides. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 179–190, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336758. doi: 10.1145/2786805.2786843. URL https://doi.org/10.1145/2786805.2786843.

[3] Casper Boone, Carolin Brandt, and Andy Zaidman. Fixing continuous integration tests from within the ide with contextual information. In *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*, pages 287–297, 2022. doi: 10.1145/3524610.3527908.

[4] Codecov. The leading code coverage solution, May 2022. URL https://about.codecov.io/.

[5] Anna Derezińska. Experiences from an empirical study of programs code coverage. In Tarek Sobh, editor, *Advances in Computer and Information Sciences and Engineering*, pages 57–62, Dordrecht, 2008. Springer Netherlands. ISBN 978-1-4020-8741-7.

[6] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 345–355, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327565. doi: 10.1145/2568225.2568260. URL https://doi.org/10.1145/2568225.2568260.

[7] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. Lean ghtorrent: Github data on demand. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 384–387, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328630. doi: 10.1145/2597073.2597126. URL https://doi.org/10.1145/2597073.2597126.

[8] H. Hemmati. How effective are code coverage criteria? In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 151–156, 2015. doi: 10.1109/QRS.2015.30.

[9] Michael Hilton, Jonathan Bell, and Darko Marinov. A large-scale study of test coverage evolution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 53–63, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359375. doi: 10.1145/3238147.3238183. URL `https://doi.org/10.1145/3238147.3238183`.

[10] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In *Proceedings of 16th International Conference on Software Engineering*, pages 191–200, 1994. doi: 10.1109/ICSE.1994.296778.

[11] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 435–445, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327565. doi: 10.1145/2568225.2568271. URL `https://doi.org/10.1145/2568225.2568271`.

[12] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. Code coverage at google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 955–963, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450355728. doi: 10.1145/3338906.3340459. URL `https://doi.org/10.1145/3338906.3340459`.

[13] Yong Woo Kim. Efficient use of code coverage in large-scale software development. In *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '03, page 145–155. IBM Press, 2003.

[14] P. S. Kochhar, F. Thung, D. Lo, and J. Lawall. An empirical study on the adequacy of testing in open source projects. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 215–222, 2014. doi: 10.1109/APSEC.2014.42.

[15] 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*, 66(4):1213–1228, 2017. doi: 10.1109/TR.2017.2727062.

[16] Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. Power laws in software. *ACM Transactions on Software Engineering and Methodology*, 18 (1):1–26, September 2008. ISSN 1049-331X. doi: 10.1145/1391984.1391986. URL `http://www.dmst.aueb.gr/dds/pubs/jrnl/2008-TOSEM-PowerLaws/html/LSV08.html`. Article 2.

[17] Chandra Maddila, Chetan Bansal, and Nachiappan Nagappan. Predicting pull request completion time: A case study on large scale cloud services. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 874–882, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450355728. doi: 10.1145/3338906.3340457. URL https://doi.org/10.1145/3338906.3340457.

[18] Shane Mcintosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Softw. Engg.*, 21(5):2146–2189, October 2016. ISSN 1382-3256. doi: 10.1007/s10664-015-9381-9. URL https://doi.org/10.1007/s10664-015-9381-9.

[19] D. Mondal, H. Hemmati, and S. Durocher. Exploring test suite diversification and code coverage in multi-objective test case selection. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, 2015. doi: 10.1109/ICST.2015.7102588.

[20] Akbar Siami Namin and James H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, page 57–68, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583389. doi: 10.1145/1572272.1572280. URL https://doi.org/10.1145/1572272.1572280.

[21] Saeed Parsa. *Software Testability*, pages 3–43. Springer International Publishing, Cham, 2023. ISBN 978-3-031-22057-9. doi: 10.1007/978-3-031-22057-9_1. URL https://doi.org/10.1007/978-3-031-22057-9_1.

[22] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450316149. doi: 10.1145/2393596.2393634. URL https://doi.org/10.1145/2393596.2393634.

[23] Amanda Schwartz, Daniel Puckett, Ying Meng, and Gregory Gay. Investigating faults missed by test suites achieving high code coverage. *Journal of Systems and Software*, 144:106 – 120, 2018. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2018.06.024. URL http://www.sciencedirect.com/science/article/pii/S0164121218301201.

[24] Davide Spadini, Maurício Aniche, Margaret-Anne Storey, Magiel Bruntink, and Alberto Bacchelli. When testing meets code review: Why and how developers review tests. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 677–687, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356381. doi: 10.1145/3180155.3180192. URL https://doi.org/10.1145/3180155.3180192.

[25] Igor Steinmacher, Gustavo Pinto, Igor Scaliante Wiese, and Marco A. Gerosa. Almost there: A study on quasi-contributors in open source software projects. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 256–266, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356381. doi: 10.1145/3180155.3180208. URL `https://doi.org/10.1145/3180155.3180208`.

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

[27] Mairieli Wessel and Igor Steinmacher. *The Inconvenient Side of Software Bots on Pull Requests*, page 51–55. Association for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450379632. URL `https://doi.org/10.1145/3387940.3391504`.

[28] Mairieli Wessel, Alexander Serebrenik, Igor Scaliante Wiese, Igor Steinmacher, and Marco Aurelio Gerosa. Effects of adopting code review bots on pull requests to oss projects. In *Proceedings - 2020 IEEE International Conference on Software Maintenance and Evolution, ICSME 2020*, pages 1–11, United States, September 2020. IEEE Computer Society. doi: 10.1109/ICSME46990.2020.00011.

[29] Mairieli Wessel, Alexander Serebrenik, Igor Wiese, Igor Steinmacher, and Marco A. Gerosa. What to expect from code review bots on github? a survey with oss maintainers. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*, SBES '20, page 457–462, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450387538. doi: 10.1145/3422392.3422459. URL `https://doi.org/10.1145/3422392.3422459`.

[30] Yue Yu, Bogdan Vasilescu, Huaimin Wang, Vladimir Filkov, and Premkumar T. Devanbu. Initial and eventual software quality relating to continuous integration in github. *CoRR*, abs/1606.00521, 2016. URL `http://arxiv.org/abs/1606.00521`.

[31] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997. ISSN 0360-0300. doi: 10.1145/267580.267590. URL `https://doi.org/10.1145/267580.267590`.

Appendix A

List of projects

This appendix shows the projects that were included in our dataset for research questions one and two. As you can see, the data was updated around April 2021. As stated in Chapter 3, only projects with over 100 pull requests were considered relevant. Furthermore, from those projects, only those with an R-Squared value over 0.7 were subsequently discussed in the results.

You will also see that some projects do not have pull requests listed at all. This is because there was missing data in either the Codecov database, or the GHTorrent dump.

For each project, we calculated a regression model, and the full models can be found on GitHub: `https://doi.org/10.5281/zenodo.8044949`

A. List of projects

project	contributors	watchers	stargazerCount	forkCount	primaryLanguage	codecov_updated	pull_requests	R-squared
https://github,com/getsentry/sentry	584	669	27878	3121	Python	26/04/2021	5744	0,076348167
https://github,com/pandas-dev/pandas	2662	1114	29472	12274	Python	26/04/2021	2098	0,27514342
https://github,com/python/cpython	1662	1403	37529	18573	Python	26/04/2021	1427	0,526560006
https://github,com/home-assistant/core	2705	1335	42108	13860	Python	26/04/2021	1417	0,816151843
https://github,com/scikit-learn/scikit-learn	2208	2254	45537	21370	Python	26/04/2021	1118	0,765685998
https://github,com/edx/edx-platform	996	417	5524	3030	Python	26/04/2021	1035	0,685759153
https://github,com/matplotlib/matplotlib	1224	571	13533	5769	Python	26/04/2021	980	0,594803016
https://github,com/PyTorchLightning/pytorch-lightning	435	213	13178	1560	Python	26/04/2021	790	0,337221512
https://github,com/pingcap/tidb	579	1361	27582	4332	Go	26/04/2021	747	0,756251651
https://github,com/pytest-dev/pytest	709	200	7246	1686	Python	26/04/2021	734	0,202016506
https://github,com/numpy/numpy	1311	566	16969	5444	Python	26/04/2021	727	0,346231736
https://github,com/Homebrew/brew	968	690	27730	6163	Ruby	26/04/2021	684	0,443831828
https://github,com/allenai/allennlp	240	293	9943	2047	Python	26/04/2021	632	0,224312075
https://github,com/ampproject/amphtml	1106	683	14600	3800	JavaScript	26/04/2021	597	0,219178671
https://github,com/grpc/grpc-java	247	551	8627	2908	Java	23/04/2021	594	0,247441727
https://github,com/youzan/vant	194	371	17240	8810	TypeScript	26/04/2021	593	0,426960213
https://github,com/go-gitea/gitea	958	482	24562	3010	Go	26/04/2021	578	0,616701086
https://github,com/prettier/prettier	552	399	39498	2766	JavaScript	26/04/2021	572	0,600003956
https://github,com/GoogleContainerTools/skaffold	270	208	11191	1173	Go	24/04/2021	548	0,636274214
https://github,com/bootstrap-vue/bootstrap-vue	323	307	13083	1745	JavaScript	21/04/2021	510	0,221148021
https://github,com/woocommerce/woocommerce	1324	563	6953	8954	PHP	26/04/2021	505	0,190857449
https://github,com/GoogleChrome/lighthouse	282	543	22246	7787	JavaScript	24/04/2021	488	0,451664024
https://github,com/NG-ZORRO/ng-zorro-antd	174	249	7448	2638	TypeScript	26/04/2021	475	0,211459642
https://github,com/webpack/webpack	749	1582	58065	7710	JavaScript	26/04/2021	467	0,352862126
https://github,com/nextcloud/server	940	544	14071	2521	PHP	26/04/2021	463	0,368093123
https://github,com/mirumee/saleor	201	364	10921	3624	Python	26/04/2021	441	0,54724127
https://github,com/apache/airflow	1846	744	21264	8383	Python	26/04/2021	420	0,425495413
https://github,com/vuetifyjs/vuetify	670	605	30278	5560	TypeScript	24/04/2021	418	0,541048998
https://github,com/zulip/zulip	800	378	13421	4477	Python	25/04/2021	411	0,229560732
https://github,com/dotnet/machinelearning	163	621	7477	1622	C#	20/04/2021	382	0,492259093
https://github,com/milvus-io/milvus	134	159	5566	862	Go	26/04/2021	381	0,354570235
https://github,com/ipfs/go-ipfs	307	561	11272	2137	Go	26/04/2021	374	0,60889926
https://github,com/babel/babel	1022	846	38864	4815	JavaScript	26/04/2021	362	0,765843799
https://github,com/nuxt/nuxt,js	309	801	35848	2907	JavaScript	26/04/2021	344	0,52668696
https://github,com/typescript-eslint/typescript-eslint	342	72	9425	1484	TypeScript	26/04/2021	340	0,533669542
https://github,com/hashicorp/terraform	1850	1154	26446	6560	Go	26/04/2021	336	0,624237812
https://github,com/apache/shardingsphere	286	984	13712	4665	Java	26/04/2021	327	0,131170844
https://github,com/hashicorp/packer	1396	501	12781	3128	Go	23/04/2021	323	0,366907605
https://github,com/cakephp/cakephp	754	602	8361	3466	PHP	24/04/2021	317	0,275707154
https://github,com/facebook/jest	1298	540	34951	5050	TypeScript	26/04/2021	307	0,373333749
https://github,com/SpaceVim/SpaceVim	272	341	16528	1316	Vim script	25/04/2021	296	0,300093078
https://github,com/facebookresearch/ParlAI	147	271	7137	1490	Python	26/04/2021	292	0,596177927
https://github,com/storybookjs/storybook	1479	913	60774	6036	TypeScript	26/04/2021	288	0,362125695
https://github,com/seata/seata	209	883	19299	6003	Java	26/04/2021	284	0,614965875
https://github,com/hashicorp/consul	772	991	21954	3689	Go	23/04/2021	272	0,774178506
https://github,com/python-pillow/Pillow	357	215	8459	1623	Python	26/04/2021	272	0,665812011
https://github,com/saltstack/salt	3679	578	11694	5084	Python	22/04/2021	263	0,651241506
https://github,com/apache/skywalking	368	841	16689	4906	Java	26/04/2021	254	0,645379601
https://github,com/argoproj/argo-cd	414	113	5821	1277	Go	25/04/2021	240	0,636945025
https://github,com/pytorch/vision	381	317	8927	4603	Python	26/04/2021	235	0,454660523
https://github,com/ipython/ipython	807	799	14780	4151	Python	22/04/2021	229	0,611375817
https://github,com/ReactiveX/RxJava	313	2310	44515	7407	Java	26/04/2021	225	0,215982489
https://github,com/syncthing/syncthing	266	982	36152	2988	Go	26/04/2021	223	0,889421783
https://github,com/timescale/timescaledb	51	290	10788	574	C	26/04/2021	217	0,254323181
https://github,com/aws/aws-cli	337	569	10931	2602	Python	23/04/2021	203	0,178060623
https://github,com/marko-js/marko	126	230	10211	611	JavaScript	21/04/2021	189	0,293463497
https://github,com/getsentry/sentry-javascript	352	100	5421	1011	TypeScript	26/04/2021	180	0,507964204
https://github,com/goreleaser/goreleaser	216	104	7933	535	Go	25/04/2021	180	0,606603656
https://github,com/cupy/cupy	260	118	5065	465	Python	26/04/2021	178	0,654591334
https://github,com/jaegertracing/jaeger	207	336	13318	1565	Go	26/04/2021	175	0,728598426
https://github,com/serverless/serverless	1003	988	39543	4701	JavaScript	24/04/2021	154	0,927060372
https://github,com/akveo/nebular	107	195	7022	1345	TypeScript	22/04/2021	153	0,521175531
https://github,com/iamkun/dayjs	254	295	34702	1646	JavaScript	25/04/2021	150	0,347423657
https://github,com/parse-community/parse-server	301	629	18548	4480	JavaScript	25/04/2021	149	0,719566905
https://github,com/Semantic-Org/Semantic-UI-React	307	231	12214	3608	JavaScript	23/04/2021	146	0,846693809
https://github,com/phpmyadmin/phpmyadmin	1823	291	5435	2927	PHP	26/04/2021	144	0,299254294
https://github,com/borgbackup/borg	234	155	7211	554	C	20/04/2021	139	0,678787767
https://github,com/graphql/graphql-js	179	410	17563	1794	JavaScript	25/04/2021	137	0,939049867
https://github,com/coredns/coredns	256	226	7514	1311	Go	26/04/2021	136	0,569054538
https://github,com/gin-gonic/gin	315	1351	47551	5421	Go	25/04/2021	136	0,899004198
https://github,com/apereo/cas	301	621	8588	3501	Java	26/04/2021	135	0,455080592
https://github,com/scrapy/scrapy	474	1833	40435	9154	Python	25/04/2021	134	0,72227861
https://github,com/open-mmlab/mmdetection	210	355	14622	5066	Python	26/04/2021	130	0,953426365
https://github,com/celery/celery	1110	490	17133	3981	Python	26/04/2021	128	0,418546604
https://github,com/dmlc/xgboost	513	956	20912	7980	C++	23/04/2021	122	0,753350298
https://github,com/aio-libs/aiohttp	574	237	11106	1581	Python	26/04/2021	119	0,493990114
https://github,com/pymc-devs/pymc3	312	252	5707	1379	Python	26/04/2021	119	0,308328991
https://github,com/valor-software/ngx-bootstrap	321	224	5283	1657	TypeScript	19/04/2021	119	0,74955351
https://github,com/wagtail/wagtail	564	354	10518	2211	Python	26/04/2021	119	0,472951607
https://github,com/aws-amplify/amplify-js	337	199	7996	1642	TypeScript	26/04/2021	118	0,637112818
https://github,com/goharbor/harbor	250	518	14583	3670	Go	26/04/2021	118	0,438554641
https://github,com/webpack/webpack-dev-server	243	123	6869	1228	JavaScript	26/04/2021	115	0,383781575
https://github,com/doctrine/dbal	515	116	8005	1127	PHP	23/04/2021	114	0,811051933
https://github,com/briannesbitt/Carbon	270	233	15165	1162	PHP	21/04/2021	113	0,348492864
https://github,com/tiangolo/fastapi	214	500	30177	2085	Python	24/04/2021	111	0,439987009
https://github,com/go-swagger/go-swagger	277	122	6334	985	Go	25/04/2021	110	0,36389871
https://github,com/falconry/falcon	172	283	8353	821	Python	25/04/2021	108	0,475067679
https://github,com/sebastianbergmann/phpunit	464	350	17527	1996	PHP	24/04/2021	105	0,866719067
https://github,com/rollup/rollup	274	265	20079	1000	JavaScript	25/04/2021	104	0,522270618
https://github,com/aframevr/aframe	365	535	12678	3044	JavaScript	25/04/2021	102	0,457549778
https://github,com/umijs/umi	294	261	10390	1623	TypeScript	26/04/2021	102	0,48324116
https://github,com/pion/webrtc	143	240	7028	859	Go	25/04/2021	101	0,212205711
https://github,com/swoole/swoole-src	181	883	16801	3139	C++	26/04/2021	101	0,523552787
https://github,com/ajaxorg/ace	475	645	22951	4934	JavaScript	23/04/2021	100	0,520828888
https://github,com/keras-team/autokeras	129	301	7916	1278	Python	16/04/2021	99	0,573395526
https://github,com/kubernetes/dashboard	278	281	9548	2877	Go	26/04/2021	99	0,568701529
https://github,com/mitmproxy/mitmproxy	391	597	22179	2846	Python	25/04/2021	98	0,232273765
https://github,com/Hacker0x01/react-datepicker	338	78	5673	1702	JavaScript	26/04/2021	96	0,417018604
https://github,com/pyecharts/pyecharts	30	381	10872	2430	Python	08/02/2021	96	0,392490194
https://github,com/SwifterSwift/SwifterSwift	140	283	9680	1256	Swift	19/04/2021	95	0,467062823
https://github,com/apollographql/apollo-server	441	219	11329	1705	TypeScript	25/04/2021	95	0,772114457
https://github,com/buefy/buefy	306	113	8593	988	Vue	22/04/2021	95	0,808289634
https://github,com/dapr/dapr	115	408	12869	841	Go	25/04/2021	95	0,763636139
https://github,com/metabase/metabase	275	632	24668	3289	Clojure	23/04/2021	95	0,823348242
https://github,com/uutils/coreutils	266	175	8316	649	Rust	26/04/2021	94	0,328581932
https://github,com/emotion-js/emotion	256	95	12894	819	JavaScript	26/04/2021	93	0,926791976
https://github,com/beetbox/beets	448	415	10062	1637	Python	19/04/2021	91	0,853250914

project	contributors	watchers	stargazerCount	forkCount	primaryLanguage	codecov_updated	pull_requests	R-squared
https://github,com/python-telegram-bot/python-telegram-bot	153	593	14508	3085	Python	25/04/2021	91	0,750139183
https://github,com/encode/django-rest-framework	1205	632	20814	5653	Python	26/04/2021	90	0,624782538
https://github,com/mwaskom/seaborn	152	240	8344	1406	Python	24/04/2021	90	0,421819955
https://github,com/scipy/scipy	1150	329	8164	3656	Python	26/04/2021	89	0,969248667
https://github,com/etcd-io/etcd	743	1369	35655	7607	Go	26/04/2021	87	0,302469191
https://github,com/iterative/dvc	219	122	7771	744	Python	26/04/2021	87	0,455379342
https://github,com/styleguidist/react-styleguidist	240	108	9729	1372	TypeScript	21/04/2021	86	0,60993502
https://github,com/ctripcorp/apollo	99	1316	24426	8927	Java	24/04/2021	85	0,644767417
https://github,com/grafana/loki	357	259	12787	1421	Go	21/04/2021	82	0,632902082
https://github,com/labstack/echo	215	541	19693	1752	Go	25/04/2021	82	0,413784341
https://github,com/jupyter/notebook	552	318	8197	3380	JavaScript	25/04/2021	81	0,469213615
https://github,com/react-bootstrap/react-bootstrap	428	458	19293	3088	JavaScript	26/04/2021	81	0,354803132
https://github,com/mockito/mockito	247	434	11744	2087	Java	23/04/2021	80	0,765148144
https://github,com/react-navigation/react-navigation	102	338	19637	4242	TypeScript	24/04/2021	80	0,38660857
https://github,com/avajs/ava	279	239	18944	1345	JavaScript	26/04/2021	75	0,775318593
https://github,com/aws/aws-sdk-js	174	260	6553	1327	JavaScript	23/04/2021	73	0,848896937
https://github,com/qutebrowser/qutebrowser	393	191	6774	861	Python	26/04/2021	73	0,700224695
https://github,com/spotify/luigi	556	502	14476	2248	Python	22/04/2021	73	0,514004653
https://github,com/VowpalWabbit/vowpal_wabbit	298	377	7527	1733	C++	23/04/2021	72	0,76325468
https://github,com/google/go-github	408	220	7391	1506	Go	24/04/2021	71	0,745122702
https://github,com/apache/dubbo	414	3222	35103	23448	Java	26/04/2021	70	0,869185673
https://github,com/aws/aws-sdk-php	169	241	5168	1005	PHP	23/04/2021	70	0,878415513
https://github,com/ng-bootstrap/ng-bootstrap	153	271	7738	1422	TypeScript	21/04/2021	69	0,817405378
https://github,com/anuraghazra/github-readme-stats	139	155	22720	4702	JavaScript	21/04/2021	62	0,774634878
https://github,com/catchorg/Catch2	285	457	13324	2254	C++	16/04/2021	62	0,591493739
https://github,com/rq/rq	234	219	7663	1224	Python	20/04/2021	59	0,451323859
https://github,com/moment/luxon	136	102	11044	492	JavaScript	09/04/2021	58	0,516401006
https://github,com/react-native-elements/react-native-elements	329	382	20501	4164	TypeScript	25/04/2021	57	0,947290235
https://github,com/SDWebImage/SDWebImage	304	832	23500	5702	Objective-C	25/04/2021	56	0,816535754
https://github,com/felangel/bloc	114	148	6968	1821	Dart	26/04/2021	56	0,367928778
https://github,com/kubernetes/ingress-nginx	664	276	10103	4962	Go	24/04/2021	56	0,563542169
https://github,com/tqdm/tqdm	98	199	18134	931	Python	23/04/2021	52	0,91137234
https://github,com/KaTeX/KaTeX	133	289	13572	941	JavaScript	26/04/2021	51	0,863968784
https://github,com/alibaba/fastjson	187	1373	23287	6248	Java	25/04/2021	51	0,719101386
https://github,com/prompt-toolkit/python-prompt-toolkit	177	154	7006	546	Python	09/04/2021	50	0,686481211
https://github,com/RichardKnop/machinery	126	150	5136	650	Go	13/04/2021	49	0,931062573
https://github,com/grpc-ecosystem/grpc-gateway	242	284	10654	1394	Go	23/04/2021	49	0,820751306
https://github,com/apache/superset	607	1446	38201	7242	Python	26/04/2021	47	0,89674382
https://github,com/polybar/polybar	123	125	8458	445	C++	25/04/2021	47	0,886202278
https://github,com/eggjs/egg	184	487	16924	1671	JavaScript	25/04/2021	45	0,872992491
https://github,com/mkdocs/mkdocs	191	230	12020	1731	Python	25/04/2021	43	0,946816707
https://github,com/actix/actix-web	292	215	11120	1210	Rust	22/04/2021	42	0,414156475
https://github,com/googleapis/google-api-nodejs-client	142	359	9272	1676	TypeScript	26/04/2021	41	0,981610347
https://github,com/backstage/backstage	340	174	11363	1112	TypeScript	26/04/2021	40	0,961540472
https://github,com/validatorjs/validator,js	372	229	17486	1625	JavaScript	26/04/2021	40	0,902976474
https://github,com/pytorch/pytorch	2631	1594	47814	12778	C++	26/04/2021	36	0,920161597
https://github,com/c3js/c3	169	287	9054	1454	JavaScript	16/04/2021	34	0,428364933
https://github,com/jazzband/django-debug-toolbar	235	114	6448	918	Python	23/04/2021	33	0,903575411
https://github,com/OpenZeppelin/openzeppelin-contracts	262	495	10085	4582	JavaScript	26/04/2021	32	0,947603784
https://github,com/RustPython/RustPython	198	130	7592	503	Python	23/04/2021	32	0,953376959
https://github,com/pinpoint-apm/pinpoint	124	768	11347	3402	Java	23/04/2021	32	0,952656846
https://github,com/doctrine/orm	863	287	8517	2315	PHP	25/04/2021	30	0,61953367
https://github,com/CocoaLumberjack/CocoaLumberjack	201	399	12352	2134	Objective-C	09/04/2021	28	0,747345138
https://github,com/cookiecutter/cookiecutter	249	250	14341	1450	Python	22/04/2021	28	0,940170827
https://github,com/sebastianbergmann/php-code-coverage	103	57	7767	322	PHP	23/04/2021	28	0,677815766
https://github,com/compiler-explorer/compiler-explorer	221	231	8472	979	JavaScript	25/04/2021	27	0,770112749
https://github,com/invertase/react-native-firebase	362	180	8996	1768	JavaScript	24/04/2021	27	0,970343284
https://github,com/zxing/zxing	151	1710	27623	8985	Java	12/04/2021	27	0,978901742
https://github,com/trekhleb/javascript-algorithms	142	4056	104207	17322	JavaScript	21/04/2021	25	0,954147246
https://github,com/springfox/springfox	208	281	5116	1415	Java	26/04/2021	24	0,999963671
https://github,com/uber-go/zap	81	249	12449	937	Go	20/04/2021	24	0,985482882
https://github,com/arrow-py/arrow	239	133	7367	568	Python	25/04/2021	22	0,937679781
https://github,com/graphql/graphiql	187	218	12039	1273	TypeScript	19/04/2021	21	1
https://github,com/mui-org/material-ui	2406	1413	67785	21596	JavaScript	26/04/2021	20	0,997885415
https://github,com/networkx/networkx	508	286	8980	2296	Python	26/04/2021	20	1
https://github,com/urfave/cli	227	300	15691	1374	Go	24/04/2021	20	1
https://github,com/botman/botman	111	196	5407	724	PHP	14/04/2021	19	
https://github,com/faif/python-patterns	116	1661	28163	5820	Python	25/01/2021	19	
https://github,com/mxcl/PromiseKit	153	269	13338	1363	Swift	02/04/2021	19	
https://github,com/the-control-group/voyager	385	444	10386	2486	PHP	22/04/2021	19	
https://github,com/ycm-core/YouCompleteMe	173	591	22793	2647	Python	21/04/2021	19	
https://github,com/dbcli/pgcli	155	141	9525	438	Python	12/03/2021	18	
https://github,com/alibaba/canal	158	1183	18663	5719	Java	23/04/2021	17	
https://github,com/graphite-project/graphite-web	425	285	5219	1252	JavaScript	19/04/2021	17	
https://github,com/koajs/koa	208	876	31089	2997	JavaScript	12/04/2021	17	
https://github,com/nodejs/node	3320	2961	78645	19957	JavaScript	26/04/2021	17	
https://github,com/rusty1s/pytorch_geometric	158	239	10845	1872	Python	25/04/2021	17	
https://github,com/aquasecurity/trivy	103	98	7074	615	Go	22/04/2021	14	
https://github,com/prisma/prisma	125	111	10573	414	TypeScript	26/04/2021	14	
https://github,com/shelljs/shelljs	85	174	11852	673	JavaScript	15/04/2021	14	
https://github,com/aws/serverless-application-model	219	327	7864	2013	Python	23/04/2021	13	
https://github,com/postmanlabs/newman	106	152	5182	875	JavaScript	26/04/2021	13	
https://github,com/SBoudrias/Inquirer,js	164	150	14215	935	JavaScript	08/04/2021	12	
https://github,com/apache/openwhisk	216	242	5246	1010	Scala	26/04/2021	12	
https://github,com/ossrs/srs	60	739	11745	3709	C++	26/04/2021	11	
https://github,com/Tonejs/Tone,js	78	217	10840	828	TypeScript	25/04/2021	10	
https://github,com/bvaughn/react-virtualized	210	261	21569	2660	JavaScript	24/04/2021	10	
https://github,com/ustbhuangyi/better-scroll	47	264	14303	2473	TypeScript	21/04/2021	10	
https://github,com/vueComponent/ant-design-vue	137	305	14200	2367	Vue	25/04/2021	10	
https://github,com/brettwooldridge/HikariCP	122	733	14903	2264	Java	06/04/2021	9	
https://github,com/davatorium/rofi	116	85	7304	377	C	22/04/2021	9	
https://github,com/downshift-js/downshift	195	92	9508	780	JavaScript	20/04/2021	9	
https://github,com/phalcon/cphalcon	322	684	10426	1913	PHP	25/04/2021	9	
https://github,com/slundberg/shap	156	238	12431	1842	Jupyter Notebook	20/04/2021	9	
https://github,com/typeorm/typeorm	696	342	23967	4143	TypeScript	26/04/2021	9	
https://github,com/apache/druid	471	632	10789	2905	Java	26/04/2021	8	
https://github,com/nfl/react-helmet	64	129	15022	618	JavaScript	30/03/2021	7	
https://github,com/Hammerspoon/hammerspoon	127	110	7802	412	Objective-C	25/04/2021	6	
https://github,com/beego/beego	433	1242	26265	5175	Go	26/04/2021	6	
https://github,com/doctrine/common	213	51	5150	294	PHP	24/04/2021	6	
https://github,com/jorgebucaran/hyperapp	102	334	18494	813	JavaScript	24/04/2021	6	
https://github,com/reduxjs/react-redux	243	445	20869	3004	JavaScript	24/04/2021	6	
https://github,com/sanic-org/sanic	293	431	14887	1342	Python	22/04/2021	6	
https://github,com/JuliaLang/julia	1307	993	33438	4406	Julia	25/04/2021	5	
https://github,com/PostgREST/postgrest	95	357	16909	779	Haskell	26/04/2021	5	
https://github,com/alibaba/spring-cloud-alibaba	109	963	18427	5634	Java	26/04/2021	5	
https://github,com/ansible/ansible	6382	2016	47870	20531	Python	26/04/2021	5	

A. List of projects

project	contributors	watchers	stargazerCount	forkCount	primaryLanguage	codecov_updated	pull_requests	R-squared
https://github,com/apache/rocketmq	276	867	14120	7766	Java	23/04/2021	5	
https://github,com/josdejong/mathjs	147	233	11070	981	JavaScript	26/04/2021	5	
https://github,com/liriliri/eruda	13	285	10190	820	JavaScript	11/03/2021	5	
https://github,com/tailwindlabs/tailwindcss	176	528	40668	1845	CSS	23/04/2021	5	
https://github,com/willmcgugan/rich	87	499	25429	759	Python	25/04/2021	5	
https://github,com/mozilla/nunjucks	153	142	7260	605	JavaScript	23/03/2021	4	
https://github,com/request/request	343	450	25121	3053	JavaScript	01/04/2021	4	
https://github,com/Alluxio/alluxio	1282	438	5061	2455	Java	26/04/2021	3	
https://github,com/MichMich/MagicMirror	300	676	14295	3553	JavaScript	24/04/2021	3	
https://github,com/enzymejs/enzyme	395	283	19505	2081	JavaScript	05/04/2021	3	
https://github,com/myliang/x-spreadsheet	29	244	11171	1183	JavaScript	22/04/2021	3	
https://github,com/processing/p5,js	517	492	15335	2411	JavaScript	24/04/2021	3	
https://github,com/redux-form/redux-form	373	180	12535	1663	JavaScript	19/04/2021	3	
https://github,com/robotframework/robotframework	139	470	5813	1654	Python	23/04/2021	3	
https://github,com/testing-library/react-testing-library	158	139	14349	808	JavaScript	25/04/2021	3	
https://github,com/acdlite/recompose	107	190	14716	1287	JavaScript	30/03/2021	2	
https://github,com/doctrine/inflector	78	23	10281	106	PHP	11/04/2021	2	
https://github,com/go-kit/kit	206	694	20031	2071	Go	19/04/2021	2	
https://github,com/guard/guard	185	127	5977	502	Ruby	18/09/2020	2	
https://github,com/javve/list,js	56	227	10471	894	JavaScript	26/04/2021	2	
https://github,com/koreader/koreader	183	293	8004	854	Lua	26/04/2021	2	
https://github,com/Modernizr/Modernizr	307	956	24867	3035	JavaScript	10/04/2021	1	
https://github,com/SortableJS/Vue,Draggable	38	225	14843	2258	JavaScript	01/04/2021	1	
https://github,com/andymccurdy/redis-py	248	351	9235	1950	Python	21/04/2021	1	
https://github,com/doctrine/instantiator	30	24	10069	54	PHP	18/02/2021	1	
https://github,com/dvajs/dva	101	464	15571	3069	JavaScript	16/04/2021	1	
https://github,com/pavlobu/deskreen	12	238	10405	455	TypeScript	29/03/2021	1	
https://github,com/photoprism/photoprism	71	258	12745	667	Go	26/04/2021	1	
https://github,com/ramsey/uuid	85	131	11132	415	PHP	23/04/2021	1	
https://github,com/redis/jedis	203	719	9807	3429	Java	26/04/2021	1	
https://github,com/HelloZeroNet/ZeroNet	129	850	16725	2130	JavaScript	02/03/2021	0	
https://github,com/HeroTransitions/Hero	78	385	20005	1621	Swift	11/04/2021	0	
https://github,com/LightTable/LightTable	91	453	11514	946	Clojure	08/04/2021	0	
https://github,com/MycroftAI/mycroft-core	175	299	5050	1077	Python	26/04/2021	0	
https://github,com/Tencent/ncnn	156	549	11450	2817	C++	26/04/2021	0	
https://github,com/airyland/vux	121	717	17424	3856	Vue	13/11/2020	0	
https://github,com/alibaba/ice	31	469	16260	1940	TypeScript	26/04/2021	0	
https://github,com/apache/arrow	670	331	7799	1883	C++	22/04/2021	0	
https://github,com/apache/dolphinscheduler	222	276	5672	1929	Java	26/04/2021	0	
https://github,com/argoproj/argo-workflows	383	178	8345	1477	Go	25/04/2021	0	
https://github,com/cdr/code-server	118	667	42301	3357	TypeScript	26/04/2021	0	
https://github,com/cesanta/mongoose	124	462	7083	2057	C	22/04/2021	0	
https://github,com/codecentric/spring-boot-admin	156	715	9795	2680	Java	23/04/2021	0	
https://github,com/commitizen/cz-cli	77	82	11259	443	JavaScript	17/04/2021	0	
https://github,com/conventional-changelog/conventional-changelog	114	46	5327	528	JavaScript	26/04/2021	0	
https://github,com/cssinjs/jss	123	88	6297	370	JavaScript	26/04/2021	0	
https://github,com/cube-js/cube,js	139	141	10215	920	Rust	26/04/2021	0	
https://github,com/dask/dask	444	238	8247	1266	Python	23/04/2021	0	
https://github,com/doctrine/lexer	27	22	10105	41	PHP	17/03/2021	0	
https://github,com/dromara/hutool	147	597	18760	5589	Java	25/04/2021	0	
https://github,com/dzenbot/DZNEmptyDataSet	54	262	11998	1725	Objective-C	03/02/2021	0	
https://github,com/halfrost/LeetCode-Go	29	504	17199	3037	Go	25/04/2021	0	
https://github,com/http-party/node-http-proxy	207	279	12070	1731	JavaScript	26/04/2021	0	
https://github,com/jenssegers/laravel-mongodb	158	186	5485	1234	PHP	22/04/2021	0	
https://github,com/k6io/k6	92	197	12092	608	Go	26/04/2021	0	
https://github,com/koel/koel	59	348	12502	1595	PHP	19/04/2021	0	
https://github,com/logaretm/vee-validate	294	122	8411	1009	TypeScript	25/04/2021	0	
https://github,com/magic-wormhole/magic-wormhole	51	214	12141	445	Python	20/04/2021	0	
https://github,com/nektos/act	80	84	13120	360	Go	23/04/2021	0	
https://github,com/open-falcon/falcon-plus	119	414	6478	1454	Go	07/04/2021	0	
https://github,com/php/php-src	1244	1505	30512	6631	C	25/04/2021	0	
https://github,com/pixijs/pixi,js	409	1060	32566	4292	TypeScript	24/04/2021	0	
https://github,com/probot/probot	156	113	6754	785	TypeScript	26/04/2021	0	
https://github,com/reduxjs/reselect	92	171	17867	637	JavaScript	11/03/2021	0	
https://github,com/sindresorhus/got	142	104	9327	555	TypeScript	15/04/2021	0	
https://github,com/strapi/strapi	732	632	36051	4347	JavaScript	26/04/2021	0	
https://github,com/substack/tape	101	68	5459	311	JavaScript	03/03/2021	0	
https://github,com/tikv/tikv	306	341	9198	1418	Rust	26/04/2021	0	
https://github,com/twitter/finagle	480	572	7942	1379	Scala	22/04/2021	0	
https://github,com/v2fly/v2ray-core	112	276	8649	1445	Go	26/04/2021	0	
https://github,com/vercel/vercel	167	103	7023	915	TypeScript	26/04/2021	0	
https://github,com/vim/vim	2	697	23516	3500	Vim script	25/04/2021	0	
https://github,com/yannickcr/eslint-plugin-react	439	84	7015	2076	JavaScript	25/04/2021	0	

Appendix B

Survey

This appendix includes the entire survey from Chapter 5. This PDF version of the survey is a direct export from Qualtrics, and includes the logic that was used to determine which questions were asked.

Questionnaire

Start of Block: Start

Welcome Opening Statement   Hi there, my name is Alexander and I am a student at the TU Delft, the Netherlands, currently working on my Master's thesis. And I'm inviting you to participate in a research study regarding the use of code coverage tools and how they interact with repository providers like GitHub. I am reaching out to you specifically because I found that you have made several contributions to one or multiple open-source projects that use Codecov (a code coverage tool) on Github. I was hoping to get your perspective on the use of these tools.

The purpose of this research study is **to gather personal opinions and experiences regarding the use of code coverage tools from both open-source contributors and open-source maintainers** and will take you approximately **10** minutes to complete. The data will be used for **a chapter in my Master's thesis** and may be submitted for a peer-reviewed publication.   **Informed consent**
Your participation in this study is entirely voluntary and you can withdraw at any time. You are free to omit any question.

We believe there are no known risks associated with this research study; however, as with any online-related activity, the risk of a (data) breach is always possible. To the best of our ability, your answers in this study will remain confidential. We will minimize any risks by not asking you for any personally identifiable information. Nor will any of this information be stored. Your Github username and public email address on your Github account were used to contact you, but they are not part of the survey results in any way.

The survey is entirely anonymous.

At the end of this study, the final conclusions will be published as part of my Master's thesis and may be submitted for peer-reviewed journal publication. If you do not agree with any of the conditions above, feel free to leave the survey and/or contact me at the address below! You can also contact me if you wish to stay informed of the results.

Thank you for your participation!

Alexander Sterk
a.j.h.sterk@student.tudelft.nl

End of Block: Start

Start of Block: Demographic questions

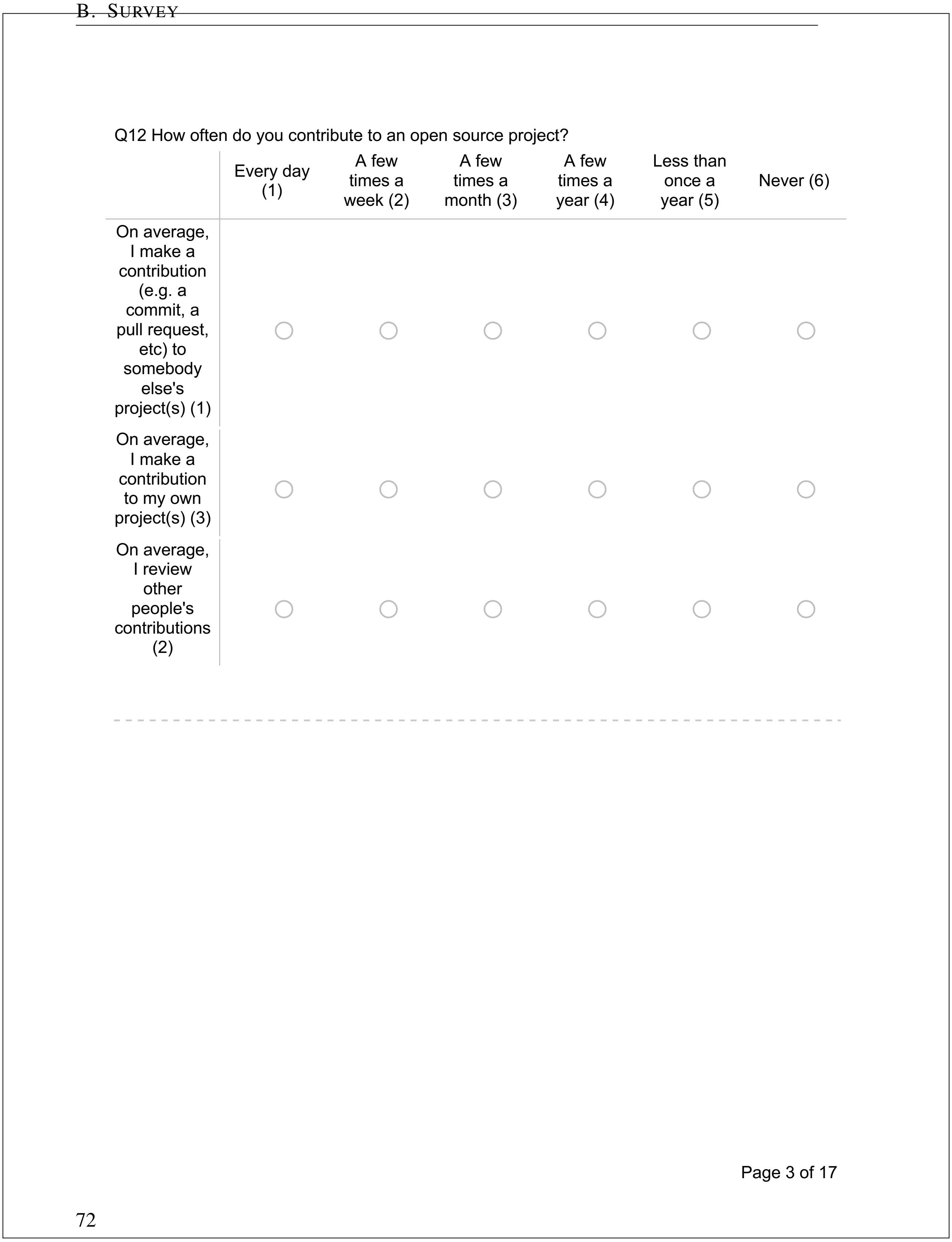
Q10 Demographic questions
In order for us to determine your experience with software development, we ask you the following general questions

Q8 For how many years have you been developing software? You can consider all hobby, study and/or work experience.

____

Q41 For how many years have you been active on open-source development platforms, such as Github, Gitlab, etc?

____



Q12 How often do you contribute to an open source project?

	Every day (1)	A few times a week (2)	A few times a month (3)	A few times a year (4)	Less than once a year (5)	Never (6)
On average, I make a contribution (e.g. a commit, a pull request, etc) to somebody else's project(s) (1)	○	○	○	○	○	○
On average, I make a contribution to my own project(s) (3)	○	○	○	○	○	○
On average, I review other people's contributions (2)	○	○	○	○	○	○

Page 3 of 17

Dev/Maintain choice
For this survey, we would like to make a distinction between code contributors and project maintainers on Github. However, we also understand there can be some overlap.
Code contributors primarily contribute to projects by opening issues or pull requests. They may or may not be part of the main development team of a project. Project maintainers primarily review other people's pull requests or issues. They are always a member or owner of a project.

Given these descriptions, please select the option that applies to you the most.

When contributing to open source projects, I primarily act as a:

○ Code contributor (1)

○ Maintainer/Project Manager (2)

Q15 Do you work in software development in a professional capacity?

○ Yes (1)

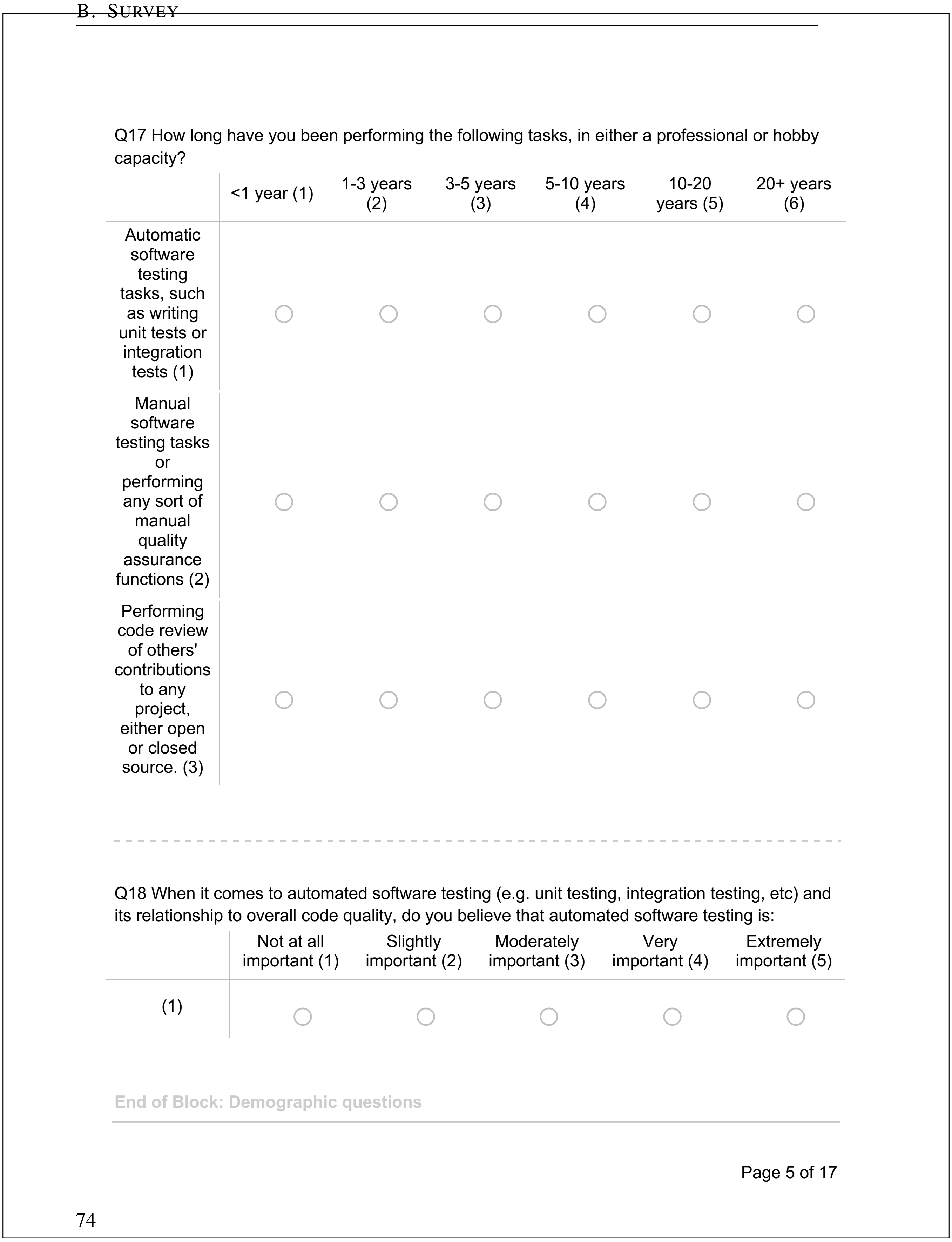
○ No (2)

Display This Question:

If Do you work in software development in a professional capacity? = Yes

Q16 What is your job title?

____



Q17 How long have you been performing the following tasks, in either a professional or hobby capacity?

	<1 year (1)	1-3 years (2)	3-5 years (3)	5-10 years (4)	10-20 years (5)	20+ years (6)
Automatic software testing tasks, such as writing unit tests or integration tests (1)	○	○	○	○	○	○
Manual software testing tasks or performing any sort of manual quality assurance functions (2)	○	○	○	○	○	○
Performing code review of others' contributions to any project, either open or closed source. (3)	○	○	○	○	○	○

Q18 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:

	Not at all important (1)	Slightly important (2)	Moderately important (3)	Very important (4)	Extremely important (5)
(1)	○	○	○	○	○

End of Block: Demographic questions

Page 5 of 17

Start of Block: Main questions

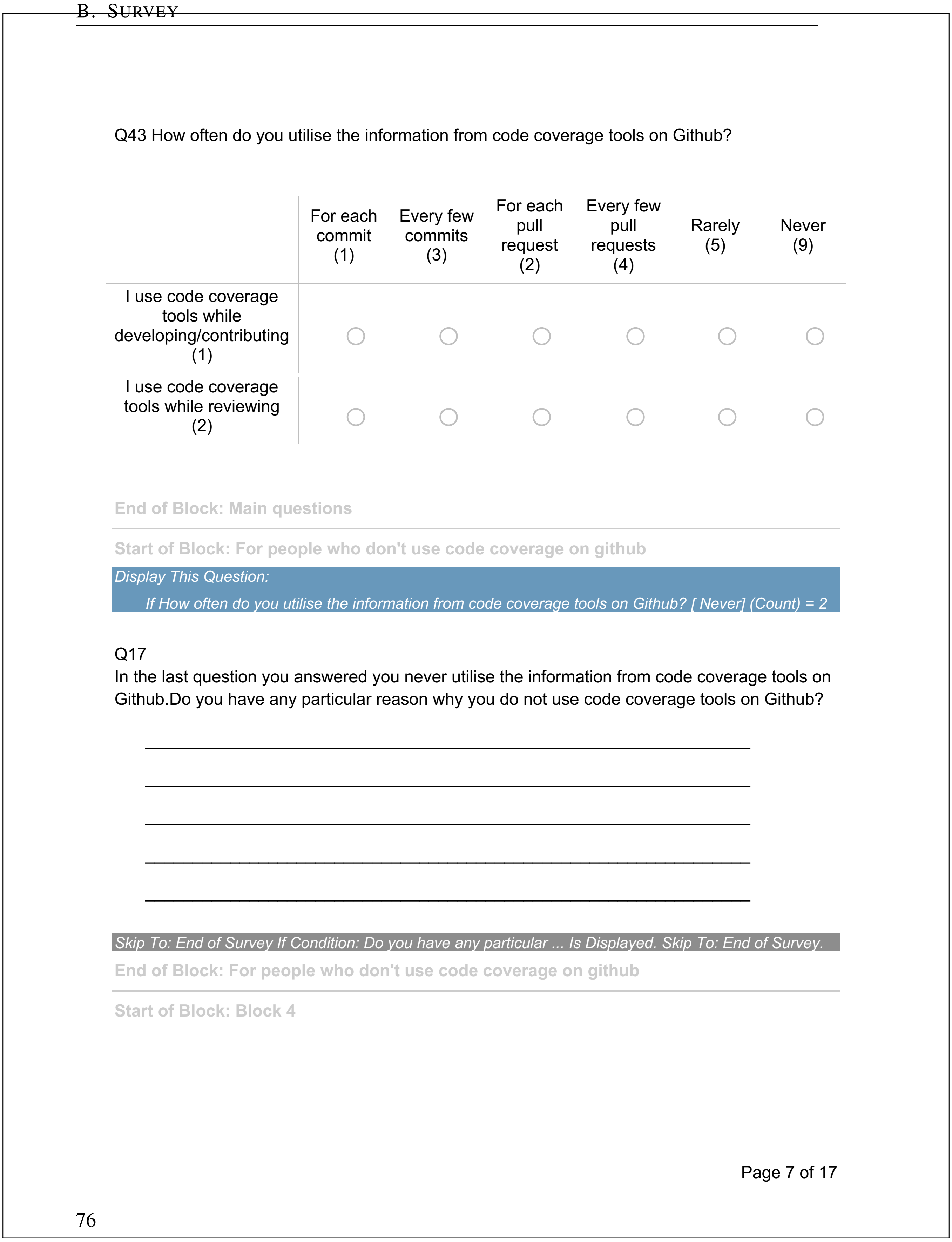
Q19 Main survey questions

Code analysis tools provide a way of measuring code quality metrics, such as code coverage, and giving feedback to developers on open source platforms directly. Examples of code quality tools that can be used on Github for code coverage are Codecov (https://about.codecov.io/), Coveralls (https://coveralls.io/) and SonarQube (https://www.sonarqube.org/). Below is an example of a Codecov comment, left on a pull request.

The following questions are all about the use of code coverage tools on open source platforms. The platform we will be focussing on is Github.

Q20 How often do you use code coverage tools outside of Github? For example, on your own machine.

	For each commit (1)	Every few commits (3)	For each pull request (2)	Every few pull requests (4)	Rarely (9)	Never (5)
I use code coverage tools (1)	○	○	○	○	○	○



Q43 How often do you utilise the information from code coverage tools on Github?

	For each commit (1)	Every few commits (3)	For each pull request (2)	Every few pull requests (4)	Rarely (5)	Never (9)
I use code coverage tools while developing/contributing (1)	○	○	○	○	○	○
I use code coverage tools while reviewing (2)	○	○	○	○	○	○

End of Block: Main questions

Start of Block: For people who don't use code coverage on github

Display This Question:

If How often do you utilise the information from code coverage tools on Github? [Never] (Count) = 2

Q17
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?

____

____

____

____

____

Skip To: End of Survey If Condition: Do you have any particular ... Is Displayed. Skip To: End of Survey.

End of Block: For people who don't use code coverage on github

Start of Block: Block 4

Page 7 of 17

Q20 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.

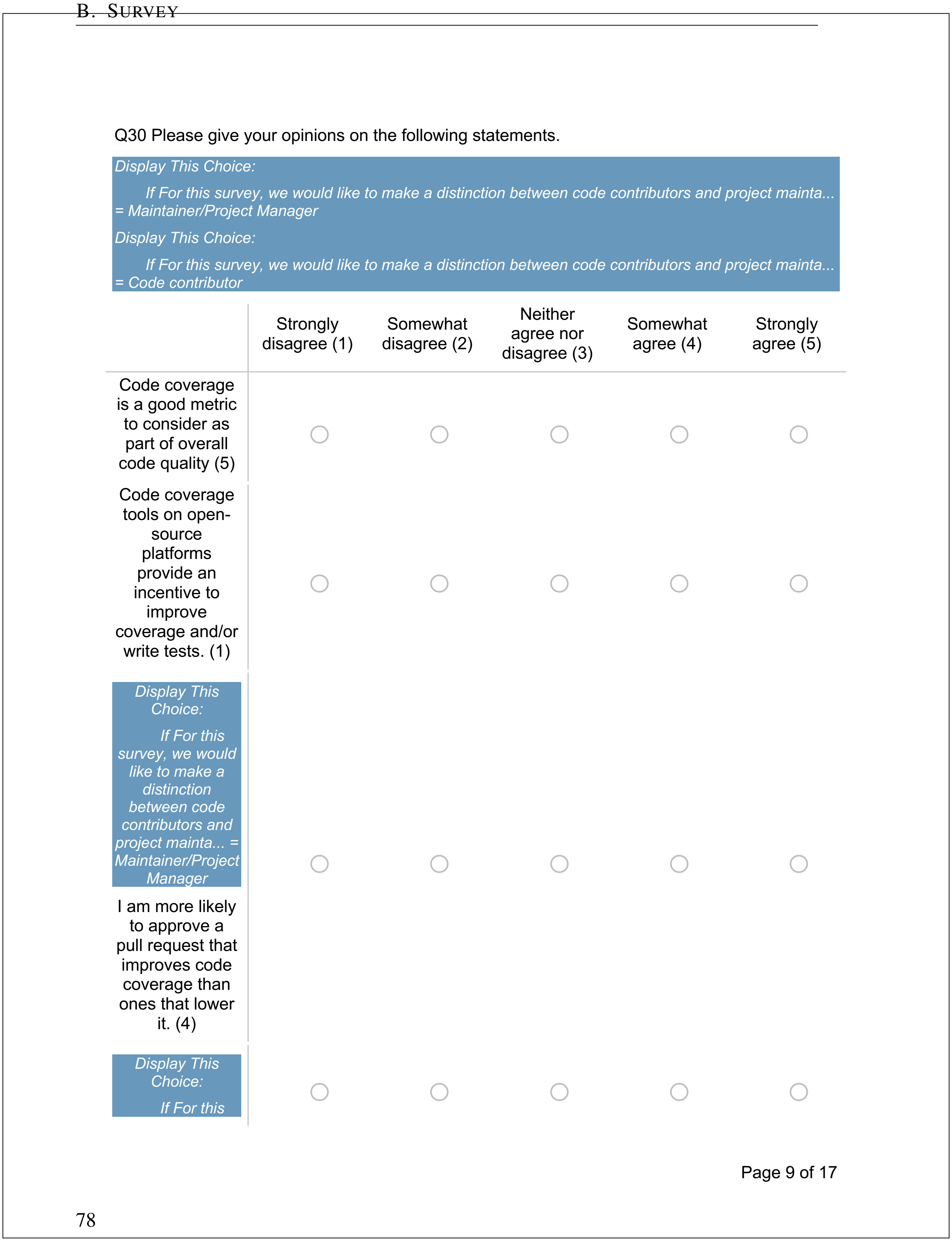
____

____

____

____

____



Q30 Please give your opinions on the following statements.

Display This Choice:

If For this survey, we would like to make a distinction between code contributors and project mainta... = Maintainer/Project Manager

Display This Choice:

If For this survey, we would like to make a distinction between code contributors and project mainta... = Code contributor

	Strongly disagree (1)	Somewhat disagree (2)	Neither agree nor disagree (3)	Somewhat agree (4)	Strongly agree (5)
Code coverage is a good metric to consider as part of overall code quality (5)	○	○	○	○	○
Code coverage tools on open-source platforms provide an incentive to improve coverage and/or write tests. (1)	○	○	○	○	○
Display This Choice: If For this survey, we would like to make a distinction between code contributors and project mainta... = Maintainer/Project Manager I am more likely to approve a pull request that improves code coverage than ones that lower it. (4)	○	○	○	○	○
Display This Choice: If For this	○	○	○	○	○

Page 9 of 17

survey, we would like to make a distinction between code contributors and project mainta... = Code contributor	
If my pull request improves coverage, it is accepted more quickly, in my experience. (6)	

Q23 How often do you write tests for projects you are contributing to?

	For each commit (1)	Every few commits (2)	For each pull request (3)	Every few pull requests (4)	Rarely (5)	Never (6)
I write tests (1)	○	○	○	○	○	○

Q24 How often do you write a test or multiple tests with (just) the intent to improve the code coverage?

	Every day (1)	A few times a week (2)	A few times a month (3)	A few times a year (4)	Less than once a year (5)	Never (6)
I write tests with the intent to improve coverage (1)	○	○	○	○	○	○

Display This Question:

If For this survey, we would like to make a distinction between code contributors and project mainta... = Code contributor

Q26
How often are you asked/encouraged to better test your contributions, in the comments of a pull request you opened?

	Every day (9)	A few times a week (10)	A few times a month (11)	A few times a year (12)	Less than once a year (13)	Never (14)
People ask me this (3)	○	○	○	○	○	○
A coverage tool asks me this (8)	○	○	○	○	○	○

Display This Question:

If How often are you asked/encouraged to better test your contributions, in the comments of a pull r... [Never] (Count) < 2

And If

How often are you asked/encouraged to better test your contributions, in the comments of a pull r... [A coverage tool asks me this] (Recode) Is Not Empty

Or How often are you asked/encouraged to better test your contributions, in the comments of a pull r... [People ask me this] (Recode) Is Not Empty

Q27 Do you remember a particularly interesting instance where this happened? How did the situation get resolved?

____

____

____

____

____

Page 11 of 17

Display This Question:

If For this survey, we would like to make a distinction between code contributors and project mainta... = Maintainer/Project Manager

Q28 How often do you have to tell a contributor to a project you maintain that their tests need to be improved, based on the results of a coverage tool?

	Every day (15)	A few times a week (16)	A few times a month (17)	A few times a year (18)	Less than once a year (19)	Never (20)
I tell people this (3)	○	○	○	○	○	○

Display This Question:

If How often do you have to tell a contributor to a project you maintain that their tests need to be... [I tell people this] (Recode) Is Not Empty

And How often do you have to tell a contributor to a project you maintain that their tests need to be... != Never

Q29 Do you remember a particularly interesting instance where this happened? How did that situation get resolved?

____

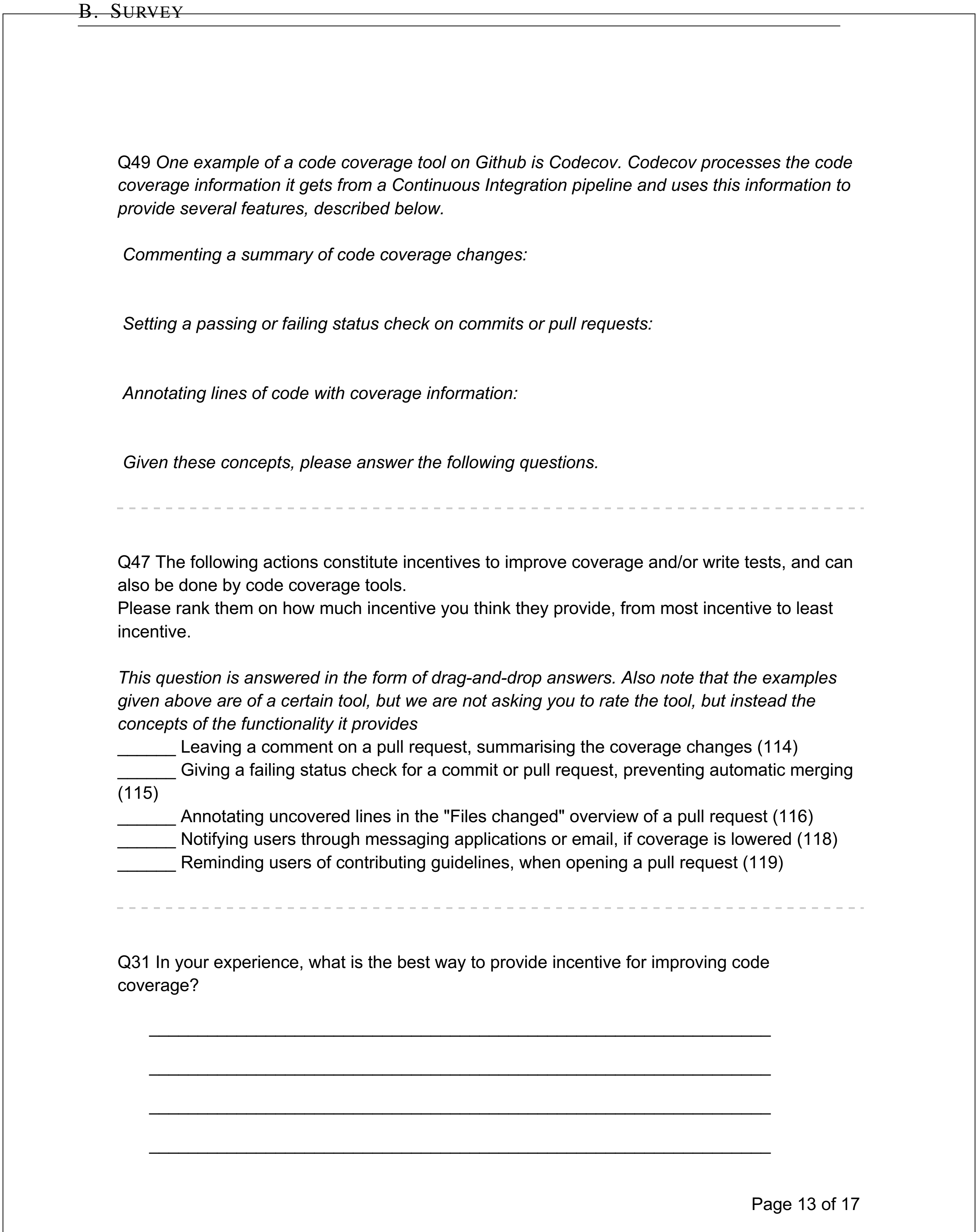
____

____

____

____

Page Break



Q49 *One example of a code coverage tool on Github is Codecov. Codecov processes the code coverage information it gets from a Continuous Integration pipeline and uses this information to provide several features, described below.*

Commenting a summary of code coverage changes:

Setting a passing or failing status check on commits or pull requests:

Annotating lines of code with coverage information:

Given these concepts, please answer the following questions.

Q47 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.

This question is answered in the form of drag-and-drop answers. Also note that the examples given above are of a certain tool, but we are not asking you to rate the tool, but instead the concepts of the functionality it provides
______ Leaving a comment on a pull request, summarising the coverage changes (114)
______ Giving a failing status check for a commit or pull request, preventing automatic merging (115)
______ Annotating uncovered lines in the "Files changed" overview of a pull request (116)
______ Notifying users through messaging applications or email, if coverage is lowered (118)
______ Reminding users of contributing guidelines, when opening a pull request (119)

Q31 In your experience, what is the best way to provide incentive for improving code coverage?

____

____

____

____

Page 13 of 17

____

End of Block: Block 4

Start of Block: Block 5

Q34 *In previous research, we have seen several pull requests with failing code coverage status checks that were merged anyway. Effectively this means that the check has been ignored.*

Q35 Can you come up with situations where you would ignore a failing coverage check? What are your reasons?

____

____

____

____

____

Q36 How often do you neglect or ignore a failing coverage check on a commit or pull request?

	Every day (3)	A few times a week (4)	A few times a month (5)	A few times a year (6)	Less than once a year (7)	Never (8)
I ignore a failing coverage check when contributing to a project (3)	○	○	○	○	○	○
I ignore a failing coverage check when reviewing a pull request (6)	○	○	○	○	○	○

End of Block: Block 5

Start of Block: Block 6

Q42 Can you give us 2 things you **like** about using a code coverage tool on an open-source platform?

____

____

____

____

____

Q43 Can you give us 2 things you **dislike** about using a code coverage tool on an open-source platform?

____

Page Break

Q44 Thank you for filling out this survey. We have just one more question for you

Was there anything unclear about this survey? Or are there any other comments you would like to make?

____

____

____

____

____

Q52 If you are open to the idea of a more in-depth interview regarding the survey topic, please leave us your email address here.
Note that your email address will not be stored with your individual responses, or made public.

____

End of Block: Block 6