

https://dspace.org/

A Longitudinal Study on the Effect of Patches on Software System Maintainability and Code Coverage

Mamba, Ernest Bonginkosi

2024
University of the Witwatersrand, Johannesburg
https://hdl.handle.net/10539/45429

Mamba, Ernest Bonginkosi . (2024). A Longitudinal Study on the Effect of Patches on Software System Maintainability and Code Coverage [Masters dissertation, University of the Witwatersrand, Johannesburg]. WIReDSpace. https://hdl.handle.net/10539/45429

Downloaded from a DSpace repository. https://dspace.org/

A Longitudinal Study on the Effect of Patches on Software System Maintainability and Code Coverage

Ernest Mamba

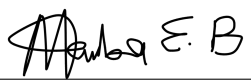
A dissertation submitted to the Faculty of Engineering and the Built Environment, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science in Engineering.

Johannesburg, October 2024

Declaration

I declare that this dissertation is my own, unaided work, except where otherwise acknowledged. It is being submitted for the degree of Master of Science in Engineering to the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination to any other university.

Signed this 20 day of October 2024



Ernest Mamba

Abstract

In the rapidly evolving landscape of software development, ensuring the quality of code patches could potentially improve the overall health and longevity of a software project. The significance of assessing patch quality arises from its pivotal role in the ongoing evolution of software projects. Patches represent the incremental changes made to the code-base, shaping the trajectory of a project's development. The identification and understanding of factors influencing patch quality could possibly contribute to enhanced software maintainability, reduced technical debt, and ultimately, a more resilient and adaptive code-base. While previous research predominantly concentrates on analysing releases as static entities, this study extends an existing study of patch testing while incorporating an examination of quality from a maintainability point of view, thereby filling a void in patch-to-patch investigations. Over $90,000$ builds spanning 201 software projects written in 17 programming languages are mined from two popular coverage services, Coveralls and Codecov. To quantify maintainability, a variant of the SIG Maintainability Model, a recognised metric designed to assess the maintainability of incremental code changes is employed. Additionally, the Change Risk Anti-Patterns (CRAP) metric is utilised to identify and measure potential risks associated with code modifications. A moderate correlation of 0.4 was observed between maintainability and patch coverage, indicating that patches with higher coverage tend to exhibit improved maintainability. Similarly, a moderate correlation was identified between the CRAP metric and patch coverage, suggesting that higher patch coverage is associated with reduced change risk anti-patterns. In contrast, patch coverage demonstrates no correlation with overall coverage, underscoring the distinctive nature of patches. However, it is noted that relying solely on patch coverage lacks comprehensive overview of coverage patterns. Thus, it is recommended to supplement it with overall system coverage for a more comprehensive understanding. Moreover, patch maintainability also exhibits no correlation with overall coverage, again, highlighting the unique nature of patches. In conclusion, the study offers valuable insights into the nuanced relationships between patch coverage, maintainability, and change risk anti-patterns, contributing to a more refined understanding of software quality in the context of software evolution.

To

my sons, Awande and Alwande Mamba and my daughter Siphosethu Mamba.

Acknowledgements

I would like to convey my sincerest gratitude to my supervisor Dr. Steve Levitt for all the support, guidance and great working and collaboration environment throughout the years.

I would also like to acknowledge the authors of ***A Large-Scale Study of Test Coverage Evolution*** for their great assistance in extending their work and clarifying some of the concepts.

Lastly, greatest thanks to the Codecov and Coveralls support team, whose constant engagements made this work possible by either fixing their web API and/or clarifying the definitions of some of the data returned by the API.

Contents

Declaration **i**

Abstract **ii**

Dedication **iii**

Acknowledgements **iv**

Contents **v**

List of Figures **x**

List of Tables **xiv**

Nomenclature **xvii**

1 Introduction **1**

1.1 Background 1

1.1.1 Software Evolution 3

1.1.2 Software Testing 4

1.2 The Concept of a Patch 5

1.2.1 Defining and Measuring Patch Size 5

1.2.2 Patch Coverage 6

1.3 Problem Statement 8

1.4 Research Questions 10

1.5 Dissertation Layout 11

2 Existing Work **13**

2.1 Defining Software Quality 13

2.1.1 Software Maintainability 15

2.1.2 Software Complexity 19

2.1.3 Software Testing 22

2.1.4 Change Risk-Anti Patterns (CRAP) Metric 23

2.2 Exploring Software Quality Evolution: An Empirical Review 24

2.2.1 Coverage and Defect Detection Relationship 25

2.2.2 Software Evolution 26

2.2.3 SIG Maintainability Model 30

2.3 Chapter Summary 32

3 Methodology **33**

3.1 Research Approach 33

3.1.1 Compute Infrastructure 34

3.2 Overview of the Dataset 35

3.3 Data Collection 36

3.3.1 Project Selection 36

3.4 Data Cleaning and Processing 38

3.4.1 Overview of data issues 38
3.5 Metrics and Measures 40
3.5.1 Coverage Service APIs 40
3.5.2 Algorithm 44
3.5.3 Delta Maintainability Model 51
3.5.4 The Case For Commit CRAP 52
3.6 Analysis Approach 53
3.7 Chapter Summary 54

4 Results and Discussion **55**

4.1 Data Summary 55
4.2 Characteristics of Patches 56
4.2.1 Co-Evolution of Production and Test Code in Patches 57
4.2.2 Analysing the Typical Size of Patches 58
4.3 Testing Dynamics Across Patches 59
4.3.1 Distribution of Patch Coverage 59
4.3.2 Impact of Patches on System Coverage 64
4.4 Evolution of Quality Across Patches 69
4.4.1 Maintainability of Fine-Grained Changes 69
4.4.2 CRAP Metric Analysis Over Revisions 72
4.5 Chapter Summary 75

5 Conclusion **77**

5.1 Key Contributions 77

5.2 Threats to Validity 80
5.2.1 Models and Definitions 80
5.2.2 Sample Size & Programming Languages 80
5.2.3 Data Integrity 81
5.3 Research Implications 82
5.3.1 Implications for Build and Coverage Tools Providers 82
5.3.2 Implications for Developers 83
5.3.3 Implications for Researchers 83
5.4 Further Research 84

References **85**

A Answers to Demographic Questions **100**

A.1 EQ-1 : Coevolution of Production and Test code 100
A.2 EQ-2: Typical Size of a Patch (Source Files Only) 102

B Patch Coverage Manual Calculation **104**

B.1 Coveralls 104
B.2 Codecov 105
B.3 Differences (Inconsistencies) 107

C SIG Delta Maintainability (SIG-DMM) **108**

C.1 Risk Profile Unit Size 108
C.2 Risk Profile Delta Unit Size 109
C.3 Risk Profile Delta Increases/Decreases Per File 109

C.4 Commit Risk Profile Delta Increases/Decreases Per File 110

C.5 Commit Risk Profile Delta Increases/Decreases Per File 110

D Virtualisation Configuration **112**

D.1 Dockerfile 112

D.2 Instantiating a Container 113

E Web API calls response **115**

F Codecov Correlating Coefficients **119**

G Project Description **126**

H Additional Figures **133**

H.1 Patch Coverage - Code and Non-Code Builds 133

List of Figures

1.1 Production file and Test file commit pair for Patch Coverage 7

2.1 ISO/IEC 25010 Software Quality Model 14

2.2 Relation between SIG Maintainability Model and source code measures and ISO/IEC 25010 quality characteristics 17

2.3 ISO/IEC 25010 - SIG Metrics - Measure Mapping Flow 18

3.1 Overview Research Methodology 34

3.2 Compute Infrastructure Architecture 35

3.3 Apache/Gobbline before (**Top**) and After (**Bottom**) data cleaning with EMA 42

3.4 Build and Commit Hash Relation 45

3.5 Algorithm pipeline-flow 47

3.6 Modified Files Block: FileMatcher function 48

3.7 Covered Lines Block: LineMapper function 49

4.1 Demographic Question 1: Coveralls - Coevolution of Production and Test code. Size of bar represents the proportion of builds over the total number of builds in each category. 58

4.2 Exploratory Question 2: Codecov - Boxlot of Source Code patch Size (Code Churn). Horizontal axis represent the number of lines modified. 59

4.3 Codecov: Distribution of patch coverage across revisions. Each colour represent the range and the size of the bar is the percentage of the patches whose coverage lies within the range. 61

4.4 Codecov Cont: Distribution of patch coverage across revisions. Each colour represent the range and the size of the bar is the percentage of the patches whose coverage lies within the range. 61

4.5 Coveralls: Distribution of patch coverage across revisions. Each colour represent the range and the size of the bar is the percentage of the patches whose coverage lies within the range. 62

4.6 Coveralls Cont: Distribution of patch coverage across revisions. Each colour represent the range and the size of the bar is the percentage of the patches whose coverage lies within the range. 62

4.7 Distribution of patch coverage across revisions. Each colour represent the range and the size of the bar is the percentage of the patches whose coverage lies within the range. 63

4.8 Native Rest API Patch Coverage From Codecov: Distribution of patch coverage across revisions. Each colour represent the range and the size of the bar is the percentage of the patches whose coverage lies within the range. 64

4.9 Native Rest API Patch Coverage From Codecov: Distribution of patch coverage across revisions. Each colour represent the range and the size of the bar is the percentage of the patches whose coverage lies within the range. 65

4.10 Codecov: Distribution of the impact of revisions to the overall system coverage. Each colour represent the range and the size of the bar is the percentage of the patches whose impact lies within the range. . . 66

4.11 Codecov Cont: Distribution of the impact of revisions to the overall system coverage. Each colour represent the range and the size of the bar is the percentage of the patches whose impact lies within the range. 66

4.12 Coveralls: Distribution of the impact of revisions to the overall system coverage. Each colour represent the range and the size of the bar is the percentage of the patches whose impact lies within the range. . 67

4.13 Coveralls Cont: Distribution of the impact of revisions to the overall system coverage. Each colour represent the range and the size of the bar is the percentage of the patches whose impact lies within the range. 67

4.14 Coveralls: distribution of Delta Maintainability (DMM). Each colour represent the range and the size of the bar is the percentage of the patches whose impact lies within the range. 70

4.15 Coveralls Cont: distribution of maintainability (DMM) . Each colour represent the range and the size of the bar is the percentage of the patches whose impact lies within the range. 70

4.16 Codecov: distribution of maintainability (DMM) across revisions. Each colour represent the range and the size of the bar is the percentage of the patches whose impact lies within the range. 71

4.17 Codecov Cont: distribution of maintainability (DMM) across revisions. Each colour represent the range and the size of the bar is the percentage of the patches whose impact lies within the range. 71

4.18 Codecov CRAP Metric: distribution of the delta CRAP metric across revisions. 73

4.19 Codecov CRAP Metric Cont: distribution of the delta CRAP metric across revisions. 73

4.20 CRAP Metric Coveralls: distribution of the delta CRAP metric across revisions. 74

4.21 CRAP Metric Coveralls Cont: distribution of the delta CRAP metric across revisions. 74

A.1 Exploratory Question 1: Coveralls Cont - Coevolution of Production and Test code. Size of bar represents the proportion of builds over the total number of builds in each category. 100

A.2 Exploratory Question 1: Codecov - Coevolution of Production and Test code. Size of bar represents the proportion of builds over the total number of builds in each category. 101

A.3 Exploratory Question 1: Codecov Cont - Coevolution of Production and Test code. Size of bar represents the proportion of builds over the total number of builds in each category. 101

A.4 Exploratory Question 2: Codecov Cont - Distribution of Source Code patch Size (Code Churn) 102

A.5 Exploratory Question 2: Coveralls Cont - Distribution of Source Code patch Size (Code Churn) 102

A.6 Exploratory Question 2: Coveralls Cont - Distribution of Source Code patch Size (Code Churn) 103

H.1 Codecov Patch Coverage distribution: The bar chart illustrates the distribution of Patch Coverage in patches affecting Code and Non-Code files, with each bar representing the percentage of patches falling within a specific coverage range. 133

H.2 Codecov Patch Coverage distribution Cont: The bar chart illustrates the distribution of Patch Coverage in patches affecting Code and Non-Code files, with each bar representing the percentage of patches falling within a specific coverage range. 134

H.3 Coveralls Patch Coverage distribution: The bar chart illustrates the distribution of Patch Coverage in patches affecting Code and Non-Code files, with each bar representing the percentage of patches falling within a specific coverage range. 134

H.4 Coveralls Patch Coverage distribution Cont: The bar chart illustrates the distribution of Patch Coverage in patches affecting Code and Non-Code files, with each bar representing the percentage of patches falling within a specific coverage range. 135

List of Tables

2.1 SIG MM properties with respective low risk thresholds 18

2.2 Summary of Covered concepts 32

4.1 Sample Representation of Projects Key Statistics 56

4.2 Projects Coverage Service Correlations 75

5.1 Summary of Research Findings 78

C.1 Commit Statistics 108

C.2 Risk Profile Per File 109

C.3 Risk Profile Delta Per File 109

C.4 Risk Profile Delta Increases/Decreases Per File 110

C.5 Commit Risk Profile Delta Increase/Decrease 110

F.1 Spearman and Kendall's Correlation Coefficient 120

F.2 Spearman and Kendall's Correlation Coefficient 121

F.3 Spearman and Kendall's Correlation Coefficient 122

F.4 Coveralls: Spearman and Kendall's Correlation Coefficient 123

F.5 Coveralls Cont: Spearman and Kendall's Correlation Coefficient . . . 124

F.6 Coveralls Cont: Spearman and Kendall's Correlation Coefficient . . . 125

G.1 Codecov Project Key Statistics 127

G.2 Codecov Project Key Statistics Cont 128

G.3 Codecov Project Key Statistics Cont 129

G.4 Coveralls Project Key Statistics 130

G.5 Coveralls Project Key Statistics Cont 131

G.6 Coveralls Project Key Statistics Cont 132

List of Algorithms

1 Patch Coverage algorithm 46

Nomenclature

API Application Programming Interface

CC Cyclometric Complexity

CRAP Change Risk Analysis and Predictions

DMM Delta Maintainability Index

KA Knowledge Area

MI Maintainability Index

SIG Software Improvement Group

SDLC Software Development Life Cycle

STLC Software Testing Life Cycle

1

Introduction

This chapter introduces a research investigation into the progression of software quality, emphasising software maintainability and testing, with a particular focus on fine-grained changes. The concepts are succinctly introduced to establish a robust foundation for the subsequent portions of the study. The chapter elucidates the contextual framework under which the evolution is examined and formulates pertinent research questions. The dissertation layout is presented for streamlined navigation.

1.1 Background

"The only thing that is constant is change."

– Heraclitus, *Greek Philosopher*

Heraclitus, the renowned Greek philosopher, asserted that the only certainty in life is change - a sentiment that resonates in the dynamic realm of software engineering. Meir M. Lehman, a prominent figure in the field of software engineering, particularly in the area of software evolution, contends that a software system's enduring utility and success hinge on its ability to evolve continuously; failure to do so leads to a decline in relevance and quality [1,2]. Lehman's laws of evolution posit that a software system's functional capabilities must evolve to maintain user satisfaction, inevitably resulting in an increase in system size and complexity over time with a concurrent decline in system quality unless actively monitored and addressed [3].

Central to the concept of software evolution is the source code, the pivot of a multifaceted process that requires the co-evolution of various artifacts, including unit tests (also known as developer tests). These tests, proven effective in identifying bugs [4], play a pivotal role in ensuring the continued proper functioning of a system. However, the intricate relationship between source code and tests becomes apparent during software evolution, as even minor changes or refactoring efforts can disrupt existing tests [5, 6]. Moonen *et al.* [6] note the susceptibility of tests to changes, while Elbaum *et al.* [7] emphasise the significant impact of even minor changes on coverage information. This underscores the indispensable role of tests in the success and evolution of software, warranting maintenance on par with the source code itself.

The study of software evolution dates back to the early 70's when Lehman introduced the concept. Subsequent research has delved into validating Lehman's laws and exploring various aspects of software evolution, including complexity, quality, and maintainability [8–10]. A noteworthy linkage between software complexity and testing has been established [11], emphasising a need for a strategically designed evolution process that aligns the quality of source and test code for successful software evolution. In addition to its influence on testing, high complexity is often seen as the main source of errors and is said to lead to an increased effort to understand, implement and modify a component, subsequently leading to undesirable quality behaviour such as fault proneness and decreased maintainability [12]. A reduction in maintainability implies that changes to application logic is often difficult which often lead to difficulty in testing the modified component [13].

Previous research efforts have extensively mined multiple released versions of software programs to understand how quality declines with increasing complexity over time [14]. While such an approach provides insights at stable points in time, the rise of Continuous Integration (CI), Continuous Delivery (CD) [15], and agile methodologies allows for a closer examination of day-to-day development activities. In this context, developers must pay meticulous attention to all commits pushed upstream, especially those with the potential to compromise or diminish software quality [16].

In contextualising the background of this study, it is noteworthy that while a significant body of research has primarily concentrated on mining various aspects of software development, two seminal studies, conducted by Marinescu *et al* [17] and Hilton *et al.* [18], stand out for their pioneering work in the exploration of fine-grained code coverage. These studies played a pivotal role in shaping the understanding of the intricate dynamics surrounding commit changes in software evolution. Acknowledging and building upon the foundations laid by Hilton *et al.* and Marinescu *et al*, this

research extends their focus by incorporating an additional layer of analysis, specifically examining the interplay between commit changes, maintainability, and fine-grained code coverage.

1.1.1 Software Evolution

Software evolution is the dynamic behaviour of a programming system as it is *maintained* and *enhanced* over its life span [19]. The emphasis here is on maintenance and enhancement. Maintenance and enhancement refers to the correction of bugs and the implementation of modifications required to allow for an existing system to perform new functionality and to perform old ones under the new conditions [19]. In order for maintenance to be possible, certain requirements are desired on the current version of the software system, one such is minimal structural complexity. Less complexity enables the project team (developers and testers) to more easily develop new and modify old functionality and test the behaviour of the newly developed product under the new conditions. As such, an important phase of the software development life cycle (SDLC) is the testing phase, which has come to have its own independent life cycle, namely, software testing life cycle (STLC).

There exist a strong relationship between structural complexity and testing [11]. At first, complexity has come to be known as a common source of error, this applies in both the abstract and concrete sense [20]. In the abstract sense, excessive complexity hinders accurate symbolic manipulations i.e changing of wording/sentence, formulas, diagrams, wording etc, leading to errors [20]. The psychological factors limiting human mental manipulations also apply to software. This is known as *cognitive complexity*. According to Briand *et al.* [21], cognitive complexity is defined as the mental burden for the person(s) who have to deal with the code, which includes developers, inspectors, testers, maintainers etc. The internal attributes of a class, such as coupling and size are indicators of cognitive complexity. Structured programming, unlike functional programming or procedural programming reduces, yet, it does not eliminate this issue. In the concrete sense, structural complexity studies have shown that complexity correlate with errors in a software module. Thus, complexity can be used to allocate testing effort through leveraging the connection between complexity and errors to focus the testing effort on the most error-prone modules of a software program [20].

Given the claim from Muslija [11] regarding a relationship between complexity and testing, researchers have conducted empirical studies to verify this relationship between

structural complexity and testing. A significant number of findings established a link between the complexity and testing. However, software testing is rather a broad field and knowledge area of its own, therefore, it is not surprising that the findings would not cover all forms of testing. Currently, a large amount of effort has been directed at studying testing at a release level and not much work has reviewed testing on a commit-by-commit basis. Hence, very few studies have been dedicated to understanding the dynamics of unit testing (a component of software testing) and maintainability at the commit level. Thus, the aim of this study was to empirically evaluating software structural quality behaviour from a testing (unit testing) and maintainability perspective as a software is modified over a period of time. This is was done by studying open source projects hosted on GitHub and collecting maintainability metrics and testing metrics over multiple patches (also known as revisions).

1.1.2 Software Testing

As previously mentioned, software testing is one of the most critical phases of the development process and a critical element of evaluating quality assurance [22]. The process of testing is usually practised for two reasons:

1. Defects Detection
2. Reliability/Quality Estimation

Moreover, testing remains the sole method for evaluating system requirements and specifications against system behaviour. This is done by applying various possible input scenarios and observing the output in comparison to the expected output. Should the test output not match the expected output, the system is modified and the new changes are tested again, and this process is done iteratively until all conditions and expected behaviour is satisfied. Because of this, the testing process is costly, time consuming and labour intensive [23].

Traditionally, there are two types of testing strategies, namely *static* and *dynamic* testing. The former method involves static analysis of a computer program. Techniques employed in performing static analysis involve informal reviews, technical reviews, code inspection and walk-through [20]. Static testing is typically done to verify conformity to things such as naming conventions, code structuring and best practices. Tools used include tools like SonarQube. The dynamic approach on the

other hand involves analysing the dynamic response of the program, that is, running the program and observing the output and analyse how that changes as a function of time. It requires supplying the system with input values and checking the output by running specific test cases. Tools used include tools like Veracode. Unit testing, integration testing, system testing and acceptance testing are a form of dynamic testing. That said, this paper aims at exploring unit testing which is a subset of dynamic testing.

1.2 The Concept of a Patch

In light of the aforementioned significant emphasis on software evolution and testing, particularly in minimising complexity and enhancing maintainability, meticulous consideration to small incremental changes, such as patches, is imperative. This section elucidates the concept of small patches and defines the true essence of fine-grained coverage.

1.2.1 Defining and Measuring Patch Size

A commit patch, or simply patch, is the number of modified code statement(s) or lines of code committed to a repository. Modification refers to altering, deleting, or adding of lines. A patch is identified by a Git hash, which is a series of forty characters generated using `SHA-1` cryptographic hash function. To illustrate this, consider Listing 1 as the first commit which defines a Python class with a single static method. In the second commit the class is updated as in Listing 2. The patch size for the first patch (`7360e9c`) would be all the lines in Listing 1 since there is no previous reference point i.e lines `{1-5}` were the added lines and no lines were either changed or deleted. The size of the patch in this case would be said to be `5` (including the empty space in line `1` i.e 4 executable lines plus 1 non-executable). However, for the second commit (`e7e2b8da`) the patch is given by the modified lines only, lines `{2,5-7,9-12}` in Listing 2 . This patch is said to be of size `8` (3 non-executable plus 5 executable lines). That is, lines `{2-5}` from the previous commit (Listing 1) did not change, therefore, they are not counted as part of the patch. For the remainder of the report, the term patch and commit are used interchangeably.

```
1
2  class Calculator:
3      @staticmethod
4      def divide(a: int, b: int) -> float:
5          return a/b
```

Listing 1: Patch of `Calculator.py` at commit 7360e9c

```
1  class Calculator:
2      # method to divide two numbers
3      @staticmethod
4      def divide(a: int, b: int) -> float:
5          # raise an exception if the denominator is zero
6          if b == 0:
7              raise ZeroDivisionError
8          return a/b
9      # method to subtract two numbers
10     @staticmethod
11     def subtract(a: int, b: int) -> int:
12         return b - a
```

Listing 2: Patch of `Calculator.py` at commit e7e2b8da

1.2.2 Patch Coverage

Marinescu *et al.* [17] defined patch coverage as "the ratio between the number of executed lines of code added or modified by a patch and the total number of executable lines in the patch, measured in the revision that adds the patch". In other words, unlike project coverage, patch coverage is a more granular measure focused on the percentage of tests covering the newly added code changes [24, 25]. `Executed` lines are lines that are invoked during a test execution whilst `executable` lines are all "source lines" that have the potential to be invoked, yet, may or may not be necessarily invoked. For example, in interpreted languages, these are lines that are interpreted at run-time whilst in compiled languages this would be lines compiled by the compiler. This definition would exclude statements like comments, as these are excluded by

both interpreter and compiler. To illustrate this, consider Listings 3 and 4 testing Listings 1 and 2 from the previous example.

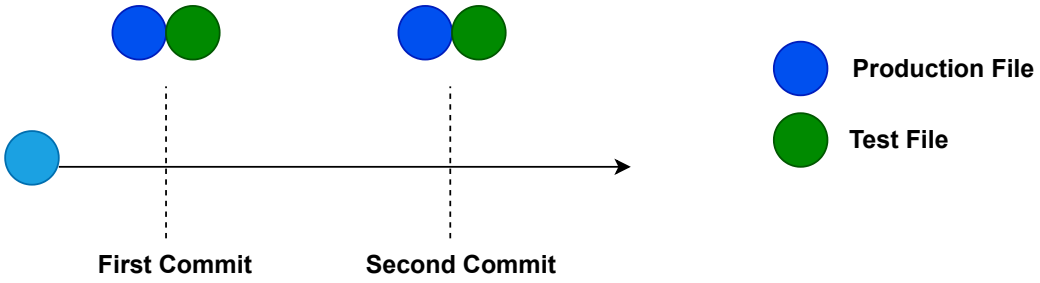


Figure 1.1: Production file and Test file commit pair for Patch Coverage

```
1  import unittest
2  from calculator import Calculator
3
4  class TestCalculator(unittest.TestCase):
5      def test_divide(self):
6          self.assertEqual(2.0,Calculator.divide(4,2))
7
8  if __name__ == '__main__':
9      unittest.main()
```

Listing 3: Patch Coverage of `CalculatorTest.py` at commit 7360e9c

To understand patch coverage, first, a segregation between program logic files and test logic files established. A production file is defined a source code file that contains the application logic i.e a file that contains code which forms the brain of the program. A test file is a source file designated to the testing of the application logic. Suppose Listing 3 (Production file in Figure 1.1 for the first commit) was committed alongside Listing 1 (Test file in Figure 1.1 for the first commit) and Listing 4 (Production file in Figure 1.1 second commit) was committed alongside Listing 2 (Test file in Figure 1.1 second commit) . If Listing 3 is committed alongside Listing 1, the total patch size (from both files) is 18 (11 executable plus 7 non-executable line). Listing 3 covers (i.e "executes") all flows in Listing 1, therefore, the patch coverage is 100%. In Listing 4 and Listing 2 for the second commit, the combined patch size is 12 (4 non-executable and 8 executable lines).

```
1  import unittest
2  from calculator import Calculator
3
4  class TestCalculator(unittest.TestCase):
5      def test_divide(self):
6          self.assertEqual(2.0, Calculator.divide(4,2))
7
8      def test_subtract(self):
9          self.assertEqual(4, Calculator.subtract(3,7))
10
11 if __name__ == '__main__':
12     unittest.main()
```

Listing 4: Patch Coverage of `CalculatorTest.py` at commit e7e2b8da

The corresponding patch coverage is found to be `85.71%` because line `{7}` (which raises an exception) is not tested in Listing 4. Note that the line `b == 0` (Listing 2) is invoked during the test run as the condition has to be evaluated before it can be skipped or executed (though skipped in this instance).

In the event that a third commit, which adds a method to test the `ZeroDivisionError` exception, with the logic (Calculator class) not modified, the patch would be 100%. That is, any patch covering existing code is a fully covered patch. Similarly, adding patch logic which is already covered by existing tests would result in 100% patch coverage.

1.3 Problem Statement

Software evolution remains a critical process in producing software products as newer and more features are added to an already existing system to form a new version in order to satisfy the ever growing demands and changing needs of consumers. Consequently, software quality, re-usability and maintenance remain a top priority for organisations and developers as the associated costs due to lack of these capabilities are very high. For example, Omari *et al.* [26] and Kadry *et al.* [27] found that the driving costs of developing a software product was maintenance and accounts for 67% of the overall costs whilst unit testing (second highest) ranges between 7% to

8%. Various techniques for enabling high quality, robust, reusable and maintainable software exist, such as Test Driven Development (TDD).

On the other hand, the fast growing adoption of the agile framework inherently means developers, quite often, deal with small incremental changes. This model of work also supports the adoption of build services like GitHub Actions [28], Jenkins [29], Travis CI [30] and many others, for the continuous integration and continuous deployment of applications. With Cassee *et al.* [31] and Zhao *et al.* [32] outlining the benefits of continuous integration and continuous builds such as quality improvement, it comes as no surprise that developers would wish to pay close attention to all commits pushed to upstream repositories as they could either break the build or reduce code quality. Embracing both agile framework and CI means product teams can now track the incremental changes to programs and receive continuous feedback more frequently than ever before. This also allows for historical data to be analysed in greater detail to prepare and improve for future changes.

A plethora of literature exists on software evolution, ranging from validation of Lehman's laws of evolution [33–36] to quality and maintenance evolution and the inter-and-intra connectivity of these concepts. Whilst a large effort has been dedicated to studying software quality evolution on a release-by-release approach, there are still very limited studies directed at investigating fine-grained (i.e commit level) analysis of testing and maintainability. For example, some studies are empirical studies dedicated at understanding commit behaviour [37–41], however, these are directed at understanding the co-evolution of production code and test code. These studies are extensively discussed in Chapter 2. Prior studies on commit level dynamics often face challenges due to researchers relying on custom code analysis instead of utilising available testing tools, hindering thorough investigation. Integrating existing tools can streamline analysis, enabling more comprehensive, as well as repeatable research efforts.

This report extends the work of Hilton *et al.* [18] and Marinescu *et al.* [17], which investigated the concept of patch coverage using 46 and 6 projects respectively. The previous work is extended by:

1. Increased Number of Projects - The total number of projects investigated in this report is 201 projects, which is multiples of both Both Hilton *et al.* [18] and Marinescu *et al.* [17]. Thus, the size has been increased for representativeness.

2. Timeline of build history - Both Hilton *et al.* and Marinescu *et al.* opted to use a minimum of 250 builds. In this investigation, however, instead of focusing on

the number of patches, the focus is placed on the time-span from the first build to the latest build. Thus, the builds are chosen based on a minimum period of six months apart.

3. Repeatability - Hilton *et al.* used open source projects, as such, the data from these studies is mined to compare and contrast the results from their study to the results of this report.

4. Multiple sources and data integrity - Hilton *et al.* mined half of their data from a single coverage service tool. This report mined coverage data from two popular coverage services, which diversify the data analysed. The integrity of the data from these coverage services is also evaluated in detail.

5. Patch Coverage Algorithm - An independent patch coverage algorithm is developed to compute patch coverage and contrast with patch coverage as returned by the coverage service API.

1.4 Research Questions

To date, two seminal studies, notably from Hilton *et al.* [18] and Marinescu *et al.* [17] investigated the dynamics of patch coverage at a commit level. These two studies yield somewhat conflicting results (see Chapter 2 for thorough discussion). The investigation at hand extends the work of both studies and adds the dimension of maintainability to holistically understand the patch behaviour over a period of time. Thus, the formulated research questions are framed as follows:

1. What is the distribution of coverage across revisions? This question, previously explored by Marinescu *et al.* [17] and reiterated by Hilton *et al.* [18], investigates the commit-by-commit basis coverage definition from Marinescu's study. Both studies yielded conflicting evidence regarding the distribution of patch coverage. This question aims at consolidating the previous finds through a large dataset and diverse programming languages.

2. How maintainable is a typical patch, how does maintainability vary across revisions? Maintainability defined in Section 2.1, is measured using the Software Improvement Group's (SIG) maintainability model.

3. To what extent is a typical patch considered "CRAPpy"? The Change Risk Anti-Patterns (CRAP) metric, as defined by Alberto Savoia [42, 43], is a metric

designed to analyse and predict the effort and time required to maintain existing code. This metric combines cyclomatic complexity and coverage to identify sub-optimal code quality. The study adopts a patch-by-patch approach to assess the CRAP metric of changes.

1.5 Dissertation Layout

This document is crafted to offer a comprehensive and well-organised research investigation. The structured format facilitates easy navigation and retrieval of relevant information, minimising the risk of overlooking key details. The subsequent sections are organised as follows:

Chapter 2: **Existing Work** presents the concepts and definitions pertaining to software quality, specifically focusing on testing and maintainability as applied in the context of the study. Subsequently, empirical studies within these domains are presented, and any identified gaps in the existing research are highlighted.

Chapter 3: **Methodology** provides a comprehensive presentation of the methodology, covering the algorithm for computing metric values, considerations and procedures for data cleaning, and the overall data processing approach. The statistical analysis approach taken is also presented.

Chapter 4: **Results and Discussion** provides a detailed account of the study results and findings, accompanied by an analysis of factors influencing the research outcomes. Additionally, the meaning of each finding within the context of the research questions is presented.

Chapter 5: **Conclusion** summarises the results, findings and contributions of the study. It outlines the implications for coverage platforms, researchers, and developers. Additionally, the document addresses challenges and threats to validity and proposes avenues for future research.

Appendices A to H provide support supporting documentation for this longitudinal study. In particular, Appendix A relates to two previously research questions of which the study extends. Appendix B details the experimental repository used for verification of the concepts and corresponding outputs (both manual and API). Appendix C clarifies the concept of patch maintainability and provides and example of how to calculate the measure value. Appendix D is the virtualisation setup to

execute the scripts in a containerised environment. Appendix E gives a sample output from the API used in conducting the investigation. Appendix F are the correlating coefficients for each projects. Appendix G provides the major statistics of the open source repositories used in conducting the research. Finally, Appendix H depicts the patch coverage results of combined source code and none-source code patches.

2

Existing Work

This chapter describes the background into software evolution and quality activities within the software development field in general, this is done to provide a conceptual framework of the underlying principles of the two aforementioned Knowledge Areas (KA) and to provide the respective software evolution and quality definitions as applicable to this study. Results from prior empirical studies is later provided to pave way for the current study.

2.1 Defining Software Quality

The definition of quality is not as straightforward as one would think. There are various definitions that are applicable to different environments. Venkataraman *et al.* [44] and Fitzpatrick [45] state that the International Standard Organisation (ISO) defines quality as "the totality of features and characteristics of a product or services that bears its ability to satisfy specified or implied needs". Fitzpatrick offered their own definition of quality as "the extent to which an industry-defined set of desirable features are incorporated into a product so as to enhance its lifetime performance". Whilst these definitions capture the essence of quality, they are relatively broad for the context of this study. An alternative view of software quality is offered by Chapell [46]. Chapell does not necessarily offer a definition but rather argues that quality should be viewed from three perspectives, namely, functional, structural and process.

- Functional: is related to the functional capabilities of the software i.e does the software perform the tasks as required by the user. The attributes of this perspective of quality include but not limited to meeting user requirements,

minimal defects, ease of use etc. In general, software testing effort is spent ensuring an increase in this type of quality.

- Structural: This view is concerned with the structure of the code itself. Chapell argues that this type of quality is often hard to test. The attributes associated with this quality are code testability, code maintainability, code readability etc. This study takes a different view in a sense that it argues that the functional quality is concerned with User Acceptance Testing (UAT) whilst structural quality includes automated testing, such as unit testing, regression etc. In essence, testing that happens at the source code level. Thus, this is the quality investigated in the study.

- Process: is concerned with developmental processes involved and human resource related aspects. The most common attributes are budgets, development methodology etc.

The definitions provided above form the basis of the ISO/IEC-9126 (now deprecated and replaced by ISO/IEC 25010:2011) model that sketches the respective characteristics and sub-characteristics of quality. The first quality model dates back to the 1970's when McCall [47] first proposed it. Subsequent models were proposed by Boehm [48] and Dromey [49, 50]. The ISO/IEC-25010 is a composite model made of McCall, Dromey, and FURPS by Robert Grady [51]. Figure 2.1 depicts the quality model as formulated for the ISO/IEC-25010 based on the above definitions.

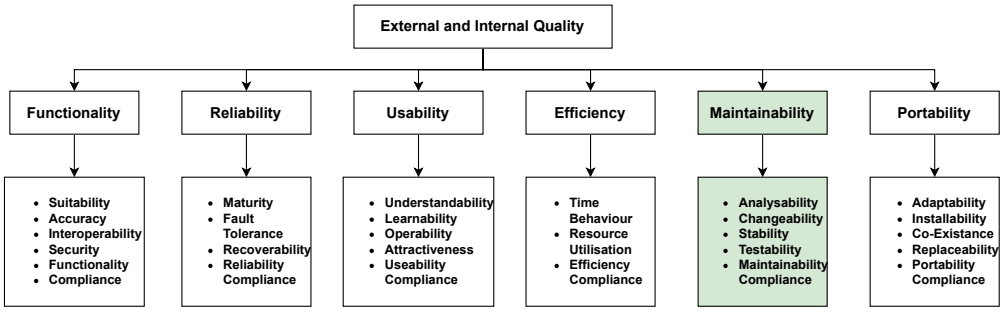


Figure 2.1: ISO/IEC 25010 Software Quality Model

The highlighted portion is what this report investigated, this is also mapped back to the structural quality perspective as defined by Chapell [46].

2.1.1 Software Maintainability

As previously seen from Figure 2.1, software maintainability is subcategory of software quality. Maintainability can be further broken down into various components or attributes. By definition, the ISO/IEC-25010 [52] defines maintainability as "the capability of the software product to be modified". An alternative yet almost similar definition is offered by the Standard Glosssary of Software Engineering Terminologies form the Institute of Electrical and Electronic Engineers (IEEE) as "the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment" [53]. Both definitions are sufficient as the report investigates changes applied to source code across the history of the application's existence. These modifications are classified as the "dimensions of software maintenance" by Swanson [54]. Swanson, argued that there are three bases of maintenance, namely, corrective, adaptive and perfective. Corrective maintenance are changes made in response to system failures. Adaptive changes are related to adjusting system to a different environment (such as OS) and/or data changes. Perfective changes are directed at improving efficiency and maintainability, examples are new features. To quantify maintainability, seminal work by Paul Oman [55] provided a metric to measure the maintainability of a program.

Maintainability Index

The Maintainability Index (MI) metric was first introduced by Paul Oman and Jack Hagemeister [55] in 1992. The MI was subsequently refined by Oman *et al.* in [56]. The goal of the MI index is to predict the maintenance burden required to maintain a piece of code. The original equation to compute the measure as proposed by Oman is given by equation (2.1). There have been subsequent variants of the equation driven by the fact that Oman and Hagemeister themselves argued that some of the properties were optional [57, 58].

$$MI = 171 - 5.2 \times \ln E - 0.23CC - 16.2 \ln LoC + 50 \sin \sqrt{2.46\ perCOM} \quad (2.1)$$

where
E is Halstead effort per module. The effort measures the cognitive effort needed for program creation [59].
CC is the Cyclomatic Complexity per module.

LoC is lines of code per module.
perCOM is number of lines of comment per module

The general rule of thumb is that if the MI measure is less than 65, it indicates the software is difficult to maintain. A value between 65 and 85 is considered as moderate maintainability which means the program can be modified but to a certain extent as there are still modules that are difficult to change. A measure value greater than 85 is considered a highly maintainable program, and thus, can be modified with relative ease. The MI suffers from a number of issues, such as consistency - the assertion that certain properties are optional does not offer reassurance in its robustness. These issues are addressed in Section 2.1.1.

SIG Maintainability Model

As part of the Software Improvement Group (SIG) Kuipers *et al.* [60] and Heitlager *et al.* [61] studied the first maintainability model as proposed by Paul Oman *et al.* [55, 56] in great detail and stated a number of limitations with the metric to measure maintainability of a program. A large part of the argument raised by [60, 61] was that the original MI is derived from a fitting curve based on experimental data, and therefore, is not in sync with the ISO/IEC 25010 software quality model as shown in Figure 2.1. Relevant comments have been restated here, and for a comprehensive critique, readers are directed to the detailed work of Heitlager *et al.* [60, 61]:

- Root-Cause analysis - With the MI being a composition of other metrics, identifying the root cause to the final value of the measure becomes complicated especially considering the fact that the Halstead volume is also a composite metric.
- Measure Computability - The MI is difficult to compute since it takes into account the number of operands and operators, which are in of themselves a topic of controversy i.e the definitions of what constitutes an operator and operand varies from language to language.
- Averaging of Complexity - It is argued that the use of averages is flawed for systems using object oriented principles because in general, the complexity per module tends to follow a power distribution law, as such, the average will inherently be lower.
- Use of Comments - The mathematical equation of MI index as introduced by Oman and Hagemeister included (optionally) source code documentation

(represented by comments) as part of the metric. The Authors argue that from their prior empirical studies and experience they often found that it is precisely complex and complicated code which often require comments, i.e simple code should be easily readable by looking at attributes such as method name, parameter name and the code itself - that is, code should be self-explanatory.

The above mentioned ambiguities or inadequacies necessitated a new maintainability model, that will 1) map the metric and ultimately the measure back to the ISO/IEC-25010 software quality model and 2) be easy to compute and consistent across multiple languages. Figure 2.2 depicts how the model, known as the SIG Maintainability Model (SIG MM) was developed through the mapping of ISO/IEC-25010 quality characteristics to source code metrics and ultimately to code properties.

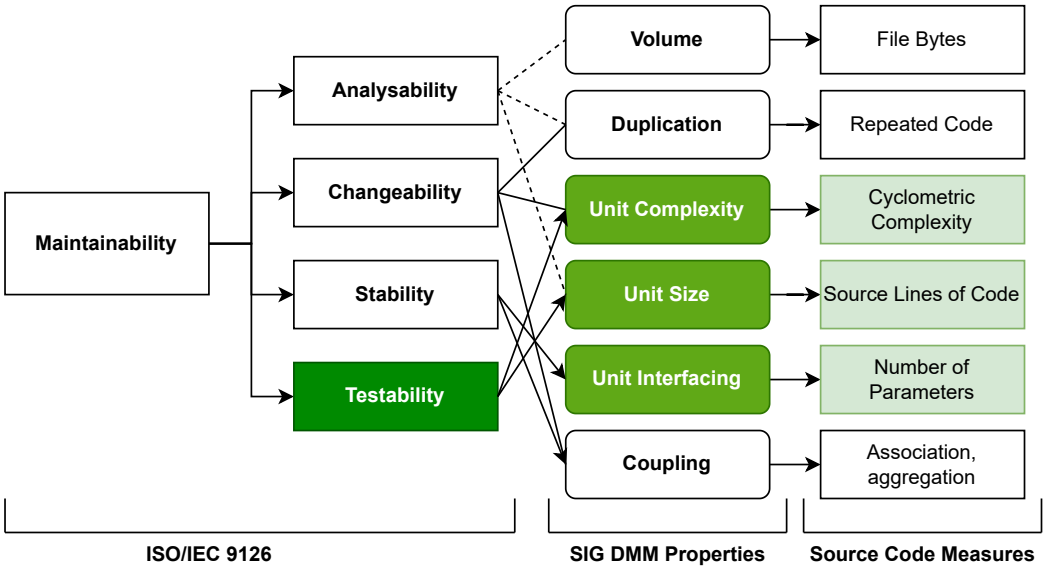


Figure 2.2: Relation between SIG Maintainability Model and source code measures and ISO/IEC 25010 quality characteristics

The first portion of the model, from left to right in Figure 2.2, is as prescribed by ISO/IEC 25010, followed by the SIG MM properties (metrics) and lastly the measures used to quantify the properties. The approach taken by Heitlager *et al.* is that the relationship between maintainability characteristic, metric and measure should be in a form of cause-influence-impact as shown in Figure 2.3. In this example, Testing (characteristic) is influenced by the size (property) and the size is measured by lines of code (measure). The lines of code impact the amount of testing done. That is, testability is a function of size, the larger a program is, the less testable it is - as larger programs are generally intertwined and often long with a lot of coupling between

objects, conversely, the smaller the program is, the more concise it is, hence, the easier it is to read, thus, the testability becomes easy as well.

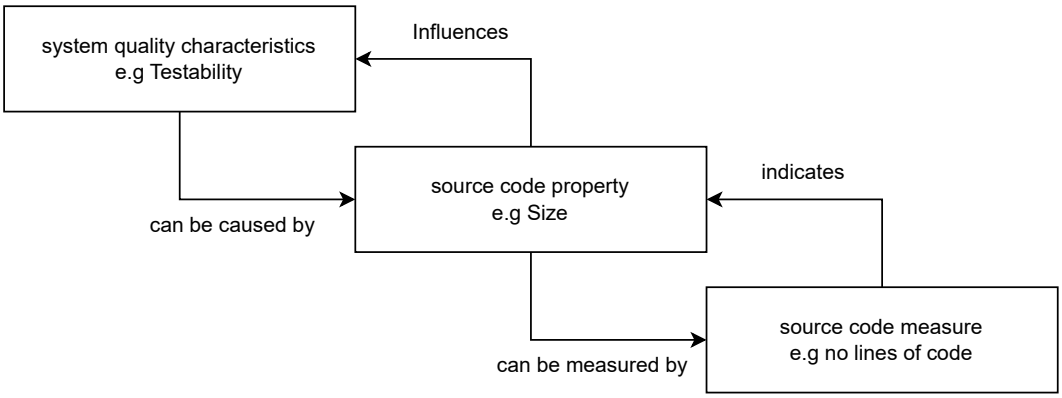


Figure 2.3: ISO/IEC 25010 - SIG Metrics - Measure Mapping Flow

In their following publications [62–64], Heitlager *et al.* provided approaches to aggregate the model measurements to enable system to system comparisons. The issue with SIG MM is that the primary measurement model is aggregated all the way to the entire system's code base. To tailor the SIG-MM for incremental changes, Di Biase *et al.* [65] refined the aggregation mechanism to focus on patches. It is worth mentioning that Di Biase *et al.* used the aggregation approaches by the original authors [62–64], yet, tailored it for small changes, and subsequently called this derivation the Delta Maintainability Model (DMM). Thus, the DMM is the same as the SIG-MM, except it is tailored (through aggregation) for measuring maintainability of fine-grained code changes such as commits. Formally, Di Biase *et al.* defined the DMM as "an index to measure the maintainability of code changes as the ratio between low risk code changes over the overall modified code changes". The idea behind DMM is to measure the ratio of change that is beneficial for maintainability.

Table 2.1: SIG MM properties with respective low risk thresholds

SIG MM Properties	Measure Examples	Threshold Value
Duplication	Commented Code, duplicated lines	-
Complexity	CC	5
Size	LOC	15
Interfacing	No of parameters	2
Coupling	DIT, Object Parameters	10

The DMM takes the measured numbers and maps these to a rating into comparison to benchmark of hundreds of industry systems [62,64]. The process to map the measures

to a rating is through a risk profile [62]. A risk profile is essentially a bin, denoted either low, medium, high and very-high. This bin groups metric outcomes using threshold values derived by statistical approaches discussed in [62], shown in Table 2.1. The risk profile is then used to identify the percentage of code in each risk category. Risk profiles can be computed for the individual metrics at unit level (method, class, file) and aggregated to the system level. In the end, the delta maintainability measure is the ratio of the total individual metric measures (at the level of concern) over the number of properties computed as expressed by Equation (2.2).

$$\text{DMM Score} = \frac{\sum \Delta\text{DMM Properties Scores}}{\text{No. Properties}} \tag{2.2}$$

2.1.2 Software Complexity

Studies concerning software complexity have been of interest for a number of decades due to the rapid growth of software production [66, 67]. A number of researchers and practitioners have investigated the concept of complexity and its associated implications, more especially from a quality point of view, not only in software engineering but in systems in general. As a result, a number of practitioners and researchers have proposed numerous definitions of complexity [68]. However, there is currently no consensus as to the exact definition of what complexity is, yet, what all researchers and practitioners agree upon are the consequences of higher complexity. According to Wade and Heydari [68], complexity cannot be attributed to a single source, hence, one cannot simply provide a concise definition without incorporating all aspects. They further argue that according to a research study they conducted, complexity can be categorised into three categories, namely behavioural complexity, structural complexity and constructive complexity and each category has its corresponding definition.

- Behavioural Complexity - the system is viewed from a black-box perspective, and complexity measures are given based on the input/output relationship of the system.
- Structural Complexity - complexity measures are given based on the architecture (structural design) of the system. A typical example of this is the size of the system, if proper design principles are followed, a system will be relatively sized accordingly (i.e size will follow a general trend), whereas in a poor design, the size is usually very high (i.e system size will deviate or be an outlier). This is

to say for "well architected" system, the size will be in accordance to the design and requirements, while for a poor design, the size generally will not follow the design.

- Constructive Complexity - complexity measures are based on difficulty in predicting future outputs from current outputs.

Wade and Heydari [68], concluded that complexity is an intrinsic property dependent upon the *context* and *observer*. According to them, a complex system is relative to the observer and context being analysed. That is, observer `A` can find a particular system difficult, while observer `B` finds the very same system relatively easy to understand. Similarly, a single system may be easy to understand in one context and may be complex in another context. Context was further broken down into three categories, *Computational, Interpretative* and *Paradigmatic*. Computational context speaks about the space-time domain aspect of complexity. Interpretative looks at how the same observer interprets the same system multiple times, as a user can have multiple interpretations of the same system. Lastly, Paradigmatic context analyses the interaction between the system and the observer. Based on the interaction with a system, a system may be complex or easy. Incorporating all these views, they argued that complexity can be defined as "the degree of difficulty in accurately predicting the behaviour of a system over time" [68].

Software complexity can be seen as one of the principle drivers of cost, maintainability and performance of software. As mentioned before, in most instances, there is no consensus or agreement of what the definition of software complexity is, yet a greater part of researchers and practitioners rely upon Zues's view of software complexity [66]. Zues defines software complexity as "the level of challenge in analysing, maintaining, testing, designing and modifying a software". Kearney *et al.* [67] provides an alternative definition (adopted from Basili [69]), where by they defined software complexity as "a measure of the resources expanded by a system while interacting with a piece of software to perform a given task. If the interacting system is a computer, then complexity is defined by the execution time and storage required to perform the computation. If the interacting system is a programmer, then complexity is defined as the difficulty of performing tasks such as coding, debugging, testing or modifying the software". Banker *et al.* present a similar definition with that of Zues, they say software complexity refers to a degree with which a system is difficult to comprehend, modify and test [70]. They further emphasise that this complexity is not about the complexity of the task the program or system is meant to perform, because two systems that are equivalent in functionality can differ greatly in their software

complexity. The causes of software complexity as identified by [71] are: lack of documentation, presence of dead code, presence of cloned code and presence of bugs/faults.

Outlined in all the different definitions provided, is that software complexity is present throughout the whole software development process and all the stages of the SDLC from planning to using the end product. Of relevance to this study are the definitions provided by Zeus and the IEEE. Software complexity analysis with metrics can provide meaningful insights to a programmer, tester and analyst, insights could include: information into software structure, identification of testing deficiencies, recommendations for program improvement, assessment of relative risky areas within the software system and identification of critical software components [72]. One of the popular ways of measuring software complexity is the decision flow graph proposed by McCabe [73].

McCabe's Cyclomatic Complexity

One of the most commonly used metrics to measure and quantify complexity is the cyclomatic complexity. The Cyclomatic Complexity (CC) metric first was introduced by Thomas J. McCabe in the late $70'S$ [73]. The metric is aimed at measuring the linearly independent path or circuits. Importantly, the CC metric ignores the size of individual code chucks but counts the number of decisions and control statements. McCabe's CC metric maps a program to a directed connected acyclic graph. The nodes of the graph denote the decisions and control statements of the program, while the edges of the graph denote the control paths which define the program logical flow. It goes without saying that a higher CC value is undesirable as it leads to higher program complexity. The CC is calculated as Equation (2.3):

$$v(G) = |E| - |N| + p \tag{2.3}$$

where $v(G)$ is the McCabe's cyclomatic number
E-Number of edges in the control graph
N-Number of nodes in the control graph
p-Number of connected components or subprograms (e.g for a single component or module, p =1)

The CC metric is also easy to compute and apply in analysing a program. Furthermore, it is used to estimate testing effort i.e it provides testers and developers with the

number of test cases required to test a program. It is often used as a quality and maintenance indicator. The issue with Cyclomatic Complexity (CC), however, is that it overlooks the program's data. Secondly, the metric considers a program to be complex if it has many number of decision and control statements, which may lead to misleading values for simple comparisons and decision structures. The following threshold values are suggested for the CC: $CC \in \{1, \ldots, 10\}$ the program is considered to be a simple program without much risk, $CC \in \{11, \ldots, 20\}$ considered more complex, moderate risk, $CC \in \{21, \ldots, 50\}$ very complex, high risk program and $CC \in \{50, \ldots, \infty\}$ very complex, very high risk and untestable [74]. The DMM model already incorporates the CC metric with threshold values based on empirical evidence [62], thus, the aforementioned thresholds are not applicable in this study.

2.1.3 Software Testing

Software testing is one the most critical phases within the SDLC, thus, it has its own life span termed the software testing life cycle (STLC) [75, 76]. In general, software testing is the activity of evaluating the functionality and correctness of a program with the intent of finding bugs [77], and *validating* and *verifying* whether the program meets its specification and requirements [22, 75, 76, 78]. Verification is concerned with the question *Are we building the system right?*, while validation asks *Are we building the right system?*. Testing of a program involves providing the program with a set of test inputs (test cases) and observing whether the *actual* output of the program matches the *expected* output [78]. Thus, testing cannot be seen as a process for finding bugs only but is a measure of other software quality metrics such as portability, maintainability, re-usability. Therefore, testing should reveal as many errors as possible in the Software Under Test (SUT), check whether it meets its requirements and also bring it to an acceptable level of quality [77]. Moreover, a complete software testing process should cover the SDLC in its entirety [78]. Riaz argues that the usage of testing for defect detection has a draw back because the software can only suggest the present of fault or errors and not their absence even when the testing itself is thorough and exhaustive [22]. He further argues that the problem associated with applying testing for reliability prediction is the input distribution used for selecting the test cases maybe fundamentally flawed.

Traditionally, the testing process has three fundamental approaches, which are black-box testing (also known as functional testing), white-box testing (also known as structural testing) and Grey-box testing. For a software to be "completely" tested, the first two approaches are to be applied together throughout the STLC in order

for the software to live up to the user's expectation in every level [22]. Riaz defines a testing strategy to be complete if it consists of testing at various points in the development process as defined by the traditional test-`V` model. Grey box and black box testing are outside the scope of this research.

The white-box approach include designing a test such that every line of source code is executed at least once, this process is known as Coverage-based testing [78], which is a sub-group of white-box testing. White-box testing cannot be done without modification of the program source code. Modifications to the program requires changing of values to force different execution paths or generation of a full range of inputs to test a particular function. The intention in white-box testing is to provide all possible and feasible flow of control paths through a subprogram traversed whilst the software is going through a test. It is important to note that this is not to say all the lines of code in the subprogram will be executed as that is impractical. The most common method of performing white-box testing is through unit testing, quantified by code coverage.

Code Coverage

Code coverage is the extent to which a structure has been exercised during testing [79,80]. Coverage can be thought of as a measure used to gain insights as to how much of the code-base is actually tested within the test suite [81]. According to Miller and Maloney, loosely translated, code coverage can be viewed as a the percentage of code structure or artefacts which is executed or covered at least once by a test suite [82,83]. There are numerous ways of calculating coverage, the most popular being calculating statement coverage, Equation (2.4). Statement Coverage (SC) measures the ratio of the number of statements executed by test suite over the total number of statements.

$$SC = \frac{\text{No. Executed Statements}}{\text{Total No. Statements}} \times 100\% \tag{2.4}$$

2.1.4 Change Risk-Anti Patterns (CRAP) Metric

The Change Risk Analysis and Predictions metric, formally known as CRAP, was introduced by Alberto Savoia in collaboration with Bob Evans in his 2007 blog posts [42, 43]. Alberto argued that the CRAP metric was developed as an index to measure and predict the risk and effort associated with modifying existing programs i.e maintaining and enhancing existing code. The author further noted that the CRAP metric of a program should be viewed from the lenses the developers were

not involved in authoring of the program, especially if they have to maintain the code. Alberto and Evans expressed the CRAP metric of a method, m, as a function of complexity and the coverage of the method, as shown by the expression given by Equation (2.5).

$$\text{CRAP(m)} = \text{comp}^2\text{(m)} \times \left(1 - \frac{\text{cov(m)}}{100}\right)^3 + \text{comp(m)} \tag{2.5}$$

Comp(m) denotes the McCabe Cyclomatic Complexity whilst *Cov(m)* is the percentage coverage of method m. Low values of the CRAP metric would typically indicate a well covered and less complex method. A high CRAP value means the change is complex and the method is not well tested by automated tests. A method is considered CRAPpy if the measure is greater than `30` [84]. For a fully covered method i.e 100% test coverage, the equation reduces to simply the complexity of the method and for a method not tested at all, the equation reduces to the square of the complexity plus the complexity as shown by Equation (2.6). Both edge cases show that the CRAP metric is tightly coupled to complexity, which is at the core of the SIG-MM as previously indicated.

$$\text{CRAP(m)} = \begin{cases} \text{comp(m)}, & \text{if cov(m)} == 100\% \\ \text{comp}^2\text{(m)} + \text{comp(m)}, & \text{cov(m)} == 0\% \end{cases} \tag{2.6}$$

The authors further note that to decrease the CRAP index value, developers either have to refactor code thereby reducing the complexity or writing automated unit tests or both, as famously quoted from his response where he said "*If you write code, write tests for it.*" when asked in a Question Answer (QA) session [85]. While not widely popular or widely investigated as a metric, the adoption of the CRAP metric in commercial tools like NDepend suggests its utility in code analysis [86].

2.2 Exploring Software Quality Evolution: An Empirical Review

As hard as it may be to succinctly define and measure software quality, numerous studies have been conducted to identify, define, measure and predict a software program's life-cycle from a quality point of view. The following are prior studies

conducted related to software quality in relation to testing, maintainability and quality evolution:

2.2.1 Coverage and Defect Detection Relationship

In their 2019 study, Vard and Miroslaw [87] explored the correlation between unit test coverage and post-unit-test defects using a substantially large proprietary software (i.e 2 million LoC), involving over 150 engineers at Ericsson. Contrary to expectations, their findings suggested that higher unit test coverage did not necessarily contribute to defect-free software. The research revealed a robust correlation between lines of code (LoC) and defects, suggesting that larger systems are more prone to defects. Despite this, the researchers acknowledged a modest decrease in defects with an increase in test coverage, emphasising a nuanced relationship between coverage and software quality. Inozemtseva and Holmes [88], generated 3100 mutation test cases across five Java systems to analyse the connection between coverage and defects. The research concluded that the correlation between coverage and fault detection effectiveness is moderate, leaning towards being low. Consequently, the authors suggest that relying solely on coverage as a quality target is not advisable due to this moderate-to-low correlation. Mockus *et al.* [89] examined two proprietary systems, one implemented in Java and the other in C/C++. Their research revealed a correlation between test coverage and a reduction in the number of field bugs. Additionally, they observed that test effort escalates exponentially with test coverage, while the decrease in customer problems correlates linearly with test coverage. Briand and Pfahl [90] conducted a study involving twelve versions of a system with the aim of identifying a causal relationship between coverage and defect prediction or identification. Surprisingly, their findings indicated an absence of any discernible link between test coverage and defect reduction.

Kochhar *et al.* [91] analysed Apache's HTTPClient and Mozilla's Rhino with 67 and 92 bugs respectively to assess the efficacy of coverage as a metric for bug identification. Their results indicated a moderate to strong correlation for statement and branch coverage in both projects. In a separate study involving 100 Java open-source systems, Kochhar *et al.* [92] found no discernible correlation between coverage and post-release bugs. Cai and Lyu [93] investigated five testing approaches, termed as testing profiles: random, functional, whole-set, normal, and exceptional testing, to evaluate coverage as an estimator of defect detection. The findings indicated that, overall, coverage possesses a moderate capability in bug detection. Particularly, for exceptional test cases (exceptional profile), coverage demonstrated a significant advantage as an

estimator for fault detection. In contrast, for normal cases (normal profile), coverage tended to be less effective in identifying bugs. Gregory *et al.* [94] examined five proprietary systems, concluding that coverage serves as a poor estimator in defect detection. Gopinath *et al.* identified a correlation between statement and branch coverage with number mutants (intentionally altered code to introduce bugs) killed in hundreds of Java open source systems [95].

2.2.2 Software Evolution

Purushothaman and Perry [96] investigated the evolution of a commercial system written in C to discern the relationship between one-line changes and the introduction of defects. Their analysis suggested that one-line changes are improbable contributors to bug introduction. Moreover, over half of code changes involve modifications to fewer than ten lines. De Oliveira *et al.* [97] examined the applicability of four of Lehman's laws (continuous change, increasing complexity, declining quality, and continuing growth) within the context of Software Product Lines (SPL). Their study focused on a system comprising 45 modules and approximately 71 bugs. The findings indicated verification of the first three laws—continuous change, increasing complexity, and declining quality—while the law of continuous growth could not be substantiated. Godfrey and Tu [33], examined 34 stable releases and 62 development releases of the Linux kernel as compressed files. The analysis revealed a linear growth pattern at the system level. Upon closer inspection into the modules, it was observed that only a select few, particularly drivers, were experiencing substantial growth, thereby contributing to the overall system expansion. In contrast, the majority of modules demonstrated a consistent size over time.

Complexity Evolution

Vard *et al.* [98] conducted a study at Ericsson and Volvo Group Truck Technology to observe the evolution of complexity in a software system in an agile/lean environment. The study indicated a continuous increase in complexity over time, measured by metrics such as Cyclomatic Complexity (CC) and Size (Lines of Code - LoC). The authors emphasised that CC serves as a valuable metric for monitoring the evolution of a software product. In their sequel study, Vard *et al.* [99], emphasised the significance of concentrating on small code units, such as functions and classes, based on an analysis of five releases from two proprietary systems. They suggest that focusing on these smaller units could facilitate the identification of problematic files and functions,

providing opportunities for manual assessment and improvement. Huihui *et al.* [100] conducted a comprehensive study on the evolution of McCabe cyclomatic complexity, examining both class and system levels. Their findings revealed that class-level complexity exhibited minimal growth over time, whereas system-level complexity displayed a linear increase. Additionally, the authors observed that the growth rate of system size, measured by Lines of Code (LoC), was slower compared to the growth rate of cyclomatic complexity.

Testing Evolution

Akour [101] *et al.* studied three Java open source projects, each with ten versions, to explore the evolution of test suite effectiveness. Test suite effectiveness was defined as the ability to detect faults within the Software Under Test (SUT). Utilising lines of code (LoC) as a measure of evolution, the study revealed a consistent positive correlation between test suite size and its effectiveness. However, the correlation between test suite effectiveness and coverage varied, ranging from weak to strong across the investigated projects. Oosterwaal *et al.* [102, 103] designed a tool named OPERIAS with the aim of comprehending the code coverage difference between fine-grained source code versions (commits) with a special interest in the influence of pull requests on code coverage. Their argument is that, although pull requests do not contain new code, they consist of a collection of commits, and as a result, they expect high test coverage in pull requests. Employing the `linux diff` algorithm, OPERIAS compares two commits, identifying modified or newly added lines or files. The study focused on three open-source projects: JUnit, Wire, and Buckitt. The results revealed that, among the projects investigated, two experienced an increase in code coverage as a result of pull requests. That is, commits which were performed directly on the branch had very low coverage whilst code merged through pull requests had higher coverage, thus, increasing the overall coverage.

Marinescu *et al.* [17] pioneered the exploration of "patch coverage" in small incremental changes, as previously stated in Section 1.2.1, and subsequently established a formal definition for this concept. Marinescu *et al.* defined patch coverage as the number of executed lines added in a patch over the total executable lines in that patch. To investigate the co-evolution of test, code, and coverage, the authors developed the COVRIG tool and conducted a study involving six open-source software (OSS) projects. Their findings revealed that patches were either fully covered or not covered at all, with engineers seldom opting for partial coverage of new code. Additionally, the study observed that testing appeared to be a *phased* activity for five out of

six projects, occurring intermittently after extended periods of development. This pattern was observed in all except one project. Hilton *et al.* [18] expanded upon the research conducted by Marinescu *et al.*, introducing an investigation into how covered lines transition between patches and the overall impact of patch coverage. Similar to Marinescu *et al.*, they observed a pattern where patches were either fully covered or not covered at all. Notably, Hilton *et al.* identified an intriguing phenomenon termed "flipping", where some patches led to changes in the coverage status of lines, transitioning from previously covered to uncovered in the new modification. This is fascinating because, under ideal conditions, a coverage status of a line should be independent of the dynamic behaviour of the program and/or state of other lines in the program. Therefore, lines transitioning between commits could reveal some form of intertwined dependencies between lines. The flipping of lines has one or more (not limited) of the following implications:

- Inadequate Testing - Flipping of lines indicate that a seemingly innocuous modification can have a significant impact on test coverage. This raises questions about the robustness of test suites and the effectiveness of testing methodologies.
- Potentially hidden dependencies - Flipping suggests the presence of hidden dependencies or interactions within the code. It implies that modifying one part of the code can unexpectedly affect another, even if they appear unrelated. Understanding these hidden dependencies is crucial for maintaining code stability and preventing unintended consequences during development.
- Potential Risk to Code Quality - Flipping challenges the accuracy and effectiveness of code analysis tools. If a change leads to a line transitioning from covered to uncovered, it may indicate limitations in static analysis tools or highlight the need for more sophisticated techniques to identify potential issues in the code.
- Regression Risks - Developers often rely on code coverage metrics to assess the risk of introducing regressions. Flipping suggests that changes considered safe based on previous coverage analysis may still introduce unexpected issues. This challenges the conventional understanding of how code modifications impact regression risks.
- Refactoring Challenges - For developers involved in refactoring or code restructuring, flipping highlights the need for caution. Even seemingly minor changes can lead to uncovered lines, requiring a careful examination of the code-base to ensure that the refactoring process does not inadvertently reduce test coverage.

- Documentation and Understanding - Flipping underscores the importance of comprehensive documentation and understanding of code behaviour. Developers may need to document and communicate potential risks associated with certain modifications, ensuring that the knowledge is shared among team members to prevent unintended consequences.

Additionally, the study concluded that patch coverage did not significantly impact the overall system coverage, emphasising the importance of treating patches as independent entities.

Quality Evolution

In their study, Molnar and Motogna [104] analysed forty-one versions of jEdit, critiquing the MI's (Section 2.1.1) applicability in object-oriented systems. They opted for the well known Chidamber & Kemerer (CK) metric suite, developing a maintainability model that outperformed MI in detecting even subtle changes. Ganpati *et al.* [105] examined 50 versions of PHP over seven years, noting a decreasing MI over successive releases. They explored MI's relationship with LOC, CC, and Halstead volume, finding strong inverse relationships with size and complexity metrics.

A recent 2023 study was conducted by Heričko [58] where they investigated five MI variants in forty-five Java object-oriented systems, highlighting that the choice of variant could influence maintainability perception, despite the strong overall positive correlations amongst the maintainability variants. These findings are significant in that: the statement highlights the significance of findings related to the lack of consistency in defining MI as critiqued by [61]. This inconsistency raises a concern about the ambiguity in understanding how maintainability should be measured. In other words, if there is no clear and consistent definition of maintainability, it becomes challenging to establish standardised criteria or metrics for assessing and measuring it. This ambiguity in the definition of maintainability can lead to varied interpretations and methodologies in different contexts, potentially hindering effective communication and comparison of results across studies or projects. Researchers and practitioners in the field of maintainability need a clear and agreed-upon definition to ensure that they are assessing the same aspects and characteristics when evaluating the maintainability of systems or software. The reference to Heitlager [61] suggests that there has been criticism or scrutiny of the existing definitions of maintainability, prompting a recognition of the need for a more consistent and well-defined conceptualisation in order to advance the field and enhance the reliability of maintainability assessments.

Kapllani [106] analysed twelve jEdit versions and five other projects, identifying three maintainability evolution patterns. Interestingly, they found a positive correlation between LOC and MI, contrary to traditional expectations.

In another study, Molnar and Motogna [107] used thirty-eight FreeMind, forty-three jEdit, and twenty-six TuxGuitar versions developed over eighteen years. They compared the traditional MI index, technical debt ratio, and the ARiSA Compendium-based quality model [108]. The ARiSA quality model is similar to the SIG-MM in that it maps software metrics to the IS0/IEC 25010 software quality characteristics. It then assigns a strength and influence to each metric-characteristic pair based on the measure. The influence, can either be a plus (positive/direct influence), minus (negative/inverse influence) or strong (double plus or negative). The sign (plus or minus) indicate the influence (positive or negative) while the number of signs (single or double plus/minus) indicate the strength. For example, a high value of LoC will result in strong negative influence on *analyse-ability* (i.e double negatives), meaning it is very difficult to analyse large code-bases. Molnar and Motogna observed an overall improvement in the maintainability with constant refactoring effort. Additionally, they noted that maintainability and system size can co-evolve independently, while class count and LoC generally co-evolve. Moser *et al.* [109] evaluated Extreme Programming (XP) in a Java commercial software, using MI, LOC, and CK metrics. Daily data showed steady maintainability growth, while release data indicated a rapid MI decrease in initial iterations, stabilising in later releases.

2.2.3 SIG Maintainability Model

After reviewing the original MI index proposed by Paul Oman and Jack Hagemeister, concerns were raised by Kuipers *et al.* [60] and Heitlager *et al.* [61] regarding its ability to accurately measure program maintainability (refer to Section 2.1.1). Consequently, the authors developed a new model, known as the SIG Maintainability Model (SIG MM), which correlates ISO/IEC-25010 quality characteristics with metrics and measures. The SIG MM has since been found (through empirical research) useful in predicting the maintainability of system. Below are several use cases where the SIG MM has been empirical evaluated:

Wijnmaalen used the 2018 variant of the SIG model which contains more properties than the first model. The findings showed a majority of systems scored lower in maintainability rating. The overall maintainability correlation to defect resolution time dropped significantly while the 5-property original model correlated positively

with defect resolution time as presented in [110]. Strečanský *et al* [111] compared the effectiveness of MI, SIG-MM, and Software Quality Assessment based on Lifecycle Expectations (SQALE) [112, 113] in evaluating the evolution of Technical Debt (TD) using twenty-one Python packages. SQALE is a maintainability model much like the SIG-MM, except it is focused on assessing quality and maintainability of the system holistically than specific changes, as such, it considers attributes as architectural designs of the system [112, 113].

Their results indicated that the SIG MM and MI models were in sync with regards to how TD evolves over time and TD was found to increase as a function of time. Di Biase *et al.* [65] were the first to apply, at least empirically, the SIG-MM model at the lowest developmental changes i.e commit and subsequently derived a model from the SIG-MM called Delta Maintainability Model (DMM). They described the DMM as the "proportion of a change that is beneficial for maintainability". Di Biase *et al.* conducted a study with four open source and four closed source systems with a total of 3000 commits. They report that the DMM can be used in ranking commits according to the DMM scores, thus, enabling developers to perform root cause analyses and diverting effort (testing, refactoring etc.) to commits with higher technical debt. Pavlo [114] *et al.* extended the DMM model provided by [65] for specifically object oriented systems by including CK metric suite.

Luijten [110, 115] studied the effectiveness of SIG-MM with respect to time taken to resolve an issue, referred as issue resolution, measured by issues of type *defect* and *enhancement* in GitHub. The study shows that for all issues of defect, the SIG-MM properties led to an improved time to resolution whereas for enhancements, all but unit interface has no correlation to time resolution. In a similar vein, a study was conducted by Aigerim *et al.* [116] but they used proprietary projects instead of open source projects. Aigerim *et al.* used three commercial projects and only one correlated maintainability with faster issue resolution. In Wijnmaalen [117], repeated a study performed by [110, 115] to evaluate the effectiveness of the SIG model in predicting software quality. Turetken [118] proposes the SIG-MM to model a business process models due to its pragmatic approach on mapping source code properties to ISO/IEC 25010 characteristics. Malavolta *et al.* [119] mined 434 real android mobile applications to study the persistence and evolution of maintainability issues using the Software Analysis Toolkit (SAT), which provides an abstraction layer to the SIG-MM i.e application that hides the intricate details of how the metrics are mapped and how the measures are computed. Through mining weekly data, the study found that code duplication is the most prevalent issue followed by size amongst the apps studied.

In the context of exploring fine-grained changes in the study's background, the concerns mentioned Section 2.1.1 about the original MI index led to the adoption of the SIG MM. This decision was influenced by the belief that the variant of the SIG-MM, DMM, which is formulated for commit changes, DMM was better suited for this investigation as it focus on fine-grained changes for program maintainability.

2.3 Chapter Summary

An introduction and definition of the concepts that are investigated is provided by reviewing existing work in the field. A large effort in prior research has primarily focused on understanding software evolution dynamics in the context of releases, which provide snapshots after a relatively lengthy development process. However, snapshots are limited for developers involved in the intricacies of the SDLC. This is because developers deal with the smallest level of granularity possible, thus, analysing stable snapshots may potentially obscure the details of the individual commits. Moreover, analysing releases limits one's ability to understand the rationale for code changes and evolution of development decisions over time. On the other hand, analysing commits gives the developer context regarding the changes and allows tracking of impact as it pertains to individual developer's contribution. Unfortunately, there are limited studies examining commits from a software maintainability and testing perspective. Hence, this study undertakes an investigation into dynamics of software quality, as seen from maintainability and testing at a commit level. A variant of code coverage, known as patch coverage is investigated, summarised in Table 2.2.

Table 2.2: Summary of Covered concepts

Concept	**Definition**
Code coverage	ratio of lines covered/executed during test run over "Total" lines of the code base.
Patch coverage	ratio of how many of the "modified" lines in the commit are covered/executed during test run over "total modified" lines of that commit.
CRAP metric	measure of how tested or how complex a newly added code based is.
DMM	A measure of how maintainable a commit/patch is.

3

Methodology

This chapter outlines the research methodology for the empirical study, covering definitions and descriptions of collected metrics, projects selection criteria, and data integrity threat mitigation procedure. It elaborates on the use of Exponential Weighted Moving Average (EMA) for data cleaning. Moreover, it delves into the analysis approach of the correlation strength among metrics using Spearman and Kendall's definitions.

3.1 Research Approach

The primary objective of this report is to empirically assess the evolution of testing, specifically at the lowest level (unit testing), and maintainability evolution, focusing on fine-grained code changes. Prior research on code coverage evolution often involved resource-intensive and laborious processes, such as downloading, compiling, and running the source code for each release version [3, 7, 17]. Hilton *et al.* [18] pioneered an approach combining source code download and execution with the utilisation of coverage service tools like Coveralls. Downloading and compiling projects pose challenges due to version dependencies and resource requirements, leading to build failures for different release versions [3]. Marinescu *et al.* [17] attempted to address this using virtualisation, but acknowledged its continued resource intensity. In this study, a mixed approach is adopted, as illustrated in Figure 3.1. Coverage information is collected from coverage service and containers are employed for source code metrics computations per patch. This approach eliminates manual processes such as cloning, compiling, and building applications for patch statistic measurement. Instead, reliance is placed on coverage service to deliver coverage measures, with only the remaining metrics computed independently. Popular coverage services, Codecov [120] and

Coveralls [121], offer persistent build history and accessible web APIs for querying repository statistics. To empirically evaluate testing and maintainability evolution at a patch level, the build history is collected, and metric values are computed for each commit.

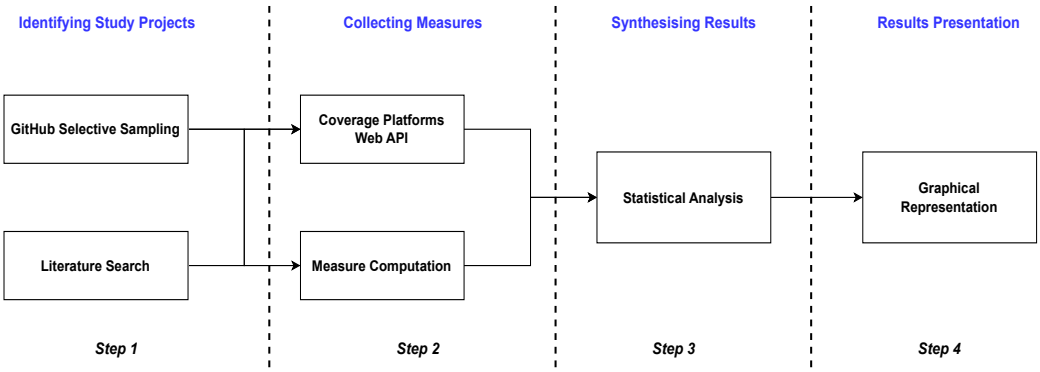


Figure 3.1: Overview Research Methodology

3.1.1 Compute Infrastructure

As previously mentioned, containers were utilised to collect and compute relevant metrics. A virtual machine hosted on the Microsoft Azure cloud running on Ubuntu-20.04 served as the host operating system. The advantages of containerisation, as emphasised by Marinescu *et al.*, are well-documented. The Docker containerisation engine [122], was chosen for its popularity and efficiency. The Docker engine was employed to create and deploy multiple containers within a virtual machine hosted in the cloud, ensuring continuous operation without interruption. To ensure data collection from the coverage services in parallel, two containers are deployed with each coverage service designated a container to collect and compute metrics for projects residing within that coverage service. A third container was used for cross coverage service projects i.e projects which had their build history in both coverage services. This was typically for projects which were migrated from one coverage service to the other. The overall architecture for a single container is depicted by Figure 3.2. A Docker volume was then used to map the results directory in the containers to a directory in the host machine to persist the data. Appendix D.1 provides reference to the `Dockerfile` used for image building, and Appendix D.2 outlines the process for instantiating a running instance of the image, which will collect and compute the metrics.

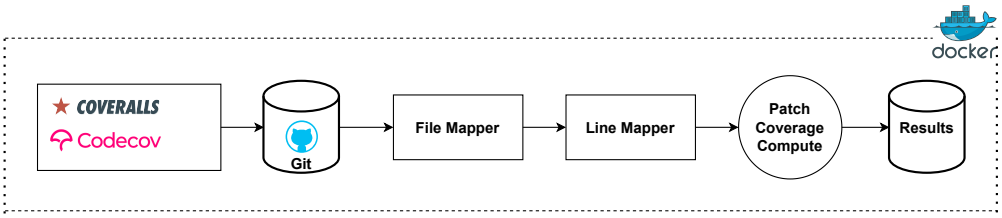


Figure 3.2: Compute Infrastructure Architecture

3.2 Overview of the Dataset

The dataset is a composite, consisting of data gathered from coverage services and computed data. Using the coverage services APIs, each build provides system coverage, Lines of Code (LoC), the build timestamp, and the build commit hash. The repository is then pulled, and for each build commit hash, the repository is checked out to the corresponding commit. DMM values, patch coverage, patch size, the number of modified files (files touched) in the patch, and delta CRAP metric are determined for that specific commit. This process continues until all commits that triggered a build are exhausted.

While the DMM comprises of five properties (see Section 2.1.1), this report focuses on unit interface, size, and complexity (highlighted green in Figure 2.2). This decision is rooted in the consideration that measuring and computing these metrics is quite straightforward and easy to perform. Moreover, there is no ambiguity in how these are defined and measured across different languages. The overall DMM score is the average of the three chosen properties, indicating the percentage of good change over the overall change. If a commit modifies only non-source files (referred to as configuration files), all source code-specific measures, such as patch coverage, DMM, and delta CRAP metric, are set to zero for that commit. The setting of these measures to zero was with the assumption that modifying non-source files should not affect any patch statistics except for patch size of configuration files. However, this was shown to be a false assumption, as modifying scripts such as `Makefile` which configure covered directories and/or file(s) does indeed impact patch coverage (refer to Chapter 4 for results and discussion).

The individual DMM measures (size, unit interface, and complexity) represent the percentage of good change over the overall change, each in its own context. For instance, DMM complexity indicates the ratio of complexity added over the overall complexity. All DMM values range between zero to one. DMM complexity is utilised to compute delta CRAP metric, implying that, in this study, delta CRAP metric

reflects the proportion of code “crappiness” added over the overall codebase delta CRAP metric.

3.3 Data Collection

Empirical studies in software engineering rely on data from software repositories, particularly within the open-source space like GitHub [123]. GitHub serves as a central data hub due to its openness and licensing nature in contrast to proprietary source control management system (SCM) [124]. Kalliamvakou *et al.* [125], highlights GitHub’s advantages and disadvantages, emphasising caution in mining data, given the prevalence of personal and inactive repositories that may be considered "toy" projects. The issue of “toy” projects is discussed in Section 3.3.1. Furthermore, a study by Tutko *et al.* [126] reveals that over 33% of journal papers lack information on data mining methods and study reproducibility. To address these issues, a rigorous approach for selecting projects and mining the data is detailed below:

3.3.1 Project Selection

A quantitative research approach is employed. The emphasis is placed on the integrity of the projects mined and metrics extracted from the project as a representative sample of open source projects. The quantitative nature lies on the number of projects and number of builds examined for each project over a significant period of time. As pointed out by Hilton *et al.* [18] and Zaidman *et al.* [3], downloading, compiling/building and running projects to collect the metrics is sometimes non-trivial as some projects fail due to dependency issues. Thus, two approaches are adopted to gather a set of repositories:

Prior Research

The initial strategy involved identifying recent research on software evolution and testing experiments, notably by Hilton *et al.* [18]. Projects studied by Hilton *et al.* were initially sourced from the Coveralls coverage service. Verification revealed that out of the forty-seven projects, seventeen were no longer available in either Coveralls or Codecov. Further investigation showed that four projects had migrated from Coveralls to Codecov. Among the remaining twenty-six projects, seven had build history in both coverage service, with most projects disabled in Coveralls but still

active in Codecov. To prioritise long-term data trends, projects with fewer than ten builds were discarded, with the aim being on datasets demonstrating significant periods of development. This resulted in the exclusion of four projects, thus, only twenty-two out of the forty-seven are studied.

Criteria-Guided Project Selection

The second approach was the manual crawling of GitHub for potential projects. To identify possible projects, the strategy employed by Pinto *et al.* [127] is adopted. That is, repositories that are 1) popular (stars, forks, number of contributors etc.), 2) actively maintained and 3) within top rated open source organisations like the Apache Foundation, Eclipse, Oracle etc. From an end-user point of view, these metrics indicate that the project is in "good" standing with the open source community. The organisation's repositories are manually inspected to identify README files containing coverage badges (for example Codecov 

and Coveralls 

). The badges indicate the latest overall system coverage per project and the respective coverage service. The coverage percentage is used to select projects with diverse coverage from projects with little coverage to fully covered projects. Manually inspecting README files is effective for organisations with few repositories but inefficient for those with thousands, as it could take quite a while to go through all repositories. Fortunately, Codecov provides an organisation endpoint where one can pass the organisation name and it will return all active and inactive repositories linked to that organisation. Coveralls on the other hand does not provide an organisation endpoint. Thus, a web scraper in Python was built to perform web scraping for all repositories in an organisations in Coveralls. The process, therefore, was to manually to identify the top 10 organisation names, then, scrape for existing individual repositories in Coveralls belonging to that organisation.

For each repository in the organisation that had builds, the date at which the repository was last built is observed. The minimum latest build date is chosen to be (2021) - recent yet far back enough to contain a relative quantifiable and meaningful build history. Moreover, no more than ten projects per organisation were picked. This was done to ensure the data is never skewed towards one organisation as some organisations may have rules about contributing and some may not. Moreover, the total number of builds and time range are observed. In an effort to strive for balance between the evolution (time) aspect and quantity (number of builds), a mixed approach is adopted. The minimum time difference between first build and latest build is set to 6 months for projects with the less than hundred builds and three

months for projects with greater than hundred builds.

3.4 Data Cleaning and Processing

3.4.1 Overview of data issues

The data cleaning and processing is based on known data issues such as data inconsistencies, gaps and misrepresented data. Thus, a data cleaning approach is adopted to cater for the following:

Development Workflow

There are various branching strategies, such as feature branch workflow, Gitflow workflow, trunk-based development, and more. It is hypothesised that each of these branching approaches would influence testing and code quality differently. To ensure consistency, data is extracted from all branches. The rationale for mining all branches is the belief that development in all branches contributes to the overall coverage. Also, the assessment is to focus on developers' incremental contribution towards building a system via a series of patches on all branches. These increments, if, incorporated into the trunk (main branch) accurately portray how the system is being built. Therefore, no assumptions are made about the specific branching strategy.

Missing Commits

One of the challenges posed by the various branching workflows discussed above is the issue of dangling commits or short-lived branches, leading to the unavailability of certain commits. When mining the commits which triggered the build, there were instances where attempting to check out the code base to the corresponding commit hash would result in a fatal Git error indicating a non-existing commit hash, as shown below. In Git, a dangling commit refers to a commit that is not referenced by any branch or tag. This situation typically arises when a commit is deleted from a branch due to previous actions such as rebase or reset, or when a branch itself is deleted.

```
SHA b'1c4ba933cefb2bd3a9ab587bf9dff152e7091137' could not be resolved, git returned: b'1c4ba933cefb2bd3a9ab587bf9dff152e7091137 missing'
```

Dangling commits are most prevalent in pull-request-based development or from short-lived branches and/or deleted commits. To address the issue of dangling commits, all commits resulting in a missing commit hash are skipped. This will be discussed in the later chapters.

Additionally, it was observed that Coveralls had a considerable number of builds where the commit hash key was returned as `null`, while other key statistics were represented correctly. Since a `null` commit hash cannot be checked out, these builds are ignored.

Data Inconsistencies

The gaps and inconsistencies is the primary source of data integrity. Various issues were noted with the extracted data.

- **CronJobs/Cron-builds** - These define regular builds on specific branches and dates. CronJobs, which typically build the last commit on a specific branch - since they are not triggered neither by a pull request or a push, contribute no value to coverage and patch statistics because they are merely scheduled jobs which trigger builds based on the latest commit. To exclude Cronjobs, unique commit hashes are sought in the entire build history to remove duplicate builds. A slight variation in coverage results for the same commit hash was identified, attributed to the dynamic nature of test coverage extraction, as noted by the Coveralls team [128]. For duplicate data, builds were extracted based on the first build with the same commit hash, using date/timestamp as an identifier i.e earliest date for similar commit hash.
- **Security Scanning** - GitHub now includes security scanning builds and branches as part of repositories. Dependabot [129] is GitHub's automatic dependency scanner, creating its own branch based on the default branch to scan for vulnerabilities. Given that this isn't considered an "actual" contribution, Dependabot's branches are filtered out in the analysis.
- **Sudden Coverage Drop** - The most common problem(s) in the CI/CD space are either build failures or failure of the coverage service API to retrieve data from a build, due to issues such as the services reported to be down and/or repository miss-configurations. This often results in coverage spiking and/or incorrectly reported. A number of issues have been opened in the help portal(s) of coverage service, for example issues [130, 131] reported a sudden drop in

coverage from a previous build which was well covered in Codecov. Coveralls, is no exception to this (see issues [132–135]). In an exchange with the Codecov team [136], it was revealed that sometimes build failures can affect how the API and subsequently the user interface may retrieve and display data from the build. For example, the Apache/libcloud build details as retrieved from Coveralls API in Listing 5 shows two consecutive builds, the first one returns correct data yet the second, return 0% coverage. Such a response is highly unlikely and is flagged as a corrupted build in order to maintain the data integrity of the study. Due to these improbable circumstances, a 30% maximum coverage drop in coverage drop is arbitrarily chosen to filter out the noise in the data. A drastic change (such as a potential rewrite or addition of a large package) would have to be added to the code-base for it coverage to drop beyond 30%. Even in that instance, the code added would have to be entirely uncovered, which is of course possible but improbable. The Exponential weighted Moving Average (EMA) is chosen to clean and smooth the data from such outliers. In contrast to other moving averages like the Simple Moving Average (SMA), the Exponential Moving Average (EMA) gives weight to both past data and the forward trajectory to calculate the next data point. In the context of code coverage, its value lies in the observation that code coverage tends to exhibit stability or gradual increases over time. Take the example of Apache Gobblin project (henceforth referred to as apache/gobblin) in Figure 3.3, there are numerous points where the coverage is picked-up as zero. Upon further investigation, it was discovered that these commits underwent minimal changes, indicating that they could not have been the sole cause of the significant decrease in coverage from 45% to 0%.

Applying a 30 points EMA to the appache/gobblin data yields **Bottom** graph in Figure 3.3, which was deemed to be a plausible enough drop in coverage.

3.5 Metrics and Measures

3.5.1 Coverage Service APIs

Codecov API

The Codecov CI coverage service's v2 API includes the `/repos/{repo_name}/report` endpoint, facilitating examination of commit-by-commit code changes by supporting a query parameter for the commit hash. This allows users to retrieve the repository's

```
[{"repo_name": "apache/libcloud",
      "branch": null,
      "commit_sha": null,
      "coverage_change": 0.0,
      "covered_percent": 0.0,
      "covered_lines": 0,
      "missed_lines": 0,
      "relevant_lines": 0,
      "covered_branches": 0,
      },
    {"repo_name": "apache/libcloud",
      "branch": "trunk",
      "commit_sha": "af264fecea1adc8ded707094854a3c64790c3285",
      "coverage_change": 36.8,
      "covered_percent": 36.791,
      "covered_lines": 18742,
      "missed_lines": 32199,
      "relevant_lines": 50941,
      "covered_branches": 0,
      },
    ... # excluded for brevity
]
```

Listing 5: Apache/Libcloud Corrupt Data in Coveralls

build state at a specific point in time. Additionally, Codecov, also offers patch coverage as part of their web API. For comparison purposes this data point is also queried. A representation of a typical response is illustrated in Listing 6.

The property of interest is the `diff` property, represented as an array that captures the different changes for a given commit, akin to the `Git diff`, yet, from a coverage standpoint. Specifically, the element at position six within this array contains the patch coverage. For instance, in Listing 6, the patch coverage for this commit is said to be 85.714%. The sample response (Listing 6) is the expected output where a change has been made to source files. However, for non-source files changes or no-change builds, the API is designed to return an array of zeroes for the `diff` property. Despite this, certain scenarios resulted in the `diff` property returning one

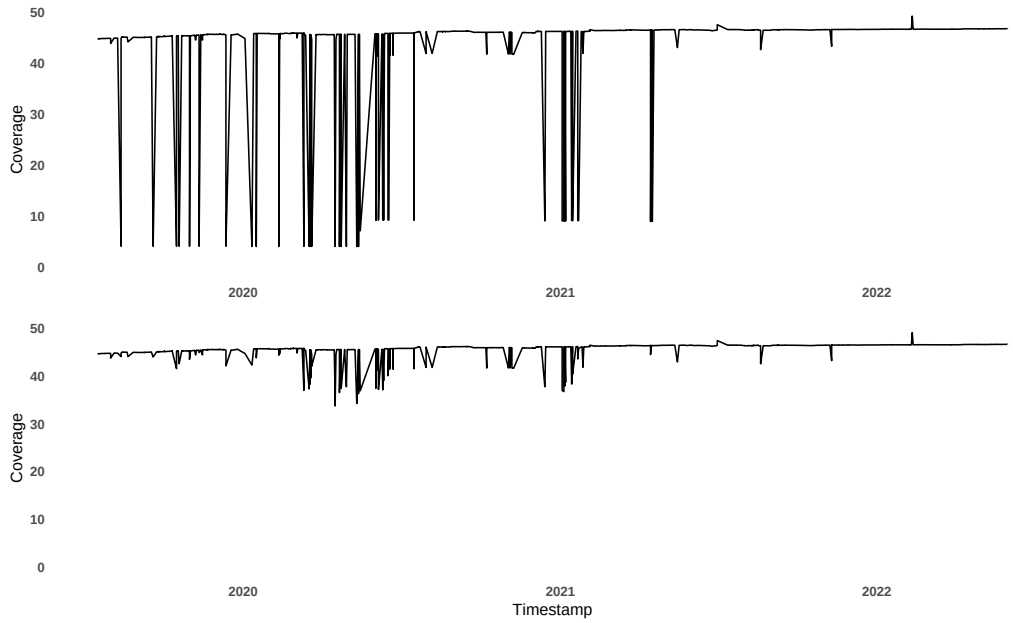


Figure 3.3: Apache/Gobbline before (**Top**) and After (**Bottom**) data cleaning with EMA

of the following variants:

- `"diff": null`
- `"diff": 0`
- `"diff": [..., null ,...]`

No definitive conclusions were drawn regarding the causes of the aforementioned variation cases, particularly since manual patch calculations in most instances yielded non-zero or non-null results. Discussion with the Codecov team revealed that these could be potentially caused by failure of the API to retrieve build information and/or build failure among other possible issues [136]. Consequently, there was a need for an approach that would be coverage service independent and consistent across all builds and programming languages. To address this, a Python algorithm was developed, and the following section provides the implementation details. However, for the purpose of comparing the accuracy of the coverage service patch coverage with that of the algorithm. the patch coverage returned by the API was extracted as part of the API calls. Additionally, the `diff` variation cases were treated as zero patch coverage.

```
{
"totals": {
        "files": 2,
        "lines": 18,
        "hits": 16,
        "misses": 2,
        "partials": 0,
        "coverage": 88.89,
        "branches": 0,
        "methods": 0,
        "messages": 0,
        "sessions": 1,
        "complexity": 0.0,
        "complexity_total": 0.0,
        "complexity_ratio": 0,
        "diff": [2,7,6,1,0,"85.71429",0,0,0,0,0,0,0]
    }
...  # code hidden for brevity
}
```

Listing 6: Codecov commit report at e7e2b8

Coveralls API

The Coveralls CI coverage service offers a publicly available web API to access build history for a project or repository. To retrieve build history statistics, Coveralls offers the `/username/repo_name.json?` endpoint. Furthermore, for commit-by-commit statistics, such as the number of files covered/uncovered, lines per file covered/uncovered, and coverage percentage per file, the `/builds/commit_hash/source_files.json` endpoint is available. When passing a file name as a query parameter in the commit endpoint, it returns a line coverage array for the specified file. This array indicates whether each line was executed (indicated by an integer greater than 0), executable but not executed (integer equal to zero), or non-executable lines (indicated by `null`). Listing 7 is typical build information returned by the Coveralls API.

```
{
    "repo_name": "apache/kylin",
    "branch": "master",
    "created_at": "2021-04-14T20:38:57Z",
    "commit_message": "KYLIN-4962 Fix NPE in ShrunkDict step",
    "commit_sha": "69368b08a770f6e54c4ecf43c538331e13cb03b2",
    "coverage_change": 0.0,
    "covered_percent": 28.093089777160824,
    "covered_lines": 26714,
    "missed_lines": 68377,
    "relevant_lines": 95091,
    "covered_branches": 0,
    "missed_branches": 0,
    "relevant_branches": 0,
    ...  # code hidden for brevity
}
```

Listing 7: Coveralls build report

3.5.2 Algorithm

While coverage services like Codecov [120] offer an endpoint with patch coverage out of the box, others, like Coveralls [121], lack this capability. To address this disparity, a Python-based algorithm was developed, leveraging the patch coverage definition outlined in Section 1.2.2. To reiterate, according to Marinescu *et al.* [17], patch coverage is "the ratio between the number of executed lines of code added or modified by a patch and the total number of executable lines in the patch, measured in the revision that adds the patch". The objectives of the algorithm were to:

1. Ensure a consistent definition of patch coverage across all services.
2. Enable data mining and extraction from various sources.
3. Facilitate comparison of patch coverage results.

Pydriller [137], developed by Spadini *et al.*, is a tool that utilises Git to mine source code statistics, ranging from repository to file-level statistics. Pydriller acts as an

abstraction layer between the Git and the metrics, offering well-formatted output and easy-to-use APIs. The algorithm developed employs Pydriller in conjunction with the Coveralls and Codecov APIs. Figure 3.5 provides the overall workflow of the mining process while Algorithm 1 provides the pseudo-code for the algorithm.

In typical developer interactions with a code repository, the process involves adding/-modifying files, staging the changes, and then pushing to the upstream repository, usually in that order. While there might be additional complexities such as pulling latest changes or resolving merge conflicts, this general process suffices for illustration. Developers often don't push after every commit; they commit changes, make additional changes, commit again, and repeat the cycle until everything is ready for the upstream push. This means the number of commits pushed can range from one to potentially hundreds in a single upstream push.

This poses a challenge from the perspective of coverage service tools. Coverage tools are typically unaware of local builds or commits and are triggered only by a push or a merge to the upstream repository, provided the build configuration is enabled for that branch. That is, if a developer makes ten commits before pushing to upstream, a coverage tool like Coveralls or Codecov will only pick up the latest commit as a trigger to build. Consequently, all the other changes are inherently lumped or coupled into this one change. This lack of a one-to-one mapping between a commit and coverage tool necessitated an approach that starts from the coverage service end, gathering all the commit hashes that triggered each build, and then iteratively checking out these commits to obtain patch information as depicted by Figure 3.4.

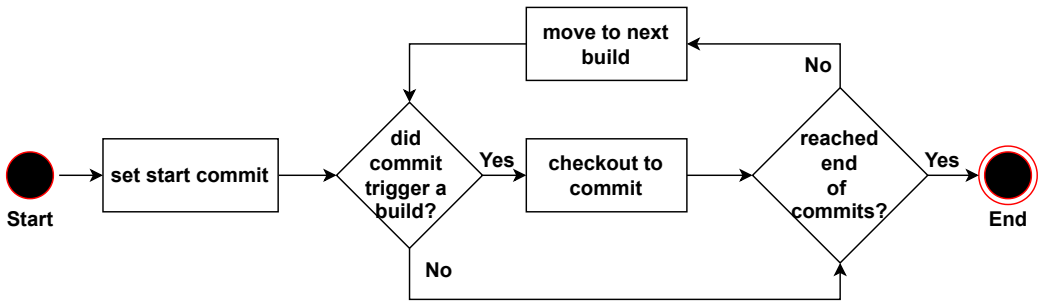


Figure 3.4: Build and Commit Hash Relation

As shown in Algorithm 1, the build information is an input, a dictionary consisting of key-value pairs with properties such as commit hashes, overall line coverage, covered lines, missed lines, and other relevant details. This build information is used to checkout the corresponding commit and compute the patch metrics.

Algorithm 1: Patch Coverage algorithm

Input : A $build\{commitSha : 4e2705, coverage : 74.3, \cdots\}$.

Output : Patch Coverage.

```
1  executedLines = executableLines = 0 // tested and testable lines
2  if !isMergeCommit(commitSha) then
3      modifiedFiles = modifiedFiles(commitSha)
4      totalModifiedFiles = len(modifiedFiles)
5      if totalModifiedFiles > 0 then
6          sourceFiles = fetchSourceFiles(commitSha)
7          forall file ∈ {1, . . . ,totalModifiedFiles-1} do
8              if fileMatcher(sourceFiles, file) then
9                  modifiedLines = modifiedLines(commitSha, Filename)
                   fileLineCoverage =
                   fileLineCoverageArray(commitSha, Filename)
                   lineMapping = lineMapper(fileLineCoverage, modifiedLines)
                   lines = len(lineMapping)
10                 forall line ∈ {1, . . . ,lines-1} do
11                     if lineMapping[line] is executable then
12                         executableLines + +
13                         if lineMapping[line] is executed then
14                             executedLines + +
15                         end if
16                     end if
17                 end forall
18             end if
19         end forall
20     end if
21 end if
22 return executedLines / executableLines × 100 // patch coverage percentage
```

From the pseudo-code, the functions `isMergeCommit`, `modifiedFiles`, and `modifiedLines` are wrapper functions utilising pre-existing methods from Pydriller's APIs. They return a boolean flag indicating whether a commit is a merge commit or not, the list of modified files, and the parsed Git diff for a modified file, respectively.

Functions `fetchSourceFiles` and `fileLineCoverageArray` use coverage service API to extract the list of tracked files by the coverage tools and the line-by-line coverage (Listing 8) information for each tracked file. Unfortunately, both Coveralls and

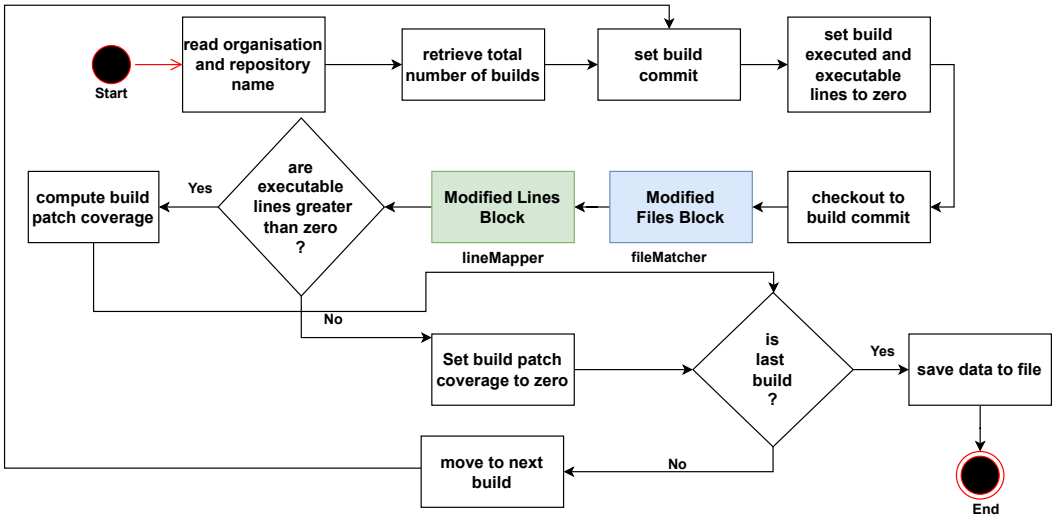


Figure 3.5: Algorithm pipeline-flow

Codecov do not explicitly provide files that were modified for a commit. Instead, for every commit, the commit reports return all source files tracked by the tool, irrespective of whether they were modified or not. This challenge led to the split of the algorithm into two folds, referred to as "mapping files" and "mapping lines" and will be discussed below:

Modified Files Mapping

The mapping of files serves as the first step. The function `fileMatcher` addresses this by taking the list of modified files from Pydriller's `ModifiedFile` API and the list of all tracked files by a coverage service tool, and by matching the file names with a regex string, it can be determined whether the modified file is or is not a source code file. That is, any source code file, whether covered or not will always be present in the tracked files array. This means that the coverage tool recognises the file as a source code file and will include it when computing coverage information. If the modified file(s) is not present in the tracked files array, therefore, it is a non-source code file, potentially a script file or configuration file. Figure 3.6 demonstrates a high level overview of how the `fileMatcher` works. The output of the `fileMatcher`, which is essentially a mapping of all files changed in that patch becomes the input to the `lineMapper` discussed below.

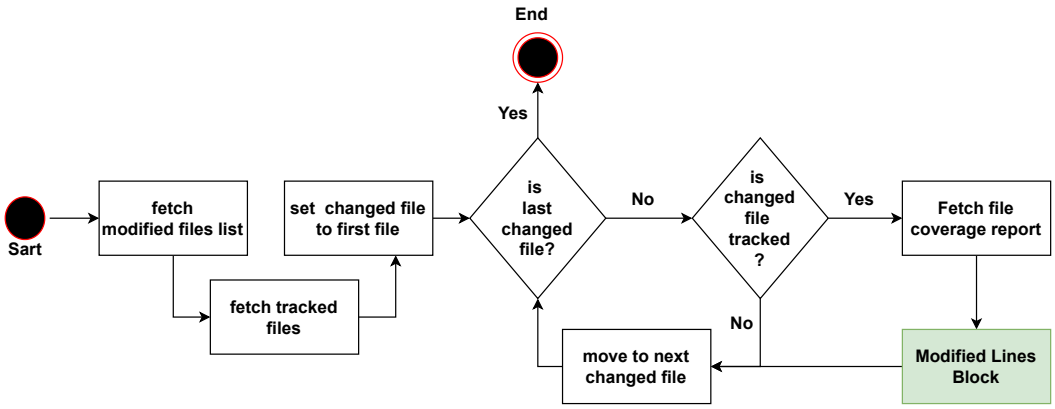


Figure 3.6: Modified Files Block: FileMatcher function

Modified Lines Mapping

The second step is the mapping of the changed lines with the corresponding coverage status from the coverage tool as illustrated by Figure 3.7. The `lineMapper` maps each line in the modified file with the line-by-line coverage report from `fileLineCoverageArray` function and returns the line with its corresponding coverage status. For each modified file, the `lineMapper` retrieves the file line-by-line coverage status and the modified lines. Each modified line is then mapped to its corresponding coverage status, and a new object is returned. For example, a "Calculator.py" file with a line-by-line coverage report shown in Listing 8 has file modifications as shown by Listing 9. Mapping the line-by-line coverage report shows that the file has line 4 (in the added property - `# method to divide two numbers`) as `null` indicating that it is not an "actual" source line and is not covered (i.e it is null), while line 8 (`if b == 0:`) has a line coverage report of 2, which means it is a source line and was covered - according to Coveralls. Lastly, line 9 (`raise ZeroDivisionError`) has line coverage of 0, again, indicating it is a source line, yet, not covered. A new object list of key-value pairs consisting the line number (specifically source lines) and its corresponding coverage status is returned, which is later used for calculating patch coverage. This process continues until all modified files have been exhausted, after-which the patch coverage is calculated.

Patch Measures Computation

The response (Listing 8) is used to illustrate the LineMapper function of a source file line coverage from Coveralls CI. It uses the indexes (position in the array) of the line coverage array to map to the modified lines in Listing 9. Codecov, on the other

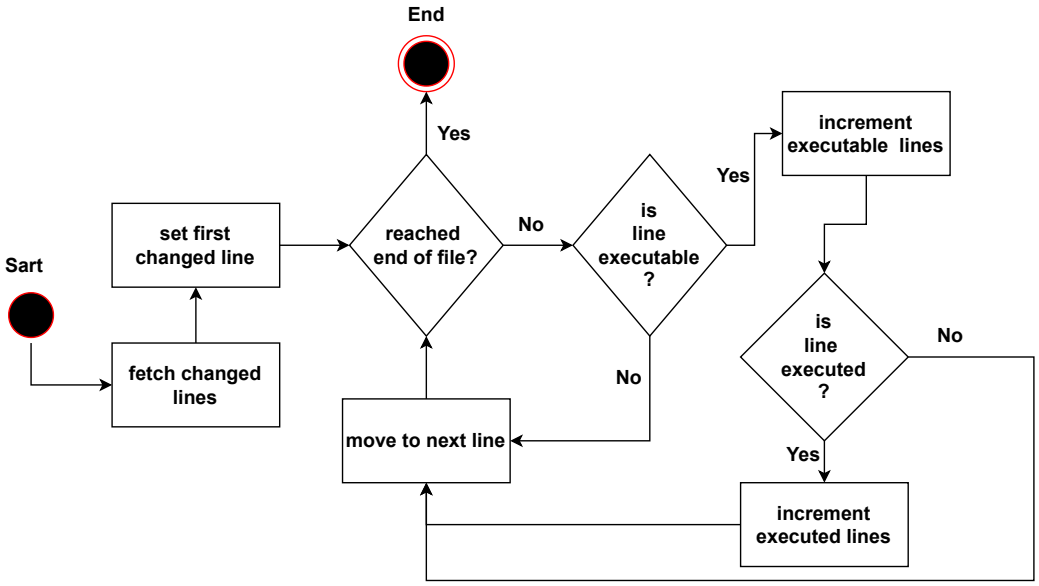


Figure 3.7: Covered Lines Block: LineMapper function

```
[
  null,
  null,
  2,
  null,
  2,
  2,
  null,
  2,
  0,
  2,
  null,
  2,
  2,
  2
]
```

Listing 8: Coveralls Line Coverage Array for `Calculator.py` at e7e2b8

hand, provides a two-element tuple per file where the first element is the line number and the second element indicates whether the line was executed or not. A sample response is shown in Listing 11. Fortunately, Codecov does not factor non-code lines

```
{
"added": [
    (4, "# method to divide two numbers"),
    (7, "# raise an exception if the denominator is zero"),
    (8, "if b == 0:"),
    (9, "raise ZeroDivisionError"),
    (11, "# method to subtract two numbers"),
    (12, "@staticmethod"),
    (13, "def subtract(a: int, b: int) -> int:"),
    (14, "return b - a")
    ],
"deleted": [ (7, "") ]
}
{
"added": [(9, "def test_subtract(self):"),
    (10, "self.assertEqual(4, Calculator.subtract(3,7))"),
    (11, "")
    ],
"deleted": [ (9, "") ]
}
```

Listing 9: Pydriller parsed `Git diff Calculator.py` commit - e7e2b8

in the line coverage array. Codecov categorises coverage status of lines into three bins: *hit*, *missed*, and *partial*. Hit lines are covered and executed lines and are represented by 0, while missed lines are not executed and are represented by 1. Codecov defines partially covered lines as branches with one or more omitted execution paths [138], and these are represented by 2. To illustrate this, consider Listing 10. If a test case consistently yields false for the condition `if ref.Name() == ""`, the subsequent statement `if ref.digest != ""` is never executed. Since `if ref.digest != ""` is contained within the `if ref.Name() == ""` condition, the latter is termed a multi-branch statement, and is invariably partially covered. This partial coverage arises from the fact that the condition `if ref.Name() == ""` is always evaluated, yet, there is never a guarantee that the child condition will always be executed. Thus, only one out of two branches is executed. Although partials are included in the output, Codecov ignores them when calculating patch and overall coverage. This approach was found to yield the same result as manual verification. The same mapping done

for Coveralls is applied to Codecov and a new list of key-value pairs with the modified code lines and the corresponding line status is returned.

```
1  func  getBestRerefenceType(ref reference) Reference {
2      if ref.Name()  == "" {
3          // Allow digest only references
4          if ref.digest != "" {
5                  digestReference(ref.digest)
6              }
7      }
8      ... // remaining code
9  }
```

Listing 10: Partialy Covered Line - Reproduced from Codecov [138]

To calculate the patch coverage, the total modified code lines executed in the patch are then divided by the total modified code lines in that patch. It is worthwhile mentioning that the total is across all modified source code files. In the event that no source code lines or files are changed (for example, only configuration files might be modified or the commit was adding comments) then the patch coverage is set to zero. This approach is adopted to all other patch metrics such as the DMM and delta CRAP metric.

3.5.3 Delta Maintainability Model

The SIG-MM model discussed in Section 2.1.1 employs five source code properties, namely duplication, size, complexity, unit interface and coupling. Analysing cloned code and coupling is resource-intensive [139]. This is because code clone identification involves tokenisation and parsing, it means when dealing with a large code base it will require large computing resources. Furthermore, mechanisms to mitigate false positives are required, which in turn necessitates sophisticated tuning methods. Additionally, what may constitute duplicate code in one language may not necessarily constitute a duplicate in another language. Thus, in an effort to keep the properties language independent, reduce ambiguity in definitions and to maintain consistency across all languages, the coupling and code duplication properties are excluded in the DMM computation. Therefore, the study adopted a three-property approach to the

```
... # excluded for brevity
"files": [
        {
            "name": "Calculator.py",
            ... # excluded for brevity
            "line_coverage": [ [ 3, 0 ], [ 5, 0 ], [ 6, 0 ],
                [ 8, 0 ], [ 9, 1 ], [ 10, 0 ],
                [ 12, 0 ], [ 13, 0 ], [ 14, 0 ]
            ]
        },
        ...  # excluded for brevity
    ]
```

Listing 11: Codecov Line Coverage Array for `Calculator.py` at commit e7e2b8

DMM provided by Spadini *et al.* [137]. The overall change DMM Score is then given as:

$$\text{DMM Score} = \frac{\Delta\text{DMM Size} + \Delta\text{DMM Interface} + \Delta\text{DMM Complexity}}{3} \tag{3.1}$$

Where:
Δdmm_size: The delta i.e net difference in size between two commits.
$\Delta dmm_interface$: The difference between the number of parameters between two consecutive commits
$\Delta dmm_complexity$: The difference in code complexity between two commits.

3.5.4 The Case For Commit CRAP

Alberto [43] defined the CRAP metric as "an index designed to analyse and predict the risk and effort associated in maintaining and enhancing an existing body of code, particularly by developers other than the original developers". The authors of CRAP metric speak of the CRAP metric from a "change" perspective. That is, they are examining the effect of changes to an already existing code-base. Similarly, the SIG-MM aims to understand the maintainability of changes to an already existing code-base. However, the original CRAP metric is defined for an entire method, not with respect to changes only. Adopting the approach that Heitlager *et al.* [61] used to

derive the SIG maintainability model and without altering the definition of the CRAP metric as provided by Alberto, a metric derived from the CRAP metric, called *Delta CRAP metric* is proposed. Unlike the original metric that considers both changed and unchanged lines of a method, the delta CRAP metric examines the CRAP metric of modified/changed lines i.e CRAP metric of a patch. Equation (3.2) represents the $\Delta CRAP^{'}$ of a patch, `p`.

$$\Delta\text{CRAP}^{'} = \Delta\text{comp}^{'} \times \left(1 - \frac{\Delta\text{cov}^{'}}{100}\right)^{3} + \Delta\text{comp}^{'} \tag{3.2}$$

In Equation (3.2), the $\Delta cov^{'}$ represents the patch coverage (coverage of changed lines) and the $\Delta comp^{'}$ represents delta score of cyclomatic complexity as calculated using the SIG-MM method. As noted previously in Section 2.1.1, the DMM delta scores of each source code property are normalised to be between zero and one [65], where zero indicates a risky change and one represents a good change. Applying the complexity delta scores in the CRAP metric equation, results in Equation (3.2). As observed in Equation (3.2), the equation remains fundamentally the same as Equation (2.5), with the substitution of overall values by delta/change values. Similarly, applying the scenarios discussed in Section 2.1.4 i.e zero patch coverage and 100% patch coverage results in Equation (3.3), which is also the same as Equation (2.6) except for replacement of method coverage with patch coverage.

$$\Delta\text{CRAP}^{'} = \begin{cases} \Delta\text{comp}^{'}, & if\ \Delta\text{cov}^{'} == 100\% \\ (\Delta\text{comp}^{'})^2 + \Delta\text{comp}^{'}, & if\ \Delta\text{cov}^{'} == 0\% \end{cases} \tag{3.3}$$

3.6 Analysis Approach

To empirically understand the evolution of software quality for fine-grained changes and the relationship between chosen quality metrics, two approaches are adopted. First, a distribution graphical representation is plotted to illustrate the evolution for each metric. Second, Spearman's correlation coefficient and Kendall's tau are computed to assess the strength of relationship between the metrics. Spearman's ρ and Kendall's τ coefficients are chosen because they do not assume any linearity between variables and do not assume any specific distribution of the data. Utilising both correlation coefficients aims at conducting cross-validation. While both coefficients are non-parametric and evaluate monotonic relationships, they vary in sensitivity. This approach ensures a more robust and comprehensive understanding of the relationships

between the metrics. To investigate the strength of the relationship, Cohen's definition of correlation strength is utilised [140], where a value less than 0.3 is considered a weak correlation, a value between 0.3 to 0.5 is deemed a moderate correlation, and a value greater than 0.5 indicates a strong correlation. The R statistical programming language [141] is used for computing correlation coefficients and plotting graphical trends.

3.7 Chapter Summary

This chapter presented the research methodology adopted in conducting this investigation. First, issues involved in downloading and compiling code-bases for metrics collection are identified such as package/library version mismatches. These issues led to the adoption of the containerisation approach. Two coverage service providers, Coveralls and Codecov, which offer persistent build histories for code coverage are discussed together with their respective API endpoints. Two seminal studies from Hilton *et al.* and Marinescu *et al.* which form the basis of this investigation was also introduced together with the research approaches adopted in these studies. Finally, a Python-based patch coverage algorithm developed for the computation of patch coverage is thoroughly presented.

4

Results and Discussion

This chapter presents the results from the gathered data. First, the results pertaining to the evolution of test and production is presented. Secondly, the patch-by-patch dynamics in terms of maintainability and testing is discussed. Lastly, the CRAP metric is analysed for patches. Where applicable, the results are presented on a per coverage service tool and in some instances the overall statistics (aggregated results) is given. The discussion with reference to the research questions and findings is provided. The implications of the results discussed here follows in Chapter 5.

4.1 Data Summary

To address the research questions, data was collected from Coveralls and Codecov, and commit metrics were computed for each commit across all projects. Using the procedure outlined in Section 3.3 of selecting projects (i.e prior project and the developed criterion), a total of 101 projects with a combined total of 35008 builds were extracted from Coveralls. After applying cleaning techniques detailed in Section 3.4, there were 34765 builds over 96 projects (only projects with a minimum of 20 builds per project, resulting in the reduced number of projects from 101 to 96). From Codecov, a total of 54151 builds from 103 projects were extracted. Applying the same cleaning techniques resulted in a total of 51927 builds over 92 projects.

In addition to mining unique data from these coverage services, 13 cross-coverage service projects were selected. These projects were either using both coverage services or were migrated from one coverage service to another. In all cases, the migration was unidirectional, typically from Coveralls to Codecov. The combined data for these

projects included 17 projects with a total of 9123 builds. After processing, there were 13 projects with a combined total of 9050 builds. In total, the report examined 201 projects with a combined total of 95742 builds across both coverage services. Table 4.1 provides the data sample, the remaining data can be found in Tables G.1 to G.6.

Table 4.1: Sample Representation of Projects Key Statistics

Identifier	Project Name	Language	Time(Months)	# Builds		LoC (K)		Coverage(%)	
				Total	Mined	Start	End	Start	End
CV01	alibaba/fastjson	Java	49	1311	490	92.12	28.1	90.54	80.27
CV02	chrisvest/stormpot	Java	18	61	40	0.644	1.05	96.89	95.61
CV03	mozilla/addons-scanner-utils	TypeScript	49	1979	462	0.23	0.44	95.26	97.48
CV04	Netflix/Priam	Java	30	236	101	6.60	72.7	36.69	47.84
CV05	Netflix/spectator-cpp	C++	29	141	74	1.23	1.02	89.89	82.25
CV06	oblac/jodd	Java	44	582	434	46.7	9.24	62.65	52.21
CV07	ReactiveX/RxJava	Java	78	2380	989	41.15	47.71	96.16	99.64
CV08	vavr-io/vavr	Java	49	970	363	11.85	12.54	97.91	92.74

From Table 4.1, the start column in both LoC and Coverage represents code lines as measured by source lines for the earliest date and line coverage. The end represents the values on the latest build date which was mined. As previously alluded to in section 3.4, not all build data were extracted due to errors, as such, the number of builds column represents the total builds and mined builds at the time of conducting this study. In addition to these, the time difference between the earliest extracted build and the latest extracted build to determine the time-span under study in terms of months is presented. For the purpose of analyses, each project is denoted by an identifier as shown in column 1. Identifiers `CV, CL` and `CM` are for Codecov projects, Coveralls projects and cross-coverage service projects respectively. Henceforth, each project is referred by its identifier.

4.2 Characteristics of Patches

Before delving into the research questions, the first task was to emphasise the importance of grasping the composition of a typical patch. To achieve this understanding, a set of Exploratory Questions (EQ), adapted from and referred as Demographic Questions (DQs) by Hilton *et al.* [18] were repeated.

4.2.1 Co-Evolution of Production and Test Code in Patches

This question, stated as "Do patches touch both code and tests?" mirrors DQ1 question in [18]. To answer this EQ, for each revision, the modified files were examined per patch. To identify modified files, Pydriller's `ModifiedFile` API is employed [137]. The `ModifiedFile` API has the `filename` attribute which holds the modified file's name. The file name is extracted and checked for existence in the build tracked files array. Tracked files are files identified by the coverage service tool as code files, they may or may not be covered. The checking is done to identify whether the file is a source file or not. If not, it will not be present in the tracked files array, thus, eliminating it. To categorise a file as a production or test file, the technique used by Kochhar *et al.* [142, 143] where they classified any source files that consists of the string "test" in the filename as a test file, otherwise it is a production file. By production it is referred to source code files designated to the application logic excluding testing. A two step approach is taken to check if the sub-string "test" exists. First, the filename as returned by the `ModifiedFile` API is checked, if the result is true then it is classified as a test file. If the file name does not have "test", the second check is conducted. The tracked file array returned gives the file names as a full path rather than actual file name only. This is useful in cases where the file name may not contain the string "test" but the directory path may contain "test". Say for example, `/src/app/test/Calculator.java`, while `Calculator.java` here does not have the sub-string "test" in the file name, it most likely has one of two purposes: 1) Act as a test file or 2) contains test utility code (otherwise known as helpers). As such, utility code located within test directories is classified as test code rather than production code.

The analysis of Figures 4.1 and A.1 to A.3 in Appendix A reveals a predominant pattern across projects: test files are rarely altered independently of production files. Notably, the modification of production files occurs more frequently without concurrent changes to test files. However, when test files are modified, there is a consistent observation that production files are also altered within the same patch. This synchronous pattern, as described by Zaidman et al. [3, 144, 145], signifies the co-evolution of code and tests. While Zeeger et al. [3, 146] observed both synchronous and phased patterns (patches exclusively modifying test files), this study finds no substantial evidence of the phased testing activity. It is perhaps worthwhile to acknowledge the draw-back of matching test file using the filenames, especially in cases where the filename does not contain the string "test" as pointed out by Kirui and Levitt [147]. Kirui and Levitt [147] highlight the reduced accuracy of identifying test and production files when filenames deviate from traditional conventions. They

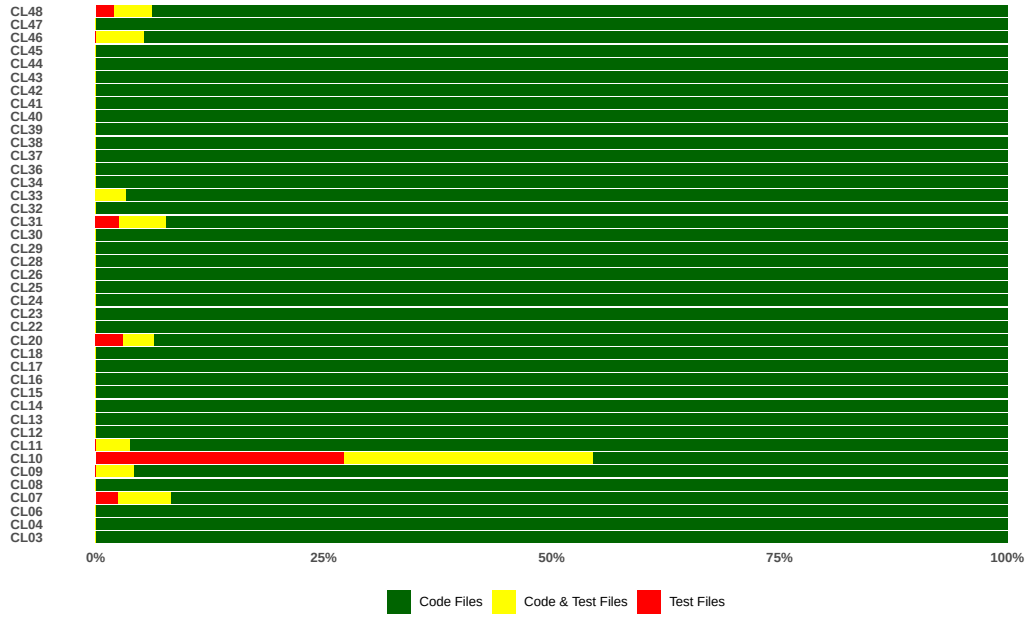


Figure 4.1: Demographic Question 1: Coveralls - Coevolution of Production and Test code. Size of bar represents the proportion of builds over the total number of builds in each category.

propose assertion-statement based approach to enhance classification accuracy, which could be employed for more precise identification of test and production test files.

4.2.2 Analysing the Typical Size of Patches

Hilton *et al.* [18] argues that smaller patches are easier for engineers to understand, while larger patches may necessitate external strategies for comprehension due to potential complexity. To address this EQ, the focus here is specifically on patches impacting source code, with an examination of code churn distribution (modified lines, added lines, and deleted lines).

The distribution of patch sizes across projects, illustrated in Figures 4.2 and A.4 to A.6, is observed to be uneven, ranging from 1 to 500 lines. An interesting example is in CV27, where in a single commit (commit hash 90bf5335), 87 files were touched, resulting in a total code churn exceeding $70K$ lines. While there are other outliers surpassing $15K$ lines, these are seen as exceptional cases. Isolating such exceptions and computing the mean reveals that, in most cases, a typical patch consists of approximately 10 to 100 lines. These findings align with the work of Hilton *et al.*. [18] and Abdulkareem *et al.* [148] but differ from Marinescu *et al.* [17], who reported a low number of lines changed, ranging from 4 to 7.

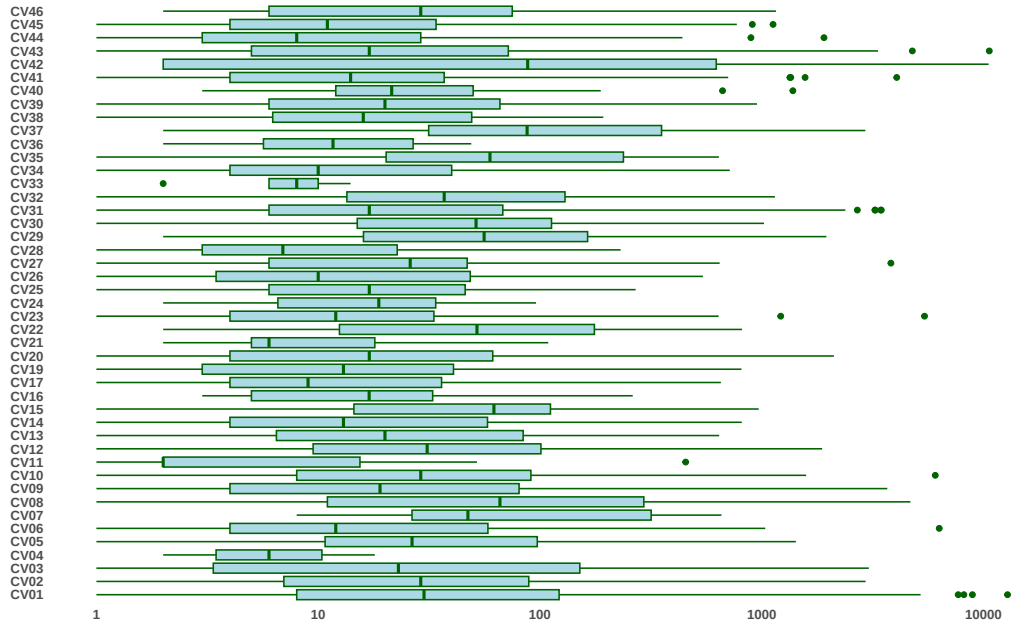


Figure 4.2: Exploratory Question 2: Codecov - Boxlot of Source Code patch Size (Code Churn). Horizontal axis represent the number of lines modified.

4.3 Testing Dynamics Across Patches

4.3.1 Distribution of Patch Coverage

Agile methodologies, like Scrum and Kanban, centre around iterations and/or sprints, emphasising developers' implementation of small, incremental changes to applications. This, therefore, means in such environments, developers are likely to care more about day-to-day changes. Thus, given that these changes (patches) are small and incremental, it might be worthwhile focusing on coverage of these changes especially for reviewers and contributors [18].

To investigate the coverage of patches, Algorithm 1 outlined in Section 3.5.2 is used to compute patch coverage for the entire build history of changed statements per project. The percentage coverage of each patch is then binned using the following bins: 0%, $(0\% - 25\%]$, $(25\% - 50\%]$, $(50\% - 75\%]$, $(75\% - 100\%)$ and 100%. These bins are the same to those of [18]. As in the case of Hilton *et al.*, the 0% and 100% bins are specifically chosen to isolate patches that are either completely uncovered or fully covered. A stacked bar graph is plotted to visualise the distribution for each project.

The investigation initially covered all patches, including both code and non-code

patches. Results are presented in Appendix H, Figures H.1 to H.4. On the surface, it is noted that a significant proportion of patches exhibit zero patch coverage, indicating that most patches are not covered. This means that a large proportion of patches are directed at modifying non-code files (i.e configuration files, scripts etc.). However, delving deep into the analysis reveals that this behaviour is attributable to a large number patches modifying non-source files. To investigate further, the focus is narrowed to isolate patches which explicitly modify code files. While patches modifying non-code files sometimes impact patch coverage (as elaborated later), the emphasis is initially on understanding patch coverage in patches modifying source code. Therefore, Figures 4.3 to 4.7 illustrate the patch coverage specifically for patches affecting one or more code files, whether they are production or test files.

With regards to patches modifying code files, a key finding emerge: The majority of patches are either fully covered or not covered at all i.e only a select few projects had partially covered bars exceeding the 0% and 100% bins. This finding validates the observations of Marinescu *et al.* [17] together with those of a recent study done Sterk [149] and disprove those of Hilton *et al.* [18]. Hilton *et al.* recognised the binary nature of patch coverage, pertaining to the 0% and 100% bins, indicating that most patches are either fully covered or entirely uncovered. However, their conclusion was that patch coverage exhibits variability across patches without a discernible pattern. The results from Hilton *et al.* fail to clearly distinguish between patches that modify source code and those modifying non-source files. Consequently, it is challenging to pinpoint the exact reason for the disparity between their results and these observations.

The primary difference between these results to those of Marinescu *et al.* lie in the fact that unlike Marinescu *et al.*, their findings showed a significant number of projects where the fully covered percentage (100%) bin surpassed the uncovered percentage (0%) bin.

While most projects adhere to the mentioned patterns, three deviations are observed. Projects CV68 and CL{43,63,69,96} exhibit zero patch coverage consistently, suggesting the absence of tests for all studied builds. Further investigation reveals that these projects had only patches affecting production files, and none of those patches touched test files. That is, among those patches modifying code files, none were covered at all.

Another intriguing case involves projects CV{04,07,82}, CM05 and CL{18,29,47} where there is no 100% bar; instead, the maximum reaches the (75% – 100%)

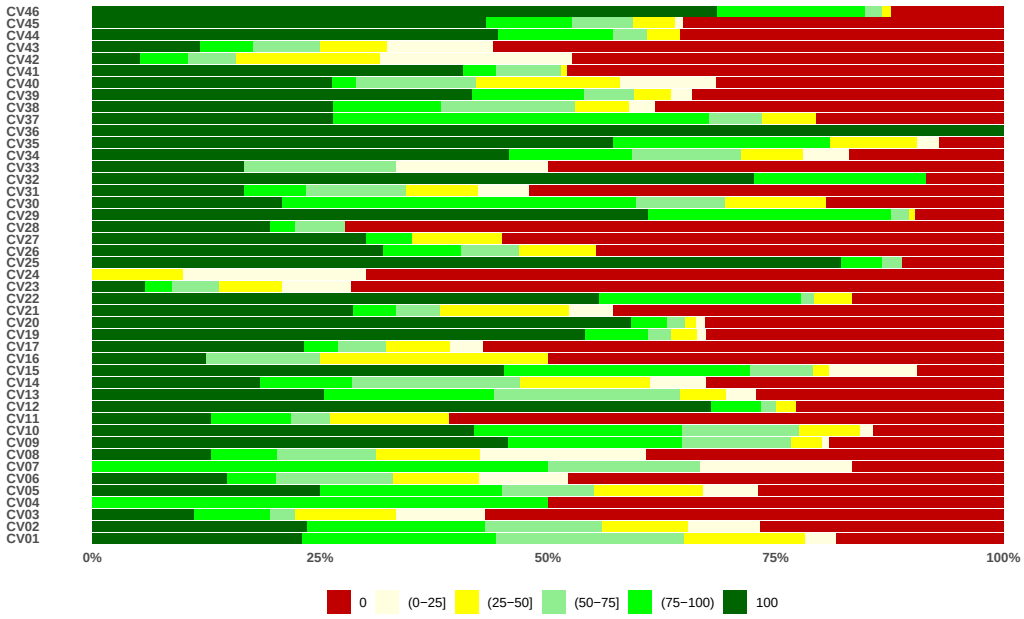


Figure 4.3: Codecov: Distribution of patch coverage across revisions. Each colour represent the range and the size of the bar is the percentage of the patches whose coverage lies within the range.

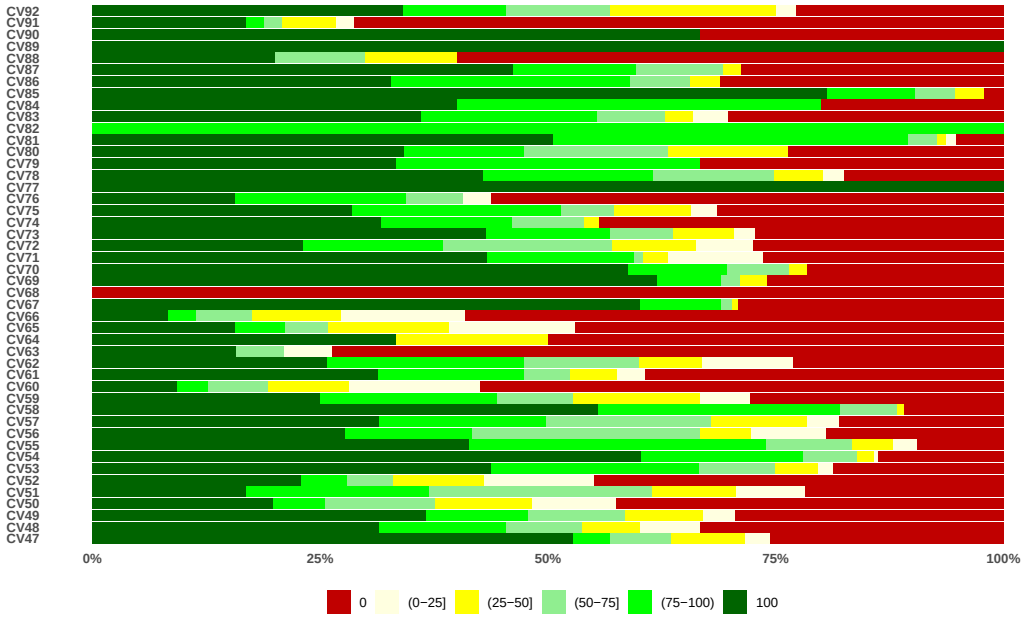


Figure 4.4: Codecov Cont: Distribution of patch coverage across revisions. Each colour represent the range and the size of the bar is the percentage of the patches whose coverage lies within the range.



Figure 4.5: Coveralls: Distribution of patch coverage across revisions. Each colour represent the range and the size of the bar is the percentage of the patches whose coverage lies within the range.

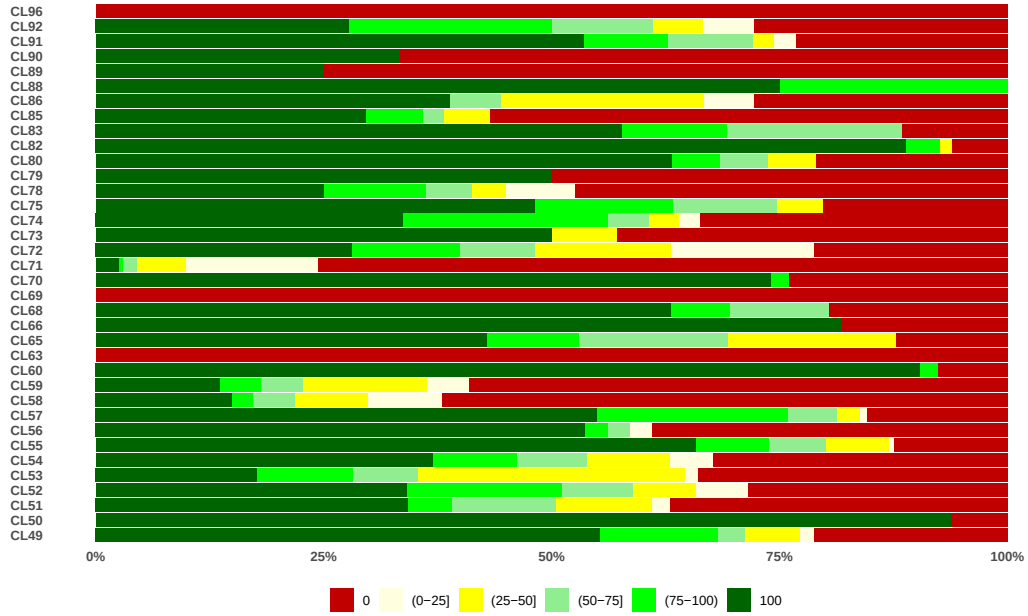


Figure 4.6: Coveralls Cont: Distribution of patch coverage across revisions. Each colour represent the range and the size of the bar is the percentage of the patches whose coverage lies within the range.

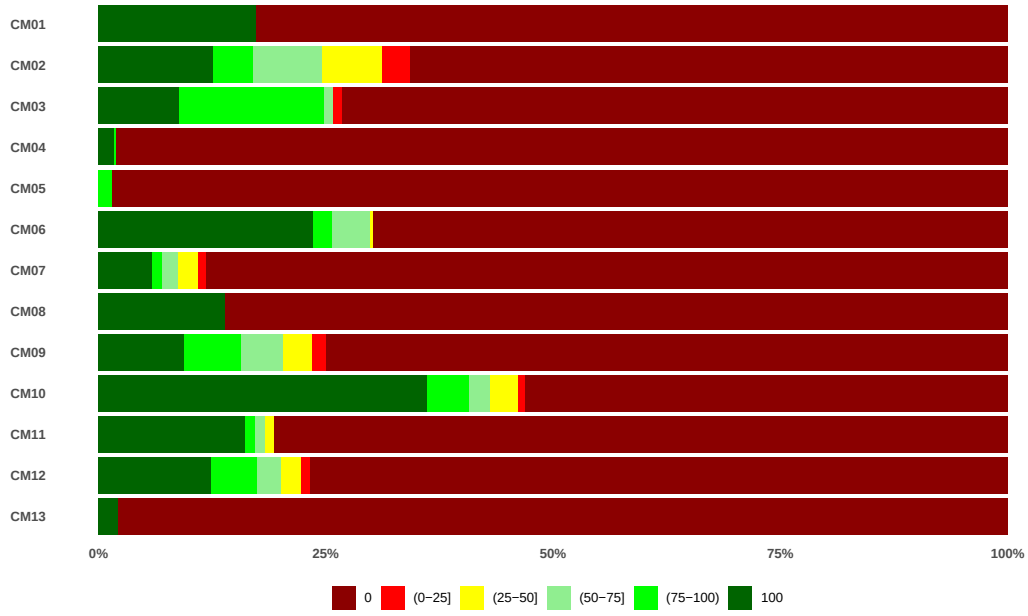


Figure 4.7: Distribution of patch coverage across revisions. Each colour represent the range and the size of the bar is the percentage of the patches whose coverage lies within the range.

range. This is interesting because it suggests that engineers could be opting for a specific coverage percentage rather than aiming for full coverage of all patch lines [150]. While it is counter-intuitive, it raises the possibility of a minimum coverage percentage prerequisite for merging commits in certain open source communities, although conclusive evidence is lacking. Finally, in projects CV{30,37} and CM03 the $(75\% - 100\%)$ coverage bar surpasses the 100% bar, echoing the previously mentioned case. It suggests a trend where developers may prioritise a specific coverage percentage over achieving 100% patch coverage. This aligns with the findings of Prause [150] who noted that developers often consider a coverage percentage between 80% to 90% as indicative of satisfactory coverage.

Comparative Analysis of Native API Patch Coverage Against Algorithm Patch Coverage

Utilising the Codecov web API's inherent patch coverage capability, the patch coverage was queried and compared to that returned by the algorithm. Notably, a discernible disparity is observed between Figure 4.3 and Figure 4.8, as well as between Figure 4.4 and Figure 4.9 respectively. To understand this behaviour Codecov team was contacted to validate the methodology behind their patch coverage computation. It was

confirmed that the mathematical expression for patch coverage aligns with the definition provided by Marinescu *et al.* [17]. Subsequently, the results were further corroborated through experimentation outlined in the Appendix B.

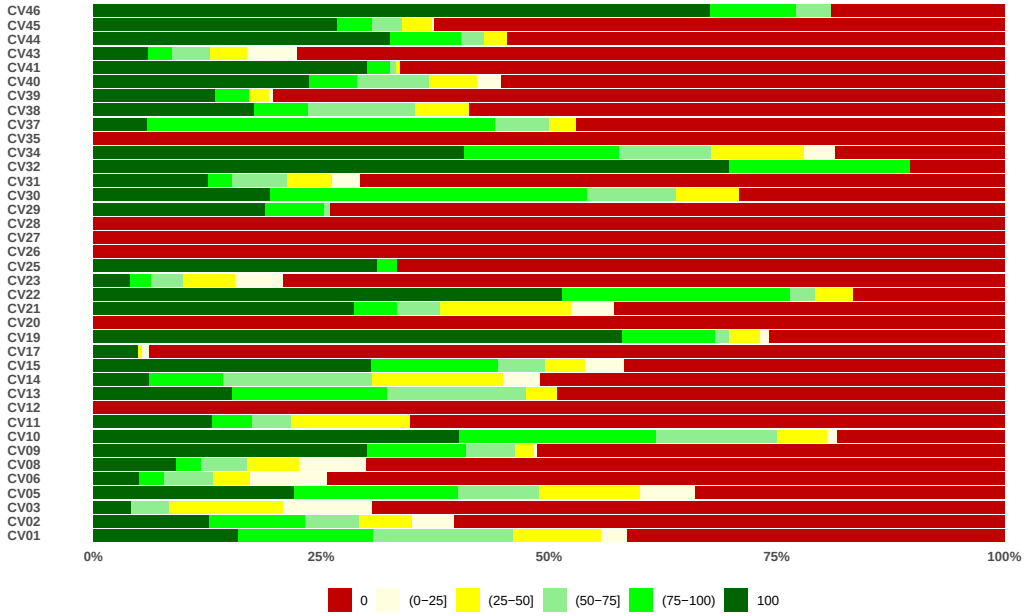


Figure 4.8: Native Rest API Patch Coverage From Codecov: Distribution of patch coverage across revisions. Each colour represent the range and the size of the bar is the percentage of the patches whose coverage lies within the range.

Upon closer examination of the variance in results from those of the algorithm's patch coverage to that of the web API, it is noted that a substantial portion of patches exhibiting non-zero patch coverage from the algorithm corresponds with those provided by the web API. The disparities arises primarily when the API returns either a null or zero value for the `diff` property, as previously discussed in Section 3.5.1. This discrepancy is influenced by the deliberate decision to treat such variations in `diff` property as zero rather than disregarding them altogether. Detailed scrutiny of the web API reveal that these instances stem from build failures or the API's inability to retrieve the build output. This assertion was corroborated through confidential correspondence with the Codecov team via an issue request [130, 131].

4.3.2 Impact of Patches on System Coverage

A patch can either have full, partial, or no coverage. However, the consequence to the overall system coverage is not known *a priori*. To assess this impact, positive influence is defined as an increase in system coverage, negative as a decrease, and no change as

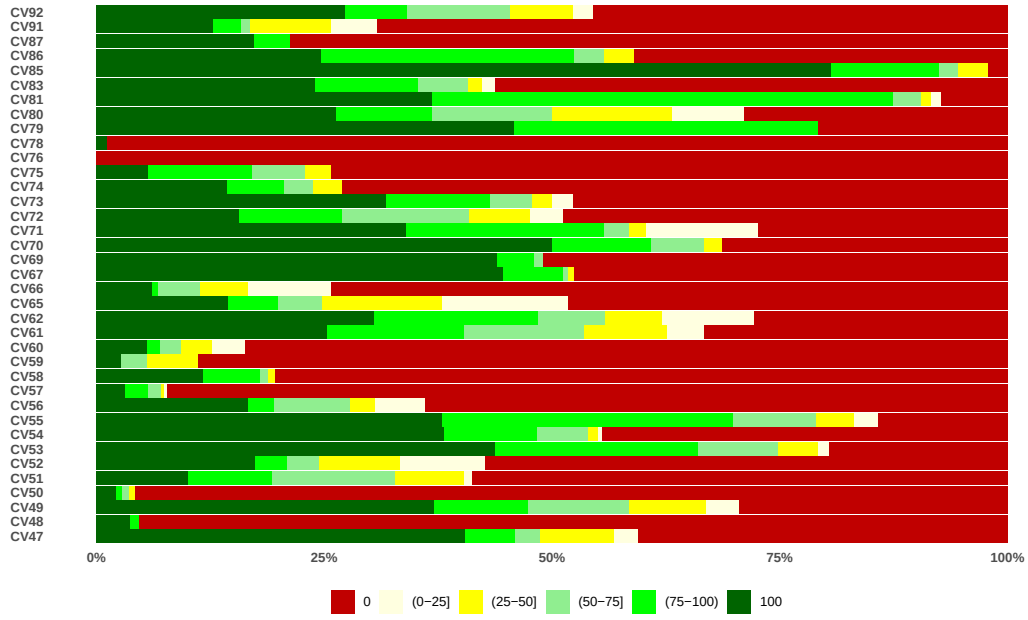


Figure 4.9: Native Rest API Patch Coverage From Codecov: Distribution of patch coverage across revisions. Each colour represent the range and the size of the bar is the percentage of the patches whose coverage lies within the range.

neutral. Patches are categorised based on whether they modify source or non-source files. In answering this question, the difference between the coverage data points is examined. If the net coverage from `d1` (where `d1` is the earlier date) to `d2` (where `d2` is the latest date) is less than zero, the patch is said to have decreased coverage, if the difference is greater than zero, the patch is said to have increased the coverage, and if the difference is zero, the patch is classified as no change to overall coverage. Each effect of the patch is then binned according to its corresponding impact, as shown by Figures 4.10 and 4.11. The plot illustrates the impact distribution of changes categorised by type (source or non-source code) with each bin representing a percentage of each bin over the total number of builds.

The analysis reveals that a substantial portion of changes appears to have a "No change/impact" effect on the system coverage, irrespective of file type (i.e source or non-source). For those that have an impact, the impact of these changes varies from patch to patch. Notably, certain patches to non-source files exhibit a significant influence on overall coverage. For instance, approximately 50% of builds in projects CV{07,17} with changes to non-source files appear to increase overall coverage - a phenomenon previously observed by Hilton *et al.* [18]. This finding is counter-intuitive, as one would typically assume that non-source code changes should have no impact on coverage, or if any, it would be minimal, attributable to the dynamic nature of code

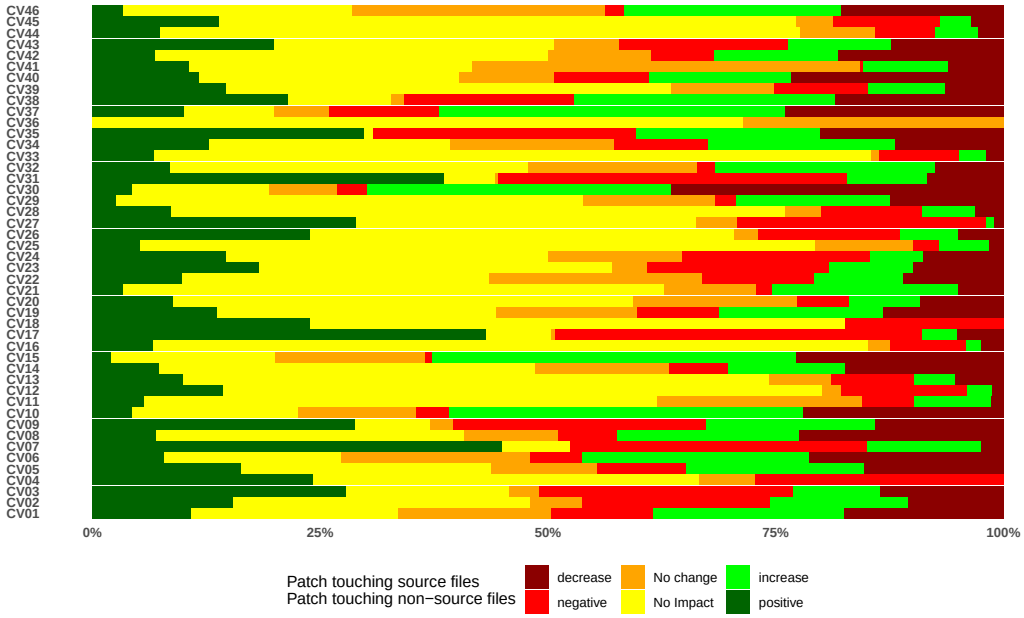


Figure 4.10: Codecov: Distribution of the impact of revisions to the overall system coverage. Each colour represent the range and the size of the bar is the percentage of the patches whose impact lies within the range.



Figure 4.11: Codecov Cont: Distribution of the impact of revisions to the overall system coverage. Each colour represent the range and the size of the bar is the percentage of the patches whose impact lies within the range.

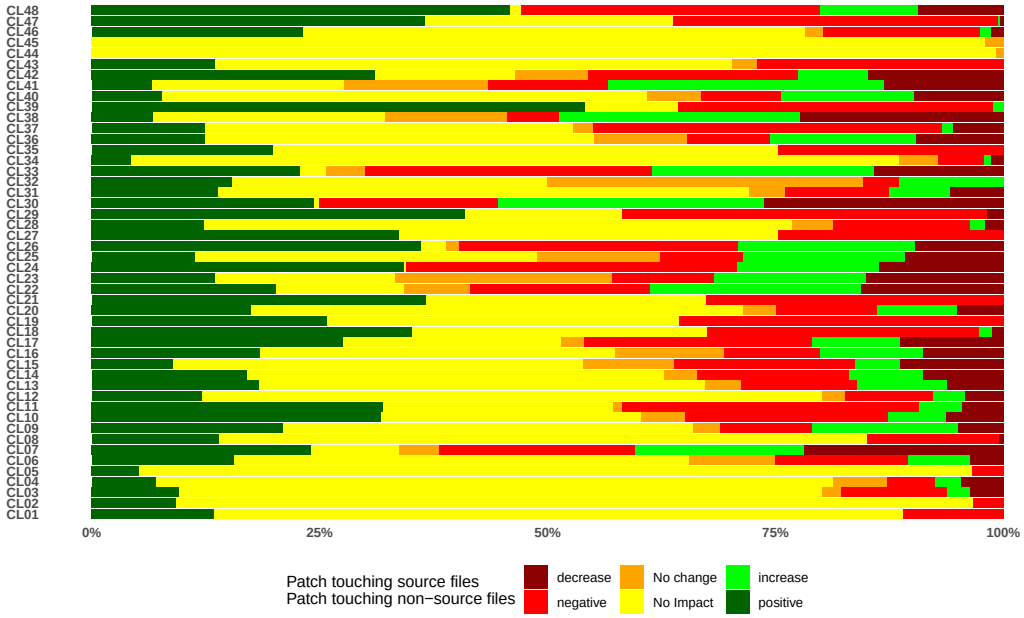


Figure 4.12: Coveralls: Distribution of the impact of revisions to the overall system coverage. Each colour represent the range and the size of the bar is the percentage of the patches whose impact lies within the range.

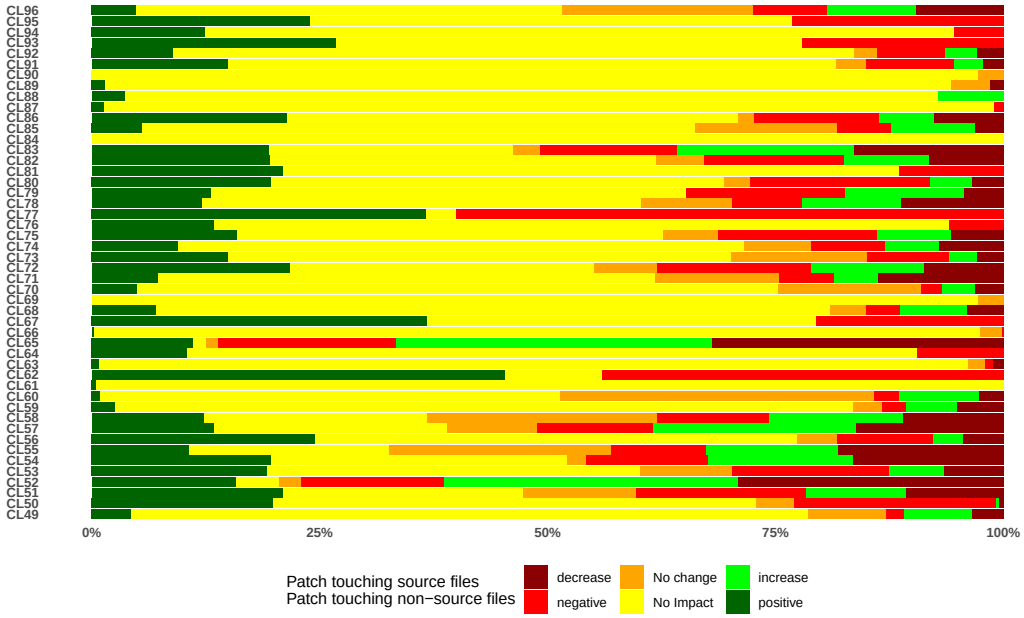


Figure 4.13: Coveralls Cont: Distribution of the impact of revisions to the overall system coverage. Each colour represent the range and the size of the bar is the percentage of the patches whose impact lies within the range.

coverage collection [128]. There are handful of reasons that could lead to unexpected coverage change including from non-source files [151]. A good example is that of CV07, patch 72ade6 which was an updated to the `Makefile` to specifically to fix tests, in fact, the commit message is titled "Correct code coverage". The change to the `Makefile` resulted in quite a significant change in the overall coverage, suggesting that non-source code changes have the potential to impact overall coverage. Other noted reasons are:

- Coverage tool config file update - Coverage tools typically require configuration files and connection strings to collect coverage reports. These configuration files serve as entry points and enable users to customise various settings, such as specifying inclusion and/or exclusion of directories, files, and other parameters. For instance, Codecov utilises a configuration file named `codecov.yml`, where users can define paths to be included or excluded. Additionally, this file allows users to set thresholds, determining whether a build is considered successful or failed based on specified criteria.
- Build Scripts - In addition to utilising coverage tool entry point files, developers frequently configure builds using scripts like `BASH` and `Makefile` to build test reports. These scripts provide a more dynamic approach, allowing for automated configuration of various aspects within the builds, including coverage reporting. Depending on the change to these scripts, the trajectory of the coverage could change quite considerably.
- Failing to upload coverage report and missing reports or failed builds - The robustness of error handling in CI pipelines significantly influences the integrity of coverage reports. Build failures, if not adequately handled, can lead to missing or corrupted coverage reports, thereby yielding unexpected results in the coverage tool. This issue was also observed in the experimental repository.

In contrast, projects CV{36,77,82,89} exhibit behaviour deemed more plausible, where changes to non-source files do not affect coverage. Instead, coverage changes are observed only when patches include code modifications. Surprisingly, the expectation that high-coverage patches would increase overall system coverage is not supported by the evidence.

To further investigate the relationship between coverage and patch coverage, Spearman and Kendall correlation coefficients are computed on both a project-by-project basis and a data-wise (all projects per coverage service), as presented in Tables F.1

to F.6. Both project level and data-wise analyses using Spearman's coefficient show virtually no correlation between the two metrics. Specifically, for Codecov, Spearman's coefficient reports $\rho = 0.0598, d = 0.1198$ and Kendall's coefficient $\tau = 0.0494, d = 0.0989$, while for Coveralls, the corresponding values are $\rho = 0.022, d = 0.044$ and $\tau = 0.018, d = 0.036$ (as given in Table 4.2). This emphasises the distinct features of individual patches, necessitating focused attention and treatment as independent entities.

4.4 Evolution of Quality Across Patches

4.4.1 Maintainability of Fine-Grained Changes

The study utilises the DMM for assessing maintainability. The DMM was designed specifically for small incremental changes to source code. With that in mind, to investigate the maintainability of fine grained changes, the first step was to filter out patches touching only non-source files. This meant that some projects, whose entire build histories were changing non-source files are excluded in the analysis. To visualise the maintainability of these changes, a distribution plot is created. According to Di Biase *et al.* [65], the DMM explores the changes to source code as the "ratio of good changes over the sum of good and bad changes", mathematically expressed as $\text{DMM} = \frac{\text{good}}{\text{good + bad}}$, the resulting value is therefore, between 0 and 1, where zero indicates a "bad" (unmaintainable) change and one indicates "good" (maintainable) change.

The first thing to note is that projects CL{01,02,05,19,21,27,35,61,62,64,67, 76,77,81, 84,87,93-95} and CV18 are not present in the DMM data. Manual inspection confirms that these projects had no changes to source files throughout the examined builds. Figures 4.14 to 4.17 depict the distribution of DMM categorised into four bins, $(0 - 0.25)$, $(0.25 - 0.50)$, $(0.50 - 0.75)$ and $(0.75 - 1.0)$. For projects with changes to source code, the maintainability varies across patch to patch, there's no specific pattern of maintenance observed. However, a few outliers are noted. For example, project CL08 has a high DMM for all patches, having the $(0.75 - 1.0)$ bar only. This is not surprising as looking at the patch coverage distribution shows that project CL08 has the 100% bar only of patch coverage. The same is observed for project CV89.

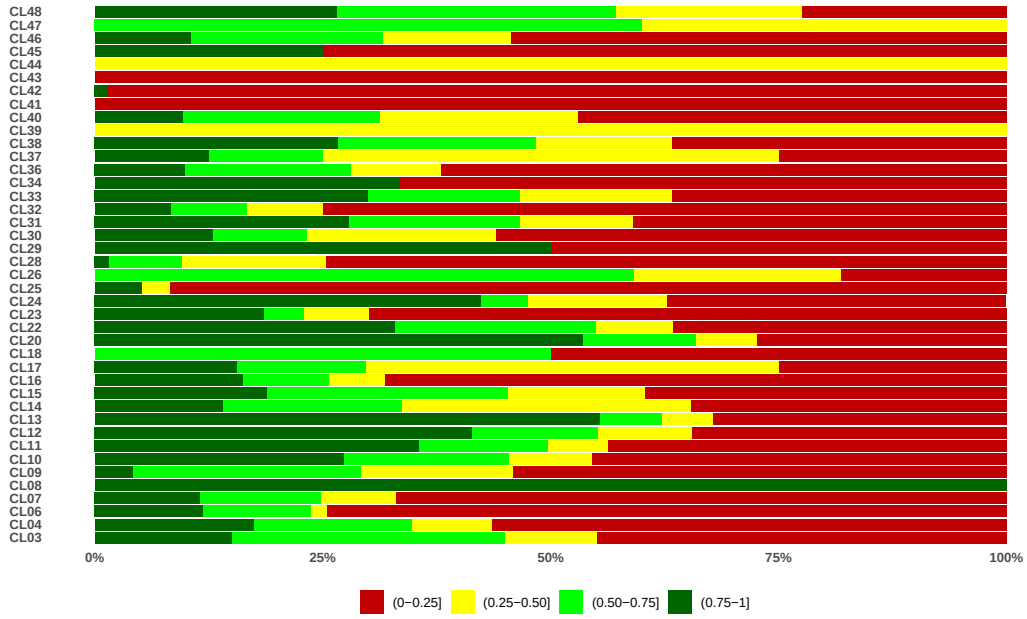


Figure 4.14: Coveralls: distribution of Delta Maintainability (DMM). Each colour represent the range and the size of the bar is the percentage of the patches whose impact lies within the range.

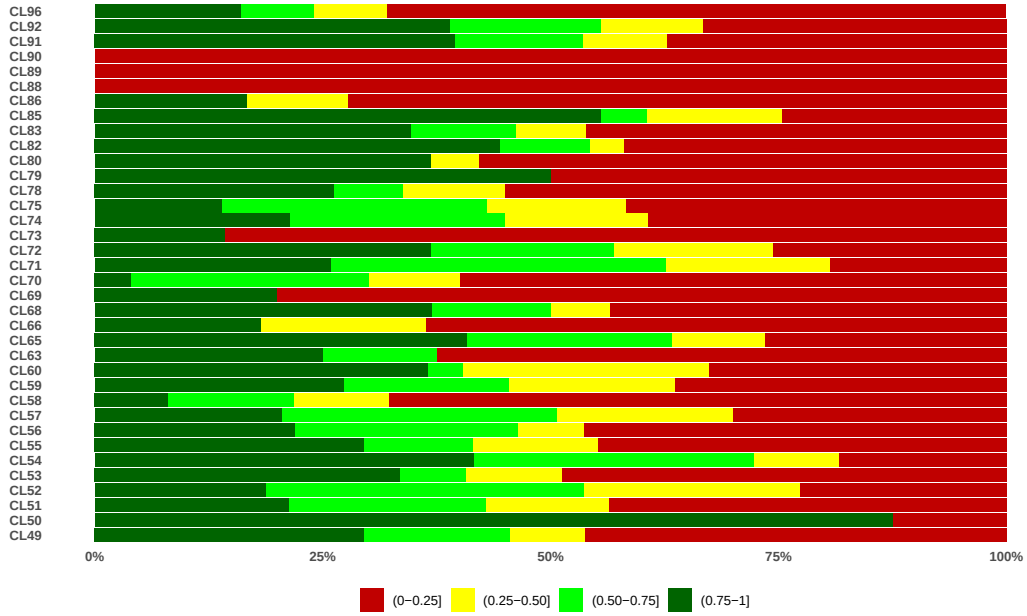


Figure 4.15: Coveralls Cont: distribution of maintainability (DMM) . Each colour represent the range and the size of the bar is the percentage of the patches whose impact lies within the range.

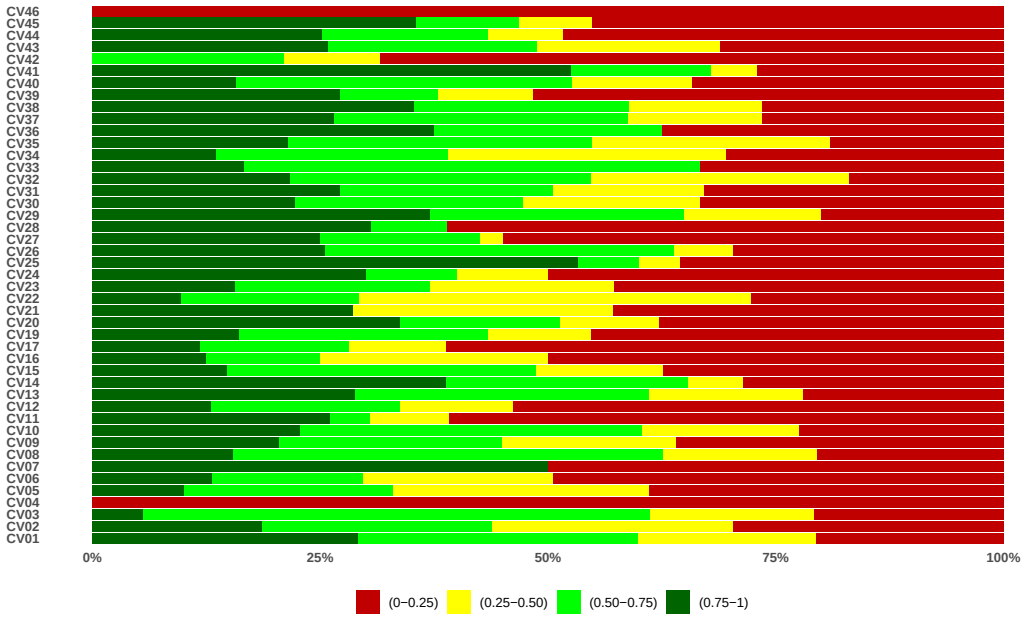


Figure 4.16: Codecov: distribution of maintainability (DMM) across revisions. Each colour represent the range and the size of the bar is the percentage of the patches whose impact lies within the range.

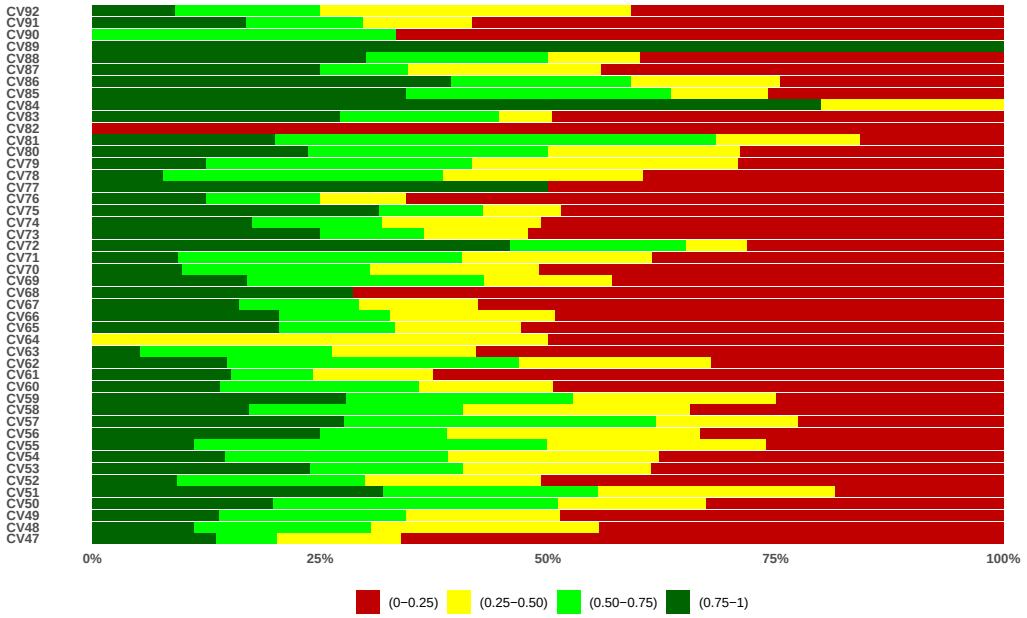


Figure 4.17: Codecov Cont: distribution of maintainability (DMM) across revisions. Each colour represent the range and the size of the bar is the percentage of the patches whose impact lies within the range.

This report conducted an examination of the impact of patch coverage on maintainability, analysing the strength of this relationship on a project-by-project basis. The findings, detailed in Tables F.1 to F.3, indicate that, for a majority of projects, the strength of the maintainability-patch coverage relationship ranges from moderate to very strong. The overall data-wise analysis reveals a moderate strength for both Spearman and Kendall coefficients, as presented in Table 4.2. This observed strength reinforces the high values of the DMM score in the distribution of the DMM.

However, when assessing the connection between maintainability and overall coverage, no significant link is apparent. This paradox mirrors the relationship observed between coverage and patch coverage, where no significant strength is noted between the two. This is intriguing because intuitively, it is well known that a maintainable software is more likely easy to modify, which in turn means that adding/modifying test code should somehow impact the code coverage.

4.4.2 CRAP Metric Analysis Over Revisions

The study explores a variant of the CRAP metric, derived from the original CRAP metric, denoted the delta CRAP metric (Equation (3.2)). This variant assesses the "goodness" of changes by considering patch coverage and patch complexity. Patches exclusively modifying code files are identified as potentially introducing risky changes. Thus, the emphasis is on isolating patches with changes solely to code files. The distribution of the CRAP metric is visualised using a box plot (whisker plot). The minimum and maximum values depicted in the box plot symbolise the two scenarios outlined in Section 3.5.4, where patch coverage can be either zero or 100%.

The analysis of CRAP metric in various software projects, as observed in Figures 4.18 to 4.21, reveals a consistent trend. Analysing the distribution of the delta CRAP metric reveals that a majority of the projects have 50% of their total number of patches with a delta CRAP metric score distributed between 0 and 1. This suggests that, for most projects, patches either incorporate tests or introduce minimal complexity. Recall from Equation (3.3) that for an uncovered patch, the CRAP metric equates to a polynomial of the complexity of the patch while for a fully covered patch, the CRAP metric equates to just the constant value of the complexity of the patch. Moreover, this behaviour could align with EQ2 discussed in Section 4.2.2 as it found that the typical patch size is between 10 to 100. It is plausible that such sizes of patches would not introduce significant complexity.

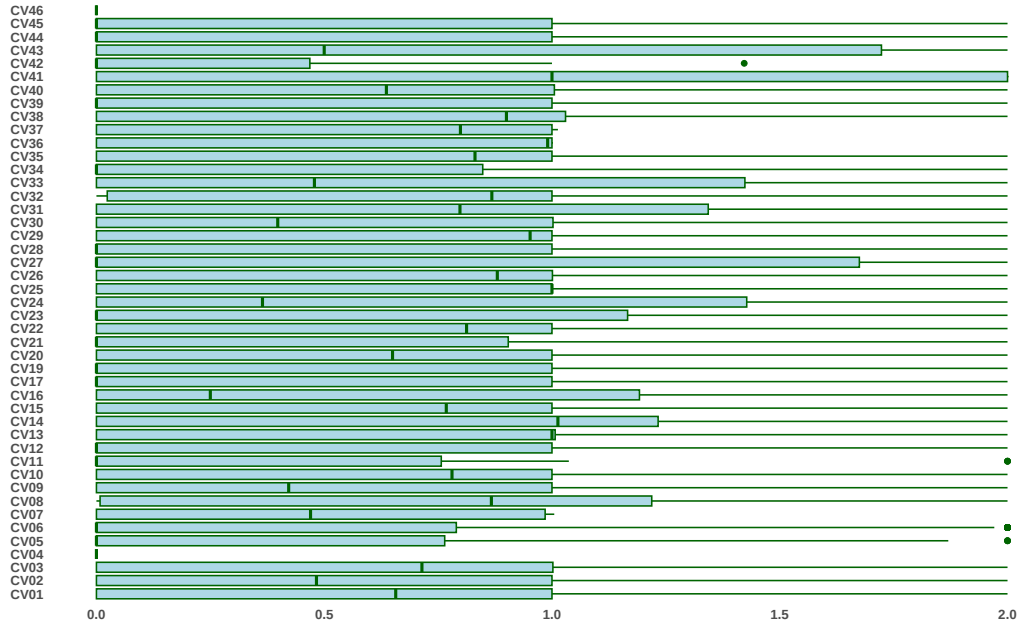


Figure 4.18: Codecov CRAP Metric: distribution of the delta CRAP metric across revisions.

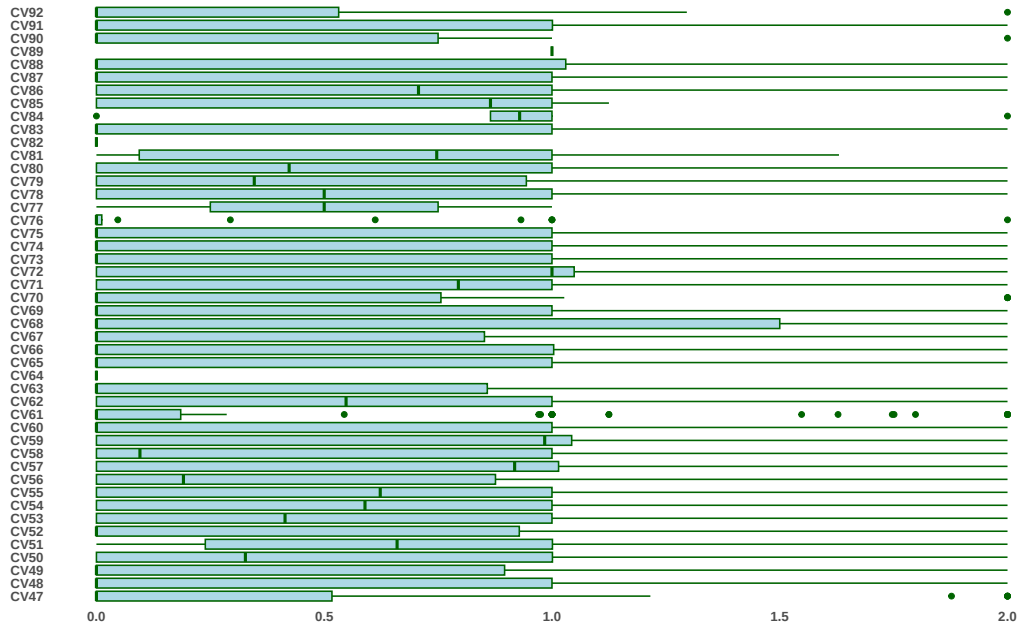


Figure 4.19: Codecov CRAP Metric Cont: distribution of the delta CRAP metric across revisions.

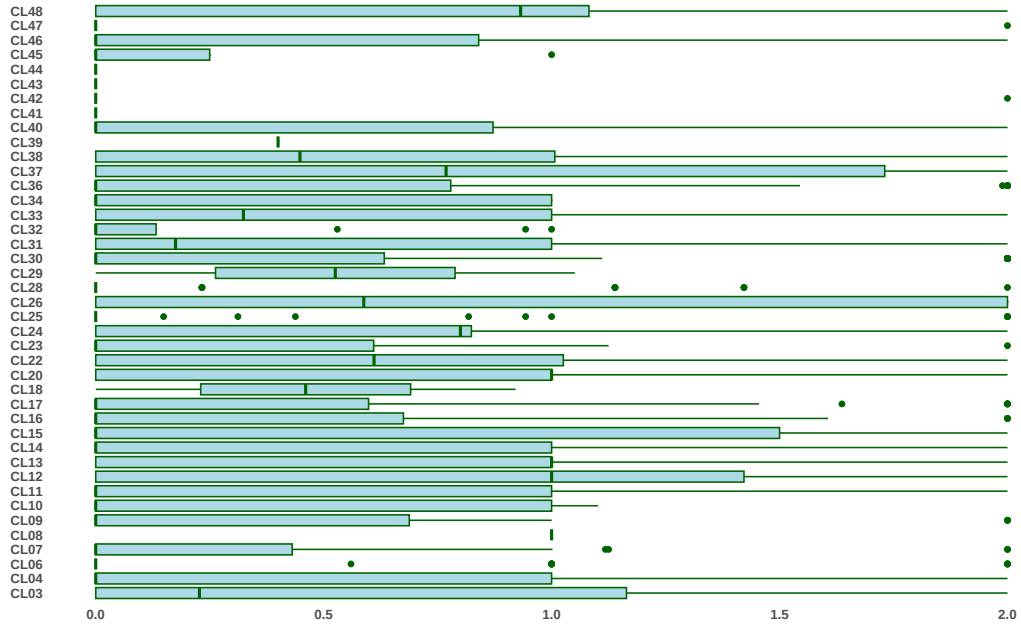


Figure 4.20: CRAP Metric Coveralls: distribution of the delta CRAP metric across revisions.

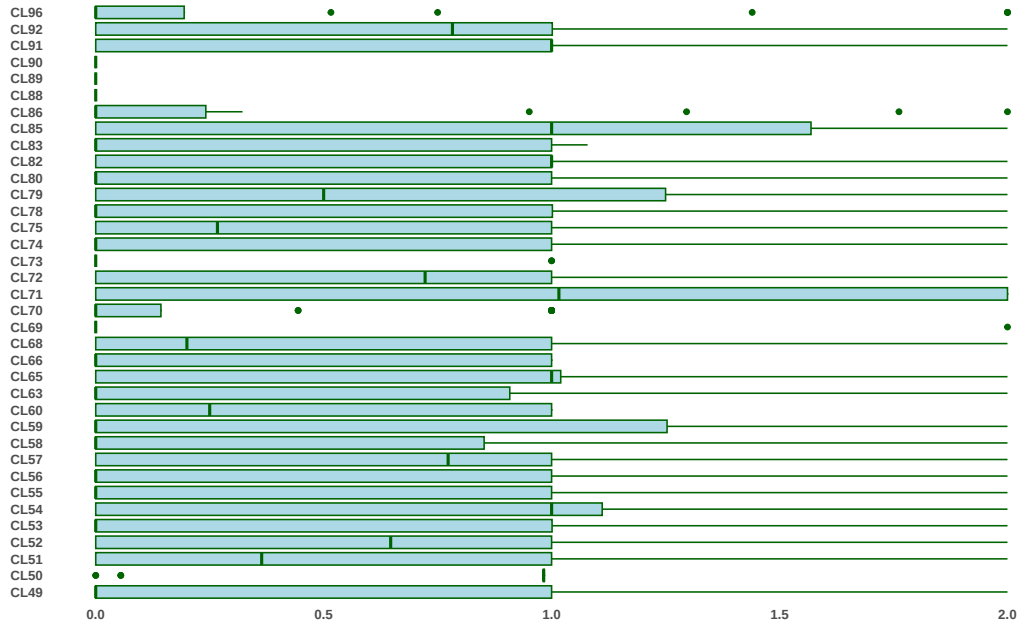


Figure 4.21: CRAP Metric Coveralls Cont: distribution of the delta CRAP metric across revisions.

The distribution of patch coverage in patches modifying source files only emphasises a notable presence of patches with testing. Analysing Spearman and Kendall coefficients in Table 4.2 further corroborates a moderate correlation between CRAP metric and patch coverage. This correlation is expected, given that the CRAP metric inherently incorporates coverage as part of its mathematical expression. Notably, the replacement of method coverage in Equation (3.2) with patch coverage was intentional, focusing on evaluating the CRAP metric for newly added code.

Project CL{42,69} stands out in its distribution, with changes concentrated solely at the edges (0 and 2) of the delta CRAP metric. This implies that the project has patches with either zero patch coverage or 100% coverage for all changes. The patch distribution graph confirms this observation (refer to Figures 4.5 and 4.6). In summary, the analysis suggests that a majority of projects exhibit a moderate level of CRAP metric.

Table 4.2: Projects Coverage Service Correlations

Relation	Codecov				Coveralls			
	Spearman		Kendall		Spearman		Kendall	
	ρ	d	τ	d	ρ	d	τ	d
Coverage v/s Patch Cov	0.0598	0.1198	0.0494	0.0989	0.022	0.044	0.018	0.036
Maintainability v/s Patch Cov	**0.469**	**1.062**	**0.424**	**0.936**	**0.426**	**0.942**	**0.398**	**0.868**
Maintainability v/s Coverage	-0.048	-0.096	-0.0360	-0.072	0.032	0.0644	0.0262	0.0524
CRAP v/s Patch Cov	**0.366**	**0.7866**	**0.329**	**0.697**	**0.337**	**0.716**	**0.314**	**0.661**
CRAP v/s Coverage	-0.046	-0.092	-0.036	-0.072	0.0339	0.0678	0.0269	0.0538

Finally, it is worth noting that because the complexity value utilised in calculating the delta CRAP metric spans from 0 to 1 (from DMM definition), and the delta CRAP metric itself being a polynomial, the outcome may be slightly skewed. This is because when any small number is squared or cubed, it typically follows a decreasing logarithmic scale, resulting in a significant reduction in magnitude. Consequently, it is not conclusively asserted that the question has been successfully addressed. Further studies are encouraged to explore this inquiry utilising alternative definition(s) of delta complexity to assess changes or introduced complexity alongside patch coverage.

4.5 Chapter Summary

This chapter presented and discussed the results of the report. A number of exploratory questions were repeated to characterise a patch. It is observed that patches are

generally small, consisting of 1 to 100 changed lines/statements. Secondly, the results showed that test code is rarely changed in isolation from application code. That is, either both application logic code and test code are changed in the same commit or only application logic code is modified. It is very rare that commits modify test code independently. In answering the question regarding the distribution of patch coverage across revisions, it is noted that patches are either covered or not covered at all, with a majority covered. Very seldom do engineers opt to partially cover their commits. Isolating commits which modified source code from commits which modified none-source code showed that patch maintainability varies across revisions in different projects, ranging from projects which are unmaintainable to projects which are fully maintainable, according to the DMM score. Lastly, the CRAP metric index showed that project's CRAPiness fell between 0 to 1, indicating projects were less CRAPpy. This is on *par* with the distribution of patch coverage, especially considering that coverage is a component of the mathematical expression of the CRAP metric. This also validates the argument that coverage does indeed improve the CRAPiness of a code base.

5

Conclusion

Three primary research questions were investigated in this report. A summary of the findings to each of the studied questions in presented in Table 5.1.

5.1 Key Contributions

In prior research, considerable attention has been given to studying quality evolution from a release perspective, offering insights into overall system at specific stable point in time. However, this approach is less beneficial for those actively involved in maintaining and evolving the source code, such as developers, testers, and quality engineers. To date, two studies (Hilton *et al.* [18] and Marinescu *et al.* [17]) have explored quality evolution from a fined-grained perspective, using testing as an attribute of maintainability, focusing specifically on patch coverage. This study extends the work of Hilton *et al.* and Marinescu *et al.* in assessing the dynamics of code coverage using patch-by-patch analysis. In addition to examining patch coverage, another dimension, patch maintainability, is explored, utilising a variant of the SIG-MM known as DMM for measuring maintainability. In addition to the maintainability dimension, a variant of the CRAP metric, called the delta CRAP metric is derived and employed to measure patch specific CRAP metric.

Concerning the initial research question - how is patch coverage distributed across the revisions, analysis of the entire build history holistically indicates a notable focus on altering non-source code files across a majority of projects. When examining patches modifying source code files only, it becomes apparent that they are typically either fully covered or entirely uncovered, with partial coverage being a rare occurrence. These findings align with observations from Marinescu *et al.* Consequently, the first research

Table 5.1: Summary of Research Findings

Research Question	**Finding**
1. What is the distribution of coverage across revisions?	Combining patches modifying both none-source and source code reveal a predominate pattern of uncovered patches. However, isolating patches modifying source code only reveal that patches are generally either entirely covered or uncovered at all. Very seldom do engineers opt for partial coverage. This finding is in alignment with that of Marinescu *et al.*, yet, contradicts that of Hilton *et al.* who noted no discernible pattern.
2. How maintainable is a typical patch, how does maintainability vary across revisions?	Studying patches modifying source code only indicate that a large number of patches across the projects are maintainable, as measure by the DMM. However, it is noted that patch coverage does not necessarily improve maintainability of the patches.
3. To what extent is a typical patch considered "CRAPpy"?	The CRAP metric value for a majority of revisions across the projects were between 0 to 1, indicating that a majority of revisions actually had test incorporated. This is first corroborated by the defining equation of the CRAP metric which explicitly shows that the CRAP metric is a function of coverage. Secondly, the patch coverage distribution plot supports this by showing a majority of revisions between the (75% to 100%) bin.

question leads to the conclusion that patch coverage distribution predominantly resides within the 0% and 100% bins. Furthermore, it is observed that there is no correlation between patch coverage and overall system coverage, indicating that the measurement of patch coverage functions as an independent entity. This absence of a relationship suggests that analysing patch coverage alone may not provide significant insights and could potentially overlook important information regarding actual coverage patterns. Therefore, it is recommended to employ patch-by-patch coverage analysis in conjunction with overall system coverage for a more comprehensive understanding.

In answering the question "how maintainable are patches", it is noted that maintainability, as measured by DMM, varies among patches with no discernible consistent pattern. However, a significant proportion of patches across multiple projects display low DMM scores, indicating poor maintainability. Moreover, there is a moderate to strong correlation between DMM and patch coverage, suggesting that thorough patch testing positively influences patch maintainability.

Finally, an empirical investigation of the CRAP metric at a commit level is conducted. A variant, devised for measuring the patch CRAP metric using patch coverage and patch complexity, is formulated. Findings reveal that the upper bound value of the CRAP metric for most projects is one, indicating that the majority of patches possess a low CRAP metric. Since the majority of patches were entirely uncovered, the polynomial expressions defining delta CRAP metric reduces to a second order (quadratic) polynomial of the complexity value. The results of the CRAP observation disprove the previously mentioned assertion that extensive patch improves (lessens) change-risk anti-patterns. Further exploration could be done through examining the relationship between testing and complexity as depicted in the CRAP metric equation.

The primary contributions with respect to extending the existing studies are:

1. Increased dataset size - The use of coverage services as the sole source of coverage information allows for a much large dataset to be collected and analysed with less effort than using bespoke solutions (i.e downloading, compiling and collecting coverage reports) for determining patch coverage. With this in mind over 90000 patches spanning over 201 projects developed in 17 programming languages were extracted and analysed.

2. Patch CRAP metric - A variant of the CRAP metric, denoted patch CRAP metric is formulated with the use of patch coverage and patch complexity.

This metric is meant to measure and assess the CRAP metric for smallest developmental changes.

3. Coverage Service Data evaluation - A Comprehensive interrogation of the data provided by coverage services is conducted. Notably, while coverage service are rich of information, it is imperative for end users, especially data miners to be cautious of the data itself. Data cleaning, processing, and verification methods are necessary to enhance the integrity of the data.

5.2 Threats to Validity

5.2.1 Models and Definitions

The study employs a variant of the Software Group Improvement's Group Maintainability Model (SIG-MM) called Delta Maintainability Model (DMM) to measure patch maintainability, a model well known in software engineering, supported by empirical data and studies. The literature survey includes studies supporting and rejecting the model, with a majority finding it useful. Additionally, the CRAP metric is used to study change risk. A variant of the CRAP metric specifically designed to focus on modified lines is used. The model is first tested successfully on a test repository. The CRAP metric has not been empirically evaluated through research studies, however, its utility in static and dynamic code analysis tools such as NDepend suggest that there is a use case for it. However, it is acknowledged that this is the first empirical research to investigate this metric, as such, quantifying its effectiveness is left open for further investigations.

5.2.2 Sample Size & Programming Languages

The large-scale study encompasses 201 projects and over 90000 builds across 17 diverse programming languages. Projects were selected based on prior research and expanded by including actively maintained projects with a substantial number of builds. To provide confidence in the representativeness of projects, the selection criteria is robustly designed to specifically mine projects with multiple contributors and a considerable number of lines of code in order to avoid "toy" projects. Identification of projects was done through a reputable repository [152] with a curated list of open-source projects calling for contributors. Data mining was performed with the consideration of not abusing or overloading public web APIs of coverage service tools.

5.2.3 Data Integrity

Trusting data from coverage platforms, the study enhances data integrity through robust preparation, processing, and cleaning mechanisms, eliminating outliers and corrupted data. The coverage services were initially tested in a Python-based experimental repository configured for automated unit testing and with manual calculations to validate API data consistency. That is in order verify both mathematical formula and methodology used by coverage services to compute and capture patch coverage, a three-step approach was conducted:

1. Automated Testing - The automated testing essentially involved configuring a CI pipeline with GitHub action. As part of the build steps, a the application is build and unit tests are ran and the coverage report is the uploaded to both Codecov and Coveralls. The coverage service web APIs are then used to query the patch coverage
2. Manual Calculation - Outlined in Appendix B, the second step was to, by hand, calculate the patch coverage of the different commits.
3. Communication with Codecov team - The last step was to verify the above results with the Codecov team through issue request in GitHub, especially in the case of difference results. The discussions in the GitHub issues involved discussions, pertaining validating the mathematical expression used in computing patch coverage and a step-by-step guideline on how the web API gets the coverage reports from build output. Additionally, the communication channel discussed at great length typical causes which often lead to incorrect data in both the API and user interface.

In case of inconsistencies, communication with maintainers of the platforms was established, leading to issue resolution. The applied techniques are deemed sufficient for filtering out inconsistent data, and improving overall integrity. All branches are mined to represent project evolution comprehensively, while merge commits are excluded due to lack of added value and potential data duplication. Concerns acknowledges the "fatal" git checkout of commits as stated in Section 3.4.1, particularly regarding to how it impacts the one-to-one mapping of builds to patches. Due to the nature of development workflows, no superior data preparation techniques exist for dangling or missing commits. The script involved implementation of logging and inspected the logs, and verified in GitHub whether the commits were no longer available or if there were script issues. From the pool of commits implicated in git

checkout issues, a select few were scrutinised for their presence on GitHub. It was discovered that these chosen commits were not accessible (i.e no longer available), a situation attributed to discrepancies in development workflows.

Another concern arises regarding the use of the Exponentially Weighted Moving Average (EMA) for mitigating sudden drops in overall coverage. This method introduces a limitation whereby coverage values potentially indicative of genuine decreases are influenced by this smoothing filter. An illustrative case is that previously discussed in Section 4.3.2, concerning project CV27 commit 90bf5335, which involved changes to 87 files and resulted in a code churn exceeding $70K$ lines. Although this commit signifies a legitimate alteration, its significant impact on code coverage prompts the EMA to potentially mask this behaviour if it falls within the 30-point weight average utilised in the EMA. Thus, this trade-off is acknowledged, and efforts were made to explore alternative approaches, including manual exclusion of commits where feasible.

5.3 Research Implications

The implications of the findings are stated as follows:

5.3.1 Implications for Build and Coverage Tools Providers

The contributions and findings above lead to the following implications for coverage tool and CI providers:

- Noting a significant number of changes modifying non-code files, it is recommended that developers and tool maintainers implement more robust methods to exclude triggering builds for files which would not impact coverage, specifically configuration files, yet, excluding scripts such as bash, Powershell etcetera. While most CI tools, already offer this functionality, developers should configure their pipelines to avoid triggering builds based on file extensions. Considering the cost of build agents and processing power, enabling such configuration at the build agent level is suggested.
- The study uncovered numerous data inconsistencies, including sudden spikes, jumps, and corrupted data, without concrete evidence from the coverage tool to explain such dynamics. Verification involved creating an experimental project

to validate tool functionality. Assistance from coverage service tool maintainers resolved issues promptly. Aligning with Hilton *et al.* [18], it is advocated that coverage service tools offer developers a means to discern whether tests cover application logic deterministically or non-deterministically. This means coverage service tools should offer a way of mechanisms to check a test suite always tests the same part(s) of the program when executed or if the path tested changes upon every run. To enhance comprehension of changes impact, presenting flipping lines with increases/decreases in coverage is suggested. As emphasised by Hilton *et al.*, these tools highlight *what* changed, but there is a need for greater focus on the *how/why* to aid developers in understanding the impact of their modifications.

5.3.2 Implications for Developers

In the observation of the impact of patch-by-patch analysis, one notes that patch coverage, unlike overall system coverage, offers a method for comprehending the intricacies and nuances of development in more profound detail. The examination of relationship between overall system coverage and patch metrics (coverage, DMM, CRAP) indicate that patch-by patch analysis alone does not yield useful insights. It is therefore, reiterated that for a more comprehensive and conclusive data, patch analysis should be used in conjunction with the overall system data. Furthermore, considerable effort should also be directed towards the maintainability of changes, as there exists a moderate to strong correlation between the patch coverage and patch maintainability i.e patching testing will positively influence the maintainability of the system.

5.3.3 Implications for Researchers

As highlighted earlier, obtaining coverage statistics is challenging due to version mismatches or incompatible dependencies from an evolutionary perspective. Coverage service tools offer abundant data for research investigations, yet, researchers must exercise caution. Thorough data analysis, advanced processing and cleaning mechanisms are necessary for meaningful insights. Accepting the data without proper scrutiny can alter the study trajectory and result in inaccurate conclusions. Nevertheless, using coverage tools alleviates the burden of dependencies, allowing researchers more time to focus on their actual study. Moreover, coverage services, offer researchers an opportunity to gather more data, resulting in a more comprehensive and data-supported

study.

5.4 Further Research

While the studied projects were actively maintained, active maintenance does not necessarily imply ongoing development; a project might be mature with only upgrades and updates to versions. Exploring projects in their infancy or mid-stages could provide insights into how project quality evolves. Actively developed projects are expected to reveal valuable insights. Regarding patch coverage, it is worthwhile to investigate its correlation with developer sentiment and community requirements. This includes determining if patch coverage is enforced by the community before a pull request is merged or if it is solely the developers' prerogative to decide whether to write tests. The author encourages other studies to replicate this work on the CRAP metric to assess its utility.

References

[1] M.M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[2] M. M. Lehman and L. A. Belady. Program evolution processes of software change. *A.P.1.C. Studies in Data Processing*, 27:9–39, November 1985.

[3] Andy Zaidman, Bart Rompaey, Arie Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, June 2011.

[4] P. Runeson. A survey of unit testing practices. *IEEE Software*, 23(4):22–29, 2006.

[5] Oscar Nierstrasz and Serge Demeyer. Object-oriented reengineering patterns. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, page 734–735, USA, 2004. IEEE Computer Society.

[6] Leon Moonen, Arie van Deursen, Andy Zaidman, and Magiel Bruntink. *On the Interplay Between Software Testing and Evolution and its Effect on Program Comprehension*, pages 173–202. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

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

[8] Ayelet Israeli and Dror G. Feitelson. The linux kernel as a case study in software evolution. *J. Syst. Softw.*, 83:485–501, 2010.

[9] Taranjeet Kaur, Nisha Ratti, and Parminder Kaur. Applicability of Lehman Laws on open source evolution: A case study. *International Journal of Computer Applications*, 93(18):40–46, May 2014.

[10] Nicholas Drouin and Mourad Badri. Investigating the applicability of lehman's laws of software evolution using metrics: An empirical study on open source software. 2016.

[11] Muslija Adnan and Enoiu Eduard. On the correlation between testing effort and software complexity metrics. October 2018.

[12] Nguyen Duc Anh. Ethe impact of design complexity on software cost and quality. Master's thesis, Technical University of Kaiserslautern, September 2010.

[13] Ahamed Thahseen, Navathanaraj Aaron, Thakshila Nanayakkara, Arsath Farves, Dilshan Silva, and Poojani Gunathilake. Analyzing the impact of software testing on software maintainability. Technical report, Research Square, May 2023.

[14] Liguo Yu and Alok Mishra. An empirical study of lehman's law on software quality evolution. *International Journal of Software Informatics*, 7:469–481, November 2013.

[15] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, page 426–437, New York, NY, USA, 2016. Association for Computing Machinery.

[16] Jeffrey Fairbanks, Akshharaa Tharigonda, and Nasir Eisty. Analyzing the effects of ci/cd on open source repositories in github and gitlab. pages 176–181, May 2023.

[17] Paul Marinescu, Petr Hosek, and Cristian Cadar. Covrig: A framework for the analysis of code, test, and coverage evolution in real software. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 93–104, New York, NY, USA, 2014. Association for Computing Machinery.

[18] Michael Hilton, Jonathan Bell, and Darko Marinov. A large-scale study of test coverage evolution. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 53–63, 2018.

[19] Chris Kemerer and Sandra Slaughter. An empirical approach to studying software evolution. *Software Engineering, IEEE Transactions on*, 25:493 – 509, August 1999.

[20] Arthur H. Watson Dolores R. Wallace and Thomas J. McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric. *Computer Systems Technology*, 8(2):235–500, August 1996.

[21] Lionel Claude Briand, Jürgen Wüst, Stefan V. Ikonomovski, and Hakim Lounis. A comprehensive investigation of quality factors in object-oriented designs: an industrial case study. 1998.

[22] Dr S.S Riaz Ahamed. Studying the feasibility and importance of software testing: An analysis. *International Journal of Engineering Science and Technology*, 1:119–128, December 2009.

[23] Abhijit Sawant, Pranit Bari, and Pramila Chawan. Software testing techniques and strategies. *International Journal of Engineering Research and Applications(IJERA)*, 2:980–986, June 2012.

[24] Serious Ben. Patch coverage. `https://seriousben.com/posts/2022-02-patch-coverage/`, Accessed February 2022.

[25] Tom Hu. Why patch coverage is more important than project coverage. `https://about.codecov.io/blog/why-patch-coverage-is-more-important-than-project-coverage/`, Accessed January 2024.

[26] Asem Omari. *Data mining for retail website design and enhanced marketing*. PhD thesis, Heinrich Heine University Düsseldorf, 2008.

[27] Seifedine Kadry. A new proposed technique to improve software regression testing cost. *International Journal of Security and its Applications*, 5, November 2011.

[28] Microsoft Corp. Github actions. `https://docs.github.com/en/actions`, Accessed June 2023.

[29] Kohsuke Kawaguchi. Jenkins: Build great things at any scale. `https://www.jenkins.io/`, Accessed June 2023.

[30] Kohsuke Kawaguchi. Build faster: Test your code in minutes. deploy with confidence. `https://www.travis-ci.com/`, Accessed June 2023.

[31] N. Cassee, B. Vasilescu, and A. Serebrenik. The silent helper: The impact of continuous integration on code reviews. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 423–434, Los Alamitos, CA, USA, February 2020. IEEE Computer Society.

[32] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. The impact of continuous integration on other software development practices: A large-scale empirical study. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 60–71, 2017.

[33] Godfrey and Qiang Tu. Evolution in open source software: a case study. In *Proceedings 2000 International Conference on Software Maintenance*, pages 131–142, 2000.

[34] Michael Godfrey and Qiang Tu. Growth, evolution, and structural change in open source software. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, IWPSE '01, page 103–106, New York, NY, USA, 2001. Association for Computing Machinery.

[35] G. Robles, J.J. Amor, J.M. Gonzalez-Barahona, and I. Herraiz. Evolution and growth in large libre software projects. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, pages 165–174, 2005.

[36] Ratti Nisha, Kaur Parminder, and Taranjeet Kaur. Applicability of Lehman Laws on open source evolution. 93:40–46, May 2014.

[37] Cosmin Marsavina, Daniele Romano, and Andy Zaidman. Studying fine-grained co-evolution patterns of production and test code. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 195–204, 2014.

[38] Stanislav Levin and Amiram Yehudai. The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–46, September 2017.

[39] Mamdouh Alenezi, Mohammed Akour, and Hiba Alsghaier. The impact of co-evolution of code production and test suites through software releases in open source software systems. *International Journal of Innovative Technology and Exploring Engineering*, 9:2737–2739, November 2019.

[40] Sinan Wang, Ming Wen, Yepang Liu, Ying Wang, and Rongxin Wu. Understanding and facilitating the co-evolution of production and test code. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 272–283, 2021.

[41] Weifeng Sun, Meng Yan, Zhongxin Liu, Xin Xia, Yan Lei, and David Lo. Revisiting the identification of the co-evolution of production and test code. *ACM Transactions on Software Engineering and Methodology*, 32:1–37, 2023.

[42] Alberto Savoia. Agitating thoughts & ideas: Pardon my french, but this code is c.r.a.p. `https://www.artima.com/weblogs/viewpost.jsp?thread=210434`, Accessed November 2023.

[43] Alberto Savoia. Agitating thoughts & ideas: Pardon my french, but this code is c.r.a.p (2). `https://www.artima.com/weblogs/viewpost.jsp?thread=210575`, Accessed November 2023.

[44] Buvaneswari Venkataraman and William Ward Jr. An introduction to software quality. Technical report, US Army Corpsof Engineers Waterways Experiment Station, June 1999.

[45] Ronan Fitzpatrick. Software quality: Definitions and strategic issues. Master's thesis, Dublin Institute of Technology, School of Computing, 1996.

[46] David Chappell. The three aspects of software quality: Functional, structural and process. `http://www.davidchappell.com/writing/white_papers/The_Three_Aspects_of_Software_Quality_v1.0-Chappell.pdf`, Accessed October 2023.

[47] Joseph P. Cavano and James A. McCall. A framework for the measurement of software quality. *SIGSOFT Softw. Eng. Notes*, 3(5):133–139, January 1978.

[48] S. Chulani and Barry Boehm. Software engineering economics. In *Software Engineering Body of Knowledge (SWEBOK)*, chapter 12, pages 203–225. Springer, January 2007.

[49] R.G. Dromey. A model for software product quality. *IEEE Transactions on Software Engineering*, 21(2):146–162, 1995.

[50] R.G. Dromey. Cornering the chimera [software quality]. *IEEE Software*, 13(1):33–43, 1996.

[51] Jim A. McCall, Paul K. Richards, and Cene F. Walters. Factors in software quality: Concept and definitions of software quality. *IEEE Software*, 1-3(1):33–43, 1977.

[52] ISQS ISO. Iso/iec 25010 : 2011 systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models. 2011.

[53] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[54] E. Burton Swanson. The dimensions of maintenance. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, page 492–497, Washington, DC, USA, 1976. IEEE Computer Society Press.

[55] P. Oman and J. Hagemeister. Metrics for assessing a software system's maintainability. In *Proceedings Conference on Software Maintenance 1992*, pages 337–344, 1992.

[56] D. Coleman, D. Ash, B. Lowther, and P. Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, 1994.

[57] Jymit Khondhu, Andrea Capiluppi, and Klaas-Jan Stol. Is it all lost? a study of inactive open source projects. pages 61–79, June 2013.

[58] Tjaša Heričko and Boštjan Šumak. Exploring maintainability index variants for software maintainability measurement in object-oriented systems. *Applied Sciences*, 13(5), 2023.

[59] V.Y. Shen, S.D. Conte, and H.E. Dunsmore. Software science revisited: A critical analysis of the theory and its empirical support. *IEEE Transactions on Software Engineering*, SE-9(2):155–165, 1983.

[60] Tobias Kuipers and Joost Visser. Maintainability index revisited-position paper-tobias kuipers. 2007.

[61] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. pages 30 – 39, October 2007.

[62] Tiago L. Alves, Christiaan Ypma, and Joost Visser. Deriving metric thresholds from benchmark data. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, 2010.

[63] Tiago L. Alves, Jose Pedro Correia, and Joost Visser. Benchmark-based aggregation of metrics to ratings. In *2011 Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement*, pages 20–29, 2011.

[64] Robert Baggen, José Pedro Correia, Katrin Schill, and Joost Visser. Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal*, 20(2):287–307, June 2012.

[65] Marco di Biase, Ayushi Rastogi, Magiel Bruntink, and Arie van Deursen. The delta maintainability model: Measuring maintainability of fine-grained code changes. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 113–122, 2019.

[66] Ijeacs Uk. Software complexity measurement: A critical review. *International Journal of Engineering and Applied Computer Science (IJEACS)*, 01:12–16, December 2016.

[67] Joseph P. Kearney, Robert L. Sedlmeyer, William B. Thompson, Michael A. Gray, and Michael A. Adler. Software complexity measurement. *Communications of the ACM*, 29(11):1044–1050, November 1986.

[68] Jon Wade and Babak Heydari. Complexity: Definition and reduction techniques some simple thoughts on complex systems. *CEUR Workshop Proceedings*, 1234:213–226, January 2014.

[69] Victor R Basili. Qualitative software complexity models: A summary. *Tutorial on Models and Methods for Software Management and Engineering, IEEE Computer Society Press*, 1980.

[70] Rajiv D. Banker, Srikant Datar, and Dani Zweig. Software complexity and maintainability. pages 247–255, January 1989.

[71] Edward E. Ogheneovo. On the relationship between software complexity and maintenance costs. *Journal of Computer and Communications*, 02:1–16, January 2014.

[72] Alice T. Lee, Todd Gunn, Tuan Pham, and Ron Ricaldi. Software analysis handbook: Software complexity analysis and software reliability estimation and prediction. 1994.

[73] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[74] KHassan Raza Bhatti. Automatic measurement of source code complexity. Master's thesis, Luleå University of Technology - Department of Computer Science, Electrical and Space Engineering, January 2010.

[75] Sukhdev Singh Ghuman. Software testing techniques. *International Journal of Computer Science and Mobile Computing*, 3(10):988–993, October 2014.

[76] Itti Hooda and Rajender Singh Chhillar. Software test process, testing types and techniques. *International Journal of Computer Applications*, 111(13):10 – 14, February 2015.

[77] Lawanna Adtha. The theory of software testing. *Intelligent Transportation Systems Journal*, 16:35–40, June 2012.

[78] Gaurav Saini and Kestina Rai. An analysis on objectives, importance and types of software testing. *International Journal of Computer Science and Mobile Computing*, 2(9):18–23, September 2013.

[79] Qian Yang, Jenny li, and David Weiss. A survey of coverage-based testing tools. *The Computer Journal*, 52, January 2006.

[80] Dr Muhammad Shahid and Suhaimi Ibrahim. An evaluation of test coverage tools in software testing. January 2011.

[81] Jorge Arturo Wong-Mozqueda, Robert Haines, and Caroline Jay. Is code quality related to test coverage? January 2015.

[82] Joan C. Miller and Clifford J. Maloney. Systematic mistake analysis of digital computer programs. *Commun. ACM*, 6(2):58–63, February 1963.

[83] Ben Smith and Laurie Williams. A survey on code coverage as a stopping criterion for unit testing. July 2019.

[84] Alberto Savoia. Google testing blog: This code is crap. `https://testing.googleblog.com/2011/02/this-code-is-crap.html`, Accessed November 2023.

[85] Bob Cramblit. Alberto Savoia on crappy code and the beauty of testing. `https://www.codeproject.com/Articles/20755/Alberto-Savoia-on-Crappy-Code-and-the-Beauty-of-Te`, Accessed August 2023.

[86] NDepend Blog. Improve your .NET code quality with NDepend: CRAP metric is a thing and it tells you about risk in your code. `https://blog.ndepend.com/crap-metric-thing-tells-risk-code/`, Accessed January 2024.

[87] Vard Antinyan and Miroslaw Staron. Mythical unit test coverage. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 267–268, 2019.

[88] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[89] Audris Mockus, Nachiappan Nagappan, and Trung T. Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *2009 3rd International*

Symposium on Empirical Software Engineering and Measurement, pages 291–301, 2009.

[90] L. Briand and D. Pfahl. Using simulation for assessing the real impact of test coverage on defect coverage. In *Proceedings 10th International Symposium on Software Reliability Engineering (Cat. No.PR00443)*, pages 148–157, 1999.

[91] Pavneet Singh Kochhar, Ferdian Thung, and D. Lo. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 560–564, 2015.

[92] Pavneet Singh Kochhar, David Lo, Julia Lawall, and Nachiappan Nagappan. Code coverage and postrelease defects: A large-scale study on open source projects. *IEEE Transactions on Reliability*, 66(4):1213–1228, 2017.

[93] Xia Cai and Michael R. Lyu. The effect of code coverage on fault detection under different testing profiles. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, May 2005.

[94] Gregory Gay, Matt Staats, Michael Whalen, and Mats P. E. Heimdahl. The risks of coverage-directed test case generation. *IEEE Transactions on Software Engineering*, 41(8):803–819, 2015.

[95] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 72–82, New York, NY, USA, 2014. Association for Computing Machinery.

[96] Ranjith Purushothaman and Dewayne E. Perry. Towards understanding software evolution : One-line changes. 2002.

[97] Raphael Pereira de Oliveira, Alcemir Rodrigues Santos, Eduardo Santana de Almeida, and Gecynalda Soares da Silva Gomes. Evaluating lehman's laws of software evolution within software product lines industrial projects. *Journal of Systems and Software*, 131:347–365, 2017.

[98] Vard Antinyan, Miroslaw Staron, Wilhelm Meding, Per Österström, Henric Bergenwall, Johan Wranker, Jörgen Hansson, and Anders Henriksson. Monitoring evolution of code complexity in agile/lean software development - a case study at two companies. 2013.

[99] Vard Antinyan, Miroslaw Staron, Jörgen Hansson, Wilhelm Meding, Per Österström, and Anders Henriksson. Monitoring evolution of code complexity and magnitude of changes. *Acta Cybern.*, 21(3):367–382, August 2014.

[100] Huihui Liu, Xufang Gong, Li Liao, and Bixin Li. Evaluate how cyclomatic complexity changes in the context of software evolution. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 02, pages 756–761, 2018.

[101] Mohammed Akour and Mamdouh Alenezi. Test suites effectiveness evolution in open source systems: empirical study. *Indonesian Journal of Electrical Engineering and Computer Science*, 2020.

[102] Oosterwaal Sebastiaan. Combining source code and test coverage changes in pull requests. Master's thesis, Software Engineering Research Group, Department of Software Technology Faculty EEMCS, Delft University of Technology Delft, the Netherlands, February 2015.

[103] Sebastiaan Oosterwaal, Arie van Deursen, Roberta Coelho, Anand Ashok Sawant, and Alberto Bacchelli. Visualizing code and coverage changes for code review. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 1038–1041, New York, NY, USA, 2016. Association for Computing Machinery.

[104] Arthur Molnar and Simona Motogna. Discovering maintainability changes in large software systems. In *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, IWSM Mensura '17, page 88–93, New York, NY, USA, 2017. Association for Computing Machinery.

[105] Anita Ganpati, Arvind Kalia, and Hardeep Singh. Maintainability index over multiple releases: A case study php open source software. *International journal of engineering research and technology*, 1, 2012.

[106] Gerta Kapllani, Ilya Khomyakov, Ruzilya Mirgalimova, and Alberto Sillitti. An empirical analysis of the maintainability evolution of open source systems. In Vladimir Ivanov, Artem Kruglov, Sergey Masyagin, Alberto Sillitti, and Giancarlo Succi, editors, *Open Source Systems*, pages 78–86, Cham, 2020. Springer International Publishing.

[107] Arthur-Jozsef Molnar and Simona Motogna. Longitudinal evaluation of open-source software maintainability. In *International Conference on Evaluation of Novel Approaches to Software Engineering*, 2020.

[108] Rüdiger Lincke and Welf Löwe. Compendium of software quality standards and metrics - version 1.0. Technical report, 2007.

[109] Raimund Moser, Marco Scotto, Alberto Sillitti, and Giancarlo Succi. Does xp deliver quality and maintainable code? XP'07, page 105–114, Berlin, Heidelberg, 2007. Springer-Verlag.

[110] Dennis Bijlsma, Miguel Ferreira, Bart Luijten, and Joost Visser. Faster issue resolution with higher technical quality of software. *Software Quality Journal - SQJ*, 20:1–21, June 2012.

[111] Stanislav Strečanský, Peterand Chren and Bruno Rossi. Comparing maintainability index, sig method, and sqale for technical debt identification. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, page 121–124, New York, NY, USA, 2020. Association for Computing Machinery.

[112] Jean-Louis Letouzey and Michel Ilkiewicz. Managing technical debt with the sqale method. *IEEE Software*, 29(6):44–51, 2012.

[113] Jean-Louis Letouzey. The sqale method for managing technical debt : Definition document. Technical Report Version: 1.1, March 2016.

[114] Pavlo Skladannyi, Olena Nehodenko, Svitlana Shevchenko, Oksana Zolotukhina, and Vitalii Nehodenko. Modified delta maintainability model of object-oriented software. pages 1–8, December 2022.

[115] Bart J.H. Luijten. The influence of software maintainability on issue handling. Master's thesis, Delft University of Technology Delft, Department of Software Technology Faculty EEMCS, January 2009.

[116] Aigerim Issabayeva, Ariadi Nugroho, and Joost Visser. Issue handling performance in proprietary software projects. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 209–212, 2012.

[117] Joren Wijnmaalen. The relation between software maintainability and issue resolution time: A replication study. In *Seminar on Advanced Techniques and Tools for Software Evolution*, volume 2510, 2019.

[118] Oktay Turetken. Towards a maintainability model for business processes: Adapting a software maintainability model (position paper). In *2013 IEEE 1st International Workshop on Communicating Business Process and Software Models Quality, Understandability, and Maintainability (CPSM)*, pages 1–4, 2013.

[119] Ivano Malavolta, Roberto Verdecchia, Bojan Filipovic, Magiel Bruntink, and Patricia Lago. How maintainability issues of android apps evolve. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344, 2018.

[120] Codecov by Sentry. Code coverage: More than a metric. `https://about.codecov.io/`, Accessed November 2023.

[121] Nick Merwin and Lee Donahoe. Coveralls: deliver better code. `https://coveralls.io/`, Accessed November 2023.

[122] Docker Inc. Make better, secure software from the start. `https://www.docker.com/`, Accessed August 2023.

[123] GitHub Inc. Let's build from here: The world's leading ai-powered developer platform. `https://github.com/`, Accessed November 2023.

[124] Vishal Midha and Prashant Palvia. Factors affecting the success of open source software. *Journal of Systems and Software*, 85(4):895–905, 2012.

[125] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 92–101, New York, NY, USA, 2014. Association for Computing Machinery.

[126] Adam Tutko, Austin Z. Henley, and Audris Mockus. How are software repositories mined? a systematic literature review of workflows, methodologies, reproducibility, and tools, 2022.

[127] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. Understanding myths and realities of test-suite evolution. In *SIGSOFT FSE*, 2012.

[128] LEMUR Heavy Industries: lemurheavy/coveralls-public Repository. Api returning same commit hash yet different coverage percentage and date. `https://github.com/lemurheavy/coveralls-public/issues/1648`, Personal Communication July 2023.

[129] GitHub Inc. Dependabot - automated dependency updates built into github. `https://github.com/dependabot`, Accessed December 2023.

[130] Project coverage drops -.1 `https://community.codecov.com/t/project-coverage-drops-1-and-unable-to-pass-the-check/104`, Accessed December 2023.

[131] Coverage-dropped-by-60-after-merging-a-commit. https://community.codecov.com/t/coverage-dropped-by-60-after-merging-a-commit/4129, Accessed December 2023.

[132] Coverage is shown as 0% #590. https://github.com/lemurheavy/coveralls-public/issues/590, Accessed December 2023.

[133] coverage dropped from 70+% to 10%. https://github.com/lemurheavy/coveralls-public/issues/591, Accessed December 2023.

[134] Coveralls started reporting 0% but travis has always reported 100%. https://github.com/lemurheavy/coveralls-public/issues/848, Accessed December 2023.

[135] Coveralls does not work anymore. https://github.com/lemurheavy/coveralls-public/issues/1567, Accessed December 2023.

[136] Codecov by Sentinel: Codecov/feedback Repository. Patch coverage formula vs overall coverage ratio. https://github.com/codecov/feedback/issues/55, Personal Communication August 2023.

[137] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. Pydriller: Python framework for mining software repositories. *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 908–911, October 2018.

[138] Viewing source code. https://docs.codecov.com/docs/viewing-source-code, Accessed February 2024.

[139] Jorge Martínez Gil. Source code clone detection using unsupervised similarity measures. *ArXiv*, abs/2401.09885, 2024.

[140] Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates (LEA), 2 edition, 1988.

[141] R: A language and environment for statistical computing, r foundation for statistical computing. https://www.r-project.org/, Accessed Janauary 2024.

[142] Pavneet Singh Kochhar, Tegawendé F. Bissyandé, David Lo, and Lingxiao Jiang. An empirical study of adoption of software testing in open source projects. In *2013 13th International Conference on Quality Software*, pages 103–112, 2013.

[143] Pavneet Singh Kochhar, Tegawendé F. Bissyandé, David Lo, and Lingxiao Jiang. Adoption of software testing in open source projects–a preliminary study

on 50,000 projects. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 353–356, 2013.

[144] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie van Deursen. Mining software repositories to study co-evolution of production & test code. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 220–229, 2008.

[145] László Vidács and Martin Pinzger. Co-evolution analysis of production and test code by learning association rules of changes. In *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, pages 31–36, 2018.

[146] Zeeger Lubsen, Andy Zaidman, and Martin Pinzger. Using association rules to study the co-evolution of production & test code. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 151–154, 2009.

[147] Gerald Kipruto Kirui and Stephen Phillip Levitt. Matching production and test files: A comparison of filename and statement-based approaches. *Communications in Computer and Information Science(CCIS)*, 1878:3–18, 2023.

[148] Abdulkareem Alali, Huzefa Kagdi, and Jonathan I. Maletic. What's a typical commit? a characterization of open source software repositories. In *2008 16th IEEE International Conference on Program Comprehension*, pages 182–191, 2008.

[149] Alexander Sterk. Exploring code coverage in open-source development. Master's thesis, TUDelft: Software Engineering Research Group Department of Software Technology Faculty EEMCS, Delft University of Technology Delft, the Netherlands, June 2023.

[150] Christian R. Prause, Jürgen Werner, Kay Hornig, Sascha Bosecker, and Marco Kuhrmann. Is 100% Test Coverage a reasonable requirement? lessons learned from a space software project. *International Conference on Product-Focused Software Process Improvement(PROFES 2017)*, 10611:351–367, 2017.

[151] Codecov by Sentry. Unexpected coverage changes. `https://docs.codecov.com/docs/unexpected-coverage-changes#reasons-for-unexpected-changes`, Accessed November 2022.

[152] MunGell Shmavon Gazanchyan. Awesome first pr opportunities. `https://github.com/MunGell/awesome-for-beginners`, Accesssed June 2023.

[153] Marco di Biase, Ayushi Rastogi, Magiel Bruntink, and Arie van Deursen. The delta maintainability model: Measuring maintainability of fine-grained code changes technical report. Technical report, Delft University of Technology, May 2019.

Appendix A

Answers to Demographic Questions

Several studies have been conducted to understand the co-evolution (or lack thereof) between production "logic/application" code and test code. Hilton *et al.* and Marinescu *et al.* work, also investigated these phenomenon. As such, as part of this research these questions formed an exploratory phase of the research.

A.1 EQ-1 : Coevolution of Production and Test code

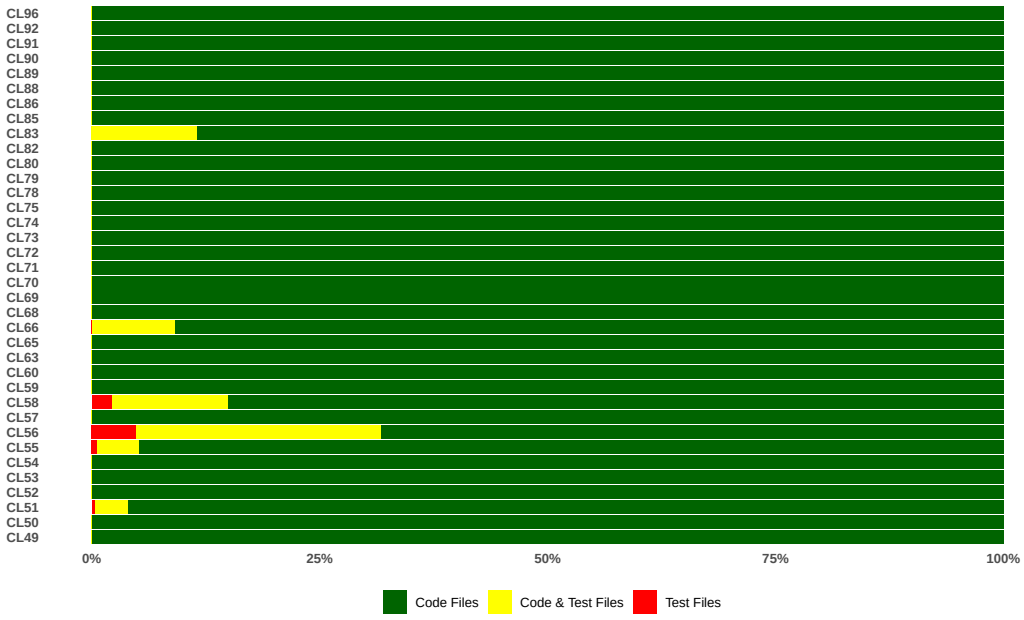


Figure A.1: Exploratory Question 1: Coveralls Cont - Coevolution of Production and Test code. Size of bar represents the proportion of builds over the total number of builds in each category.

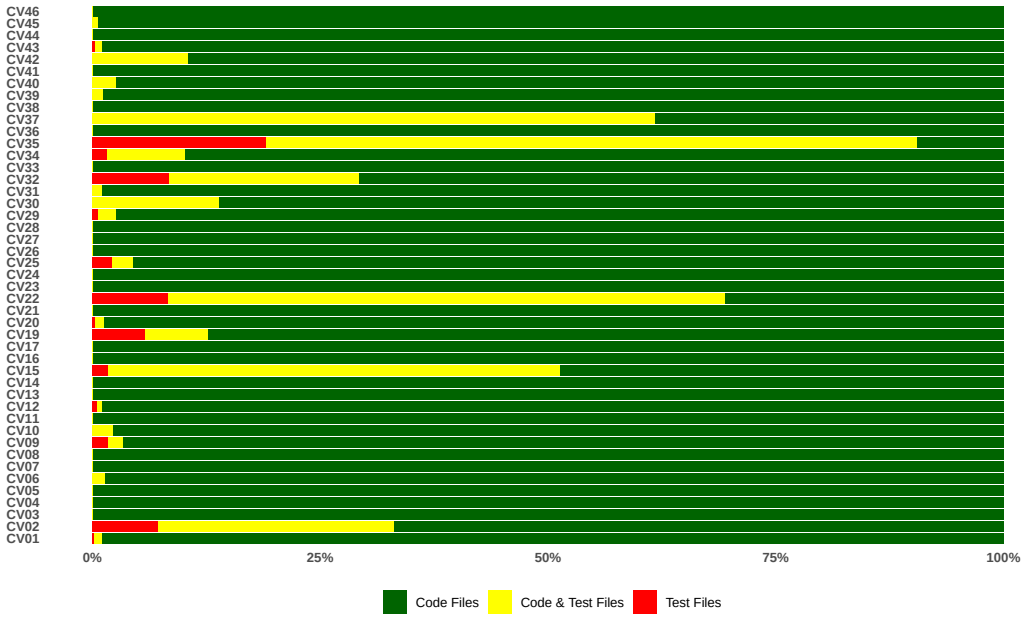


Figure A.2: Exploratory Question 1: Codecov - Coevolution of Production and Test code. Size of bar represents the proportion of builds over the total number of builds in each category.

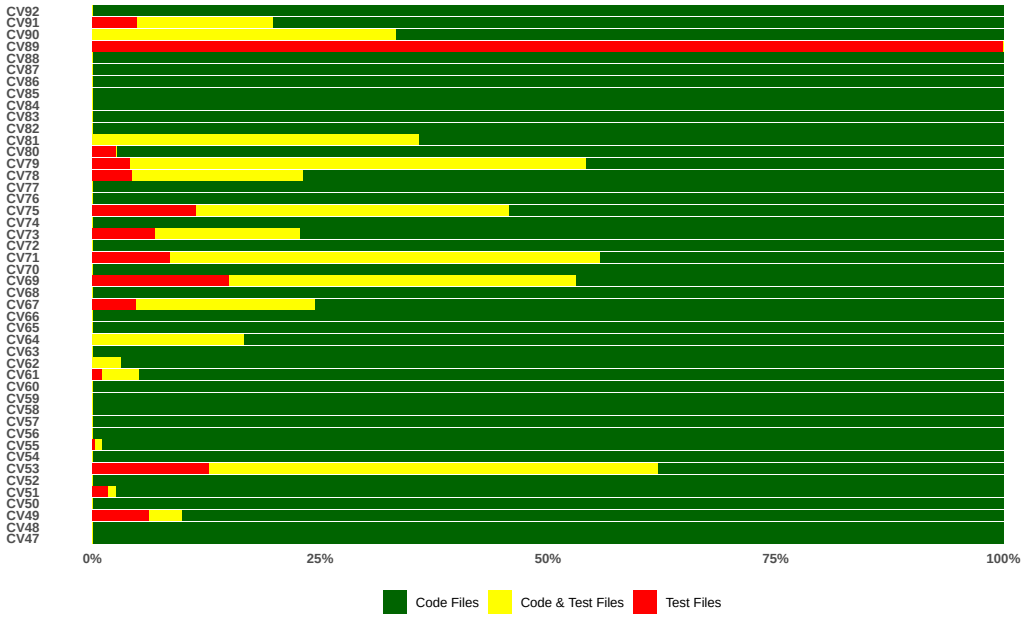


Figure A.3: Exploratory Question 1: Codecov Cont - Coevolution of Production and Test code. Size of bar represents the proportion of builds over the total number of builds in each category.

A.2 EQ-2: Typical Size of a Patch (Source Files Only)

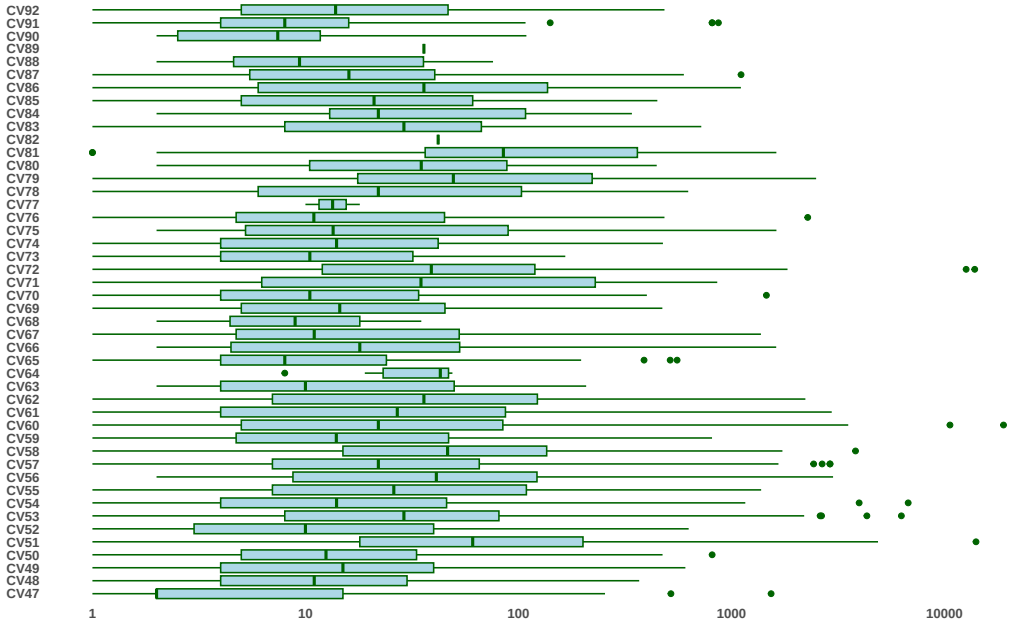


Figure A.4: Exploratory Question 2: Codecov Cont - Distribution of Source Code patch Size (Code Churn)

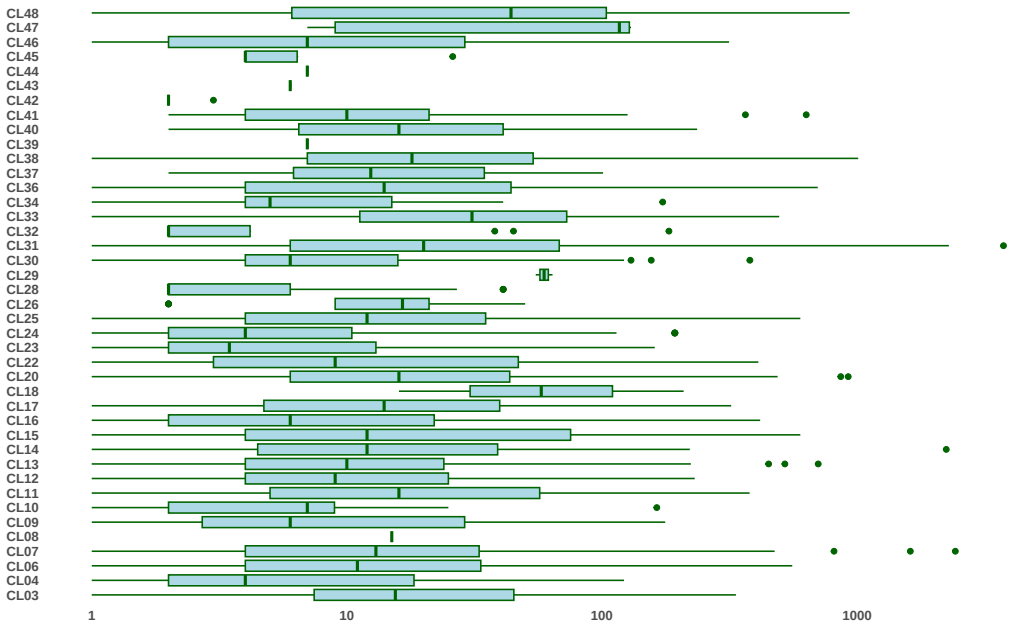


Figure A.5: Exploratory Question 2: Coveralls Cont - Distribution of Source Code patch Size (Code Churn)

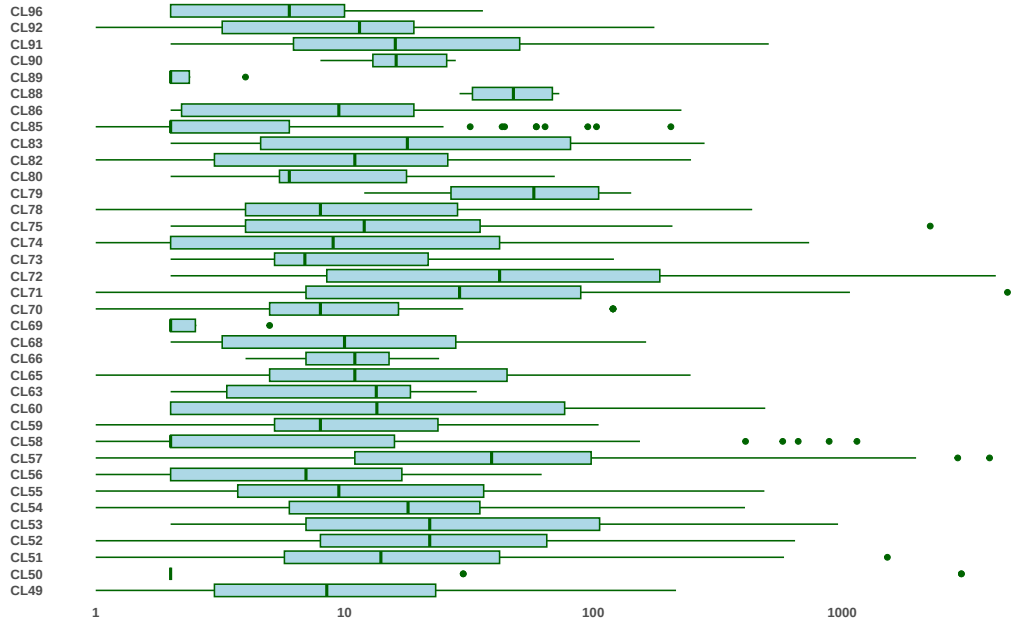


Figure A.6: Exploratory Question 2: Coveralls Cont - Distribution of Source Code patch Size (Code Churn)

Appendix B

Patch Coverage Manual Calculation

The Codecov API natively provides the patch coverage out of the box. However, the Coveralls API does not natively support this functionality. Therefore, an experimental repository was created to validate both the concept of patch coverage as per Marinescu *et al.* definition and also understand the workings of Line-by-Line coverage from the coverage service tools used.

B.1 Coveralls

The Coveralls File API (Person.py) returns a coverage array as shown below Listing 12. 0 indicates that the line was not covered, 2 indicated how many times line was executed and *null* is a line that is not relevant such as a comment or space.

Since `Coveralls` doesn't inherently provide the *patch coverage*, I used Equation (B.1) to calculated it. Using PyDriller's *Modified File* API, for each commit, I iterated through all changed files and call the Coveralls array API which retrieve these coverage arrays similar to the one above. PyDriller's parsed endpoint of the *Modified File* API provides exact details of the lines which were modified, I used that to Map to the coverage array. After which depending on the line number it will indicate whether it is an executable line, irrelevant line and whether it was executed.

$$Patch\ Coverage = \frac{Executed\ Lines\ in\ Changed\ Files}{Executable\ Lines\ in\ Changed\ Files} \tag{B.1}$$

Coveralls calculates the Overall coverage in a similar fashion as the patch coverage, except Overall coverage caters for both **Changed** and **UnChanged** files i.e **ALL** files as expressed by Equation (B.2) . The simpler and easier way to get the coverage

```
[
  2,
  2,
  2,
  0,
  null,
  2,
  2,
  0,
  null,
  2,
  2,
  0
]
```

Listing 12: Person.py file for Commit: 999294f5cd6233686c54a6fb9e19c45cea98db20

is by calling the builds endpoint of the coveralls API. However to confirm consistency (in the formula), calling the Commit SourceFiles API (which returns *ALL* files, irrespective of whether changed or not) will return Listing 13. Iterating through all these source adding up *"covered line count"* and *"relevant line count"* eventually taking the ratio of the two will return the commit overall coverage, for example, from the response in Listing 13:

$$Overall\ Coverage = \frac{Total\ Executed\ Lines\ in\ ALL\ Files}{Total\ Executable\ Lines\ in\ ALL\ Files} \tag{B.2}$$

$$Overall = \frac{21+18+7+24+13+15+8+27}{25+23+10+25+14+18+11+28} \times 100 = \frac{133}{154} = 86.3636\% \tag{B.3}$$

B.2 Codecov

Fortunately, Codecov inherently support the concept of patch coverage. Calling the Commit Report API. The easier way to get patch coverage would be to the `diff[5]` in the totals property from Listing 13. However, to validate Equation (B.1), I iterated through the files array in the Commit Report API and retrieved the line array for

```
{
    "total": 8,
    "total_pages": 1,
    "current_page": 1,
    "source_files": "[{
        "name": "algorithms.py",
        "relevant_line_count": 25,
        "covered_line_count": 21,
        "missed_line_count": 4,
        "covered_percent": 84.0
    },
    {
        "name": "identity_validator.py",
        "relevant_line_count": 23,
        "covered_line_count": 18,
        "missed_line_count": 5,
        "covered_percent": 78.261
    },
    {
        "name": "person.py",
        "relevant_line_count": 10,
        "covered_line_count": 7,
        "missed_line_count": 3,
        "covered_percent": 70.0
    },
    {
        "name": "polynomial.py",
        "relevant_line_count": 25,
        "covered_line_count": 24,
        "missed_line_count": 1,
        "covered_percent": 96.0
    },

    ... # excluded for brevity
]"
}
```


Listing 13: Commit SourceFiles API: 999294f5cd6233686c54a6fb9e19c45cea98db20

each file. The File line coverage array is made of array of two elements where the first elements is the line number and the second element states whether the line was executed, missed (executable line but not executed) or partially hit/executed. 0 indicates the line was executed, 1 indicates the line was missed and 2 is a line that was partially hit/executed. CodeCov doesn't include irrelevant lines in the response. Similar to Coveralls's SourceFiles API, this report API returns **ALL** files irrespective of whether they were changed or not. Thus, to filter out UnChanged files, I used PyDriller's Modified File API. In order to get the patch coverage, I applied the same equation (Equation (B.1)) but except that I also included partially hit lines. The result are consistent with Coveralls patch coverage and The results we see from the $diff$[5] property in "totals" in the API response and/or CodeCov UI.

B.3 Differences (Inconsistencies)

While the *Patch Coverage* is similar from Calculating it using Equation (B.1) for Coveralls and CodeCov (both calculating it again and comparing what we see from the UI/API). The `Overall Coverage` seems to differ slightly. As can bee seen from both APIs, Coveralls returns 86.363% whilst 81.25% is returned by CodeCov. Coveralls is consistent in their equation. The only notable difference (thus far) between the two tools is the concept of `partially hit` lines.

Appendix C

SIG Delta Maintainability (SIG-DMM)

This chapter presents an example of how to calculate the delta scores for unit size which will eventually form part of the composite of the delta maintainability index. This example follows through the technical report provided by Di Biase [153]. The example code can be found in DMM Calculation . Below are the statistics in table C.1. The example uses size but the exact same procedure is applicable to the other DMM properties.

Table C.1: Commit Statistics

Commit	Size (LoC)	
	search.py	sort.py
711abc6f37	27	22
fff1442594	17	16

Commit	Method (LoC)		Method (LoC)	
	linear	binary	bubble	insertion
711abc6f37	10	17	13	9
fff1442594	5	12	7	9

C.1 Risk Profile Unit Size

The first step in the calculation of the delta score using the SIG-DMM model involves categorising the units according to the risk profile. For context, a unit in this case is a method. To categorise into either high or low, the benchmark already derived by [63, 64] is used, which is 15 LoC. Both files in the DMM Calculation consist of two methods. Thus the methods will be categorised according to the benchmark.

From Table C.1, the first commit fff144, does not have a method with a higher LoC than 15, thus all methods fall under low risk. For search.py the low risk total lines are

Table C.2: Risk Profile Per File

Commit Hash	Filename	Low	High
711abc6f37	search.py	10	17
	sort.py	22	0
fff1442594	search.py	17	0
	sort.py	16	0

17 (methods combined is 12 + 5 = 17), for sort.py low risk is 16 (methods combined in 7 + 9). In both files, the high risk is zero. In the second commit 711abc6, For search.py the low risk total lines are 10 because only the *linear_search* method has less than 15, the binary method now falls into the high risk because it has 17 LoC in the second commit which is greater than 15. For sort.py none of the methods has 15 or more lines in the second commit, hence, the high risk profile is zero and the low risk profile is 22 (methods combined is 13 + 9). Table C.2 shows the categorised files per file in each commit.

C.2 Risk Profile Delta Unit Size

The delta risk profile per file is essentially the difference per file between successive commits. It is always in the form of $new_commit - old_commit$ per file for each risk profile. For example, the low risk profile for file search.py is given by -7 ($10 - 17$ = -7) as shown in Table C.3. The corresponding high risk profile is 17 ($17 - 0 = 17$) .

Table C.3: Risk Profile Delta Per File

Filename	Low	High
search.py	-7	17
sort.py	6	0

C.3 Risk Profile Delta Increases/Decreases Per File

Th next step is to take risk profile deltas from Appendix C.2 and organise them to decreases and increases base. Decreases are seen as a overall net negative delta value whilst increases are seen as net positive net deltas. From Table C.3 the only grouping the increase and decrease deltas leads to Table C.4 . It is important to note

that these are the absolute values and excludes the negative sign. The sign merely indicates whether it was a decreases or increase.

Table C.4: Risk Profile Delta Increases/Decreases Per File

Filename	Increase		Decrease	
	low	high	low	high
search.py	0	17	7	0
sort.py	6	0	0	0

C.4 Commit Risk Profile Delta Increases/Decreases Per File

The overall commit risk delta increases and decreases are the sum from Table C.4 for all files. The resulting data is Table C.5.

Table C.5: Commit Risk Profile Delta Increase/Decrease

Commit	Increase		Decrease	
	low	high	low	high
711abc6f375	6	17	7	0

C.5 Commit Risk Profile Delta Increases/Decreases Per File

$$\begin{aligned} LRPD &= CRPDI_{low} + CRPDD_{high} \\ &= 6 + 0 \\ &= 6 \end{aligned} \tag{C.1}$$

$$\begin{aligned} HRPD &= CRPDD_{low} + CRPDI_{high} \\ &= 17 + 7 \\ &= 23 \end{aligned} \tag{C.2}$$

$$\begin{aligned} Delta\ Score &= \frac{LRPD}{LRPD + HRPD} \\ &= \frac{6}{6 + 23} \\ &= 0.207 \end{aligned} \tag{C.3}$$

Appendix D

Virtualisation Configuration

This Appendix presents the virtualisation configuration, specifically, the containerisation script used to create and instantiate an environment used to compute and collect the patch metrics. The containerisation tool of choice was Docker due to its popularity. The scripting language was bash as these containers were running on top of an Ubuntu machine.

D.1 Dockerfile

Listing D.1: Dockerfile

```
FROM ubuntu:22.04
RUN apt-get update && apt-get install -y \
    sudo \
    build-essential \
    autoconf \
    automake \
    libtool \
    git \
    curl \
    openssh-server \
    --no-install-recommends
RUN sudo apt install software-properties-common -y
RUN sudo add-apt-repository ppa:ubuntu-toolchain-r/test
RUN sudo apt update -y && sudo apt install gcc-9 -y
RUN sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-9 100
RUN sudo update-alternatives --install /usr/bin/gcov gcov /usr/bin/gcov-9 100
RUN add-apt-repository -y ppa:deadsnakes/ppa
RUN apt update
RUN apt install -y python3.10
RUN mkdir -p /var/run/sshd
```

```
RUN mkdir /root/.ssh
ADD id_rsa.pub /root/.ssh/authorized_keys
RUN chown -R root:root /root/.ssh
RUN chmod -R 700 /root/.ssh
RUN apt-get install -yq --reinstall ca-certificates \
    software-properties-common
RUN apt-get install -y python3.10
RUN update-alternatives --install \
    /usr/bin/python3 python3 /usr/bin/python3.10 1 && \
    update-alternatives --set python3 /usr/bin/python3.10
RUN apt-get install -yq \
    libxml2-dev \
    build-essential \
    zlib1g-dev \
    libncurses5-dev \
    libgdbm-dev \
    libnss3-dev \
    libssl-dev \
    libreadline-dev \
    libffi-dev \
    libsqlite3-dev \
    libbz2-dev \
    wget
ENV DEBIAN_FRONTEND=noninteractive
ENV TZ=Africa/Harare
RUN apt-get update && \
    apt-get install -y software-properties-common && \
    ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && \
    echo $TZ > /etc/timezone
RUN apt-get install -yq --reinstall ca-certificates
RUN apt update -y && apt upgrade -y
RUN apt install -y vim vim-tiny
RUN echo 'alias python=/usr/bin/python3' >> /${USER}/.bashrc
RUN mkdir -p /home/${USER}/repositories
RUN git clone https://github.com/ErnestFistozz/CoverageImporterAsync.git \
/home/CoverageImporter
EXPOSE 22
CMD ["/usr/sbin/sshd", "-D"]
```

D.2 Instantiating a Container

Listing D.2: Instance of Data Collector

```
#!/usr/bin/bash
docker build --file /path/to/Dockerfile \
    --tag <image-name>:<version> \
```

```
    <build-context>
docker run -d -p <host-port>:22 \
    -v  /path/to/store/results:/home/root/repositories \
    --name <container-name>  <image-name>:<version>
ssh -p <host-port>  -i /path/to/private-key root@localhost
# inside the container
cd /home/root/CoverageImporter
python -m venv .venv # create a python environment
source .venv/bin/activate
pip install -r requirements.txt
nohup python <file-name> 2>&1 &  # run script in background
```

Appendix E

Web API calls response

There are two primary API endpoints from Coveralls used in this investigation, the "***files***" and "***line_ coverage***". The file endpoint depicted by Listing 14 returns a list of all covered source files with aggregated statistics of how the file was covered. On the other hand, the line coverage endpoint in Listing 15 returns a line by line coverage of specific file passed as a query parameter. Each line coverage is designated a between 0 to N describing how many times the line was executed (i.e called) during a test run. For example a zero would indicate the line was not executed during a test run while a number greater than zero would indicate how many times the line was called. A null on the other hand would mean the line is not an executable line, such as a comment or blank line.

Codecov's commit endpoint of the API returns the files and its corresponding line by line coverage in a single API call. Unlike Coveralls, Codecov uses 0 to indicate a covered lines, 1 as missed line and 2 as a partially covered line. A sample response is shown in Listing 16.

```
{
  "total": 4,
  "total_pages": 1,
  "current_page": 1,
  "source_files": [
            {
                "name":"identity_validator.py",
                "relevant_line_count":23,
                "covered_line_count":18,
                "missed_line_count":5,
                "covered_percent":"78.26"
            },
            {
                "name":"polynomial.py",
                "relevant_line_count":22,
                "covered_line_count":21,
                "missed_line_count":1,
                "covered_percent":"95.45"
            },
            ....
        ]
}
```

Listing 14: Coveralls - Covered Files Response @b398e58

```
[
  2,
  null,
  null,
  2,
  null,
  2,
  2,
  2,
  2,
  null,
  2,
  2,
  null,
  2,
  2,
  null,
  2,
  2,
  1,
  2,
  2,
  ...
]
```

Listing 15: Coveralls- File coverage Array @b398e58

```
{
    "totals": {
        "files": 4,
        "diff": [2,50,33,17,0,"66.00000",0,0,0,0,0,0,0],
        ...
    },
    "files": [
            {
                "name": "polynomial.py",
                "totals": {
                    "lines": 22,
                    ...
                },
                "line_coverage": [ [1,0], [4,0], [6,0], [7,0],
                ...
                ]
            },
            {
                "name": "test_polynomial.py",
                "totals": {
                    "lines": 28,
                    ...
                },
                "line_coverage": [ [1,0], [2,0], [5,0], [7,0],
                    ...
                ]
         },
            ...
    ]
}
```

Listing 16: Codecov - Commit report @b398e5894

Appendix F

Codecov Correlating Coefficients

The following tables presents the correlation coefficients between the various metrics. Spearman and Kendall coefficients per metric pair per coverage service tool. ρ and τ represents Spearman and Kendall respectively.

Table F.1: Spearman and Kendall's Correlation Coefficient

Symbol	Cov v/s Patch		Patch v/s MI		Patch v/s CRAP		Cov v/s MI		Cov v/s CRAP	
	ρ	τ	ρ	τ	ρ	τ	ρ	τ	ρ	τ
CV01	0.047	0.034	0.441	0.363	0.301	0.243	0.078	0.056	0.084	0.061
CV02	0.358	0.279	0.276	0.229	0.106	0.082	0.043	0.031	-0.012	-0.009
CV03	-0.024	-0.02	0.382	0.343	0.312	0.282	0.007	0.006	0.032	0.025
CV04	-0.084	-0.071	0	0	0	0	0	0	0	0
CV05	-0.158	-0.121	0.423	0.368	0.273	0.234	-0.074	-0.064	-0.053	-0.045
CV06	0.219	0.171	0.338	0.287	0.206	0.171	0.053	0.04	0.014	0.011
CV07	0.15	0.122	0.552	0.53	0.506	0.488	0.073	0.058	0.055	0.04
CV08	0.01	0.007	0.516	0.431	0.387	0.311	-0.123	-0.091	-0.129	-0.095
CV09	-0.047	-0.037	0.581	0.522	0.493	0.444	-0.02	-0.015	-0.03	-0.022
CV10	0.148	0.11	0.352	0.269	0.163	0.116	-0.009	-0.006	0.013	0.012
CV11	0.098	0.082	0.576	0.546	0.572	0.55	0.261	0.217	0.223	0.184
CV12	0.288	0.236	0.366	0.357	0.273	0.269	0.023	0.018	-0.002	-0.002
CV13	0.051	0.043	0.781	0.742	0.68	0.631	0.043	0.034	0.04	0.033
CV14	0.155	0.125	0.651	0.561	0.602	0.446	0.168	0.135	0.193	0.149
CV15	-0.016	-0.01	0.427	0.367	0.381	0.308	-0.12	-0.089	-0.135	-0.103
CV16	0.018	0.016	0.405	0.385	0.333	0.323	0.011	0.008	-0.016	-0.015
CV17	-0.109	-0.089	0.104	0.101	0.066	0.064	0.033	0.027	0.048	0.039
CV18	0	0	0	0	0	0	-0.058	-0.045	-0.039	-0.032
CV19	-0.04	-0.031	0.586	0.533	0.492	0.449	0.032	0.024	0.111	0.085
CV20	-0.226	-0.185	0.601	0.559	0.488	0.456	-0.222	-0.174	-0.194	-0.155
CV21	0.155	0.128	0.58	0.537	0.567	0.531	0.181	0.146	0.093	0.081
CV22	-0.09	-0.073	0.781	0.694	0.731	0.655	-0.078	-0.062	-0.064	-0.05
CV23	0.037	0.03	0.321	0.301	0.272	0.257	-0.024	-0.019	0.007	0.005

Table F.2: Spearman and Kendall's Correlation Coefficient

Symbol	Cov v/s Patch		Patch v/s MI		Patch v/s CRAP		Cov v/s MI		Cov v/s CRAP	
	ρ	τ	ρ	τ	ρ	τ	ρ	τ	ρ	τ
CV24	0.249	0.208	0.347	0.323	0.297	0.278	0.2	0.161	0.158	0.131
CV25	-0.067	-0.057	0.534	0.508	0.466	0.448	0.132	0.106	0.127	0.103
CV26	-0.086	-0.07	0.511	0.486	0.47	0.445	-0.119	-0.093	-0.147	-0.118
CV27	0.126	0.104	-0.026	-0.024	-0.043	-0.041	-0.009	-0.006	-0.001	0
CV28	0.101	0.084	0.205	0.188	0.143	0.13	0.091	0.069	0.088	0.065
CV29	-0.22	-0.166	0.789	0.684	0.688	0.616	-0.216	-0.164	-0.191	-0.151
CV30	0.098	0.076	0.291	0.225	0.081	0.043	0.019	0.017	-0.037	-0.028
CV31	-0.026	-0.021	0.302	0.276	0.184	0.167	0.065	0.049	0.054	0.042
CV32	-0.032	-0.023	0.726	0.613	0.656	0.58	-0.102	-0.076	-0.085	-0.068
CV33	-0.144	-0.126	0.169	0.159	0.135	0.128	-0.147	-0.128	-0.183	-0.159
CV34	0.023	0.018	0.34	0.288	0.177	0.152	-0.001	0	0.037	0.025
CV35	0.118	0.091	0.652	0.566	0.507	0.433	0.056	0.04	-0.025	-0.018
CV36	0	0	0.734	0.716	0.735	0.725	0	0	0	0
CV37	-0.26	-0.201	0.857	0.695	0.72	0.549	-0.386	-0.274	-0.366	-0.269
CV38	-0.127	-0.1	0.419	0.346	0.279	0.226	-0.007	-0.007	0.02	0.018
CV39	-0.073	-0.059	0.218	0.198	0.112	0.096	0.014	0.011	0.026	0.021
CV40	-0.06	-0.043	0.418	0.355	0.354	0.304	0.149	0.112	0.191	0.149
CV41	-0.001	-0.003	0.2	0.167	-0.127	-0.143	-0.023	-0.015	-0.057	-0.041
CV42	-0.39	-0.303	0.592	0.564	0.574	0.532	-0.177	-0.152	-0.177	-0.152
CV43	0.138	0.109	0.422	0.383	0.306	0.273	-0.079	-0.061	-0.083	-0.065
CV44	0.119	0.097	0.494	0.474	0.421	0.402	-0.162	-0.128	-0.169	-0.136
CV45	-0.011	-0.009	0.436	0.419	0.37	0.356	0.104	0.083	0.088	0.071
CV46	0.097	0.079	0	0	0	0	0	0	0	0
CV47	0.079	0.065	0.168	0.159	0.144	0.136	0.041	0.033	0.062	0.05
CV48	-0.142	-0.111	0.467	0.428	0.367	0.323	-0.135	-0.104	-0.138	-0.11
CV49	0.084	0.064	0.58	0.513	0.486	0.43	0.087	0.066	0.044	0.035
CV50	0.297	0.24	0.273	0.255	0.207	0.197	0.153	0.117	0.13	0.103
CV51	-0.095	-0.074	0.173	0.139	-0.02	-0.021	-0.162	-0.113	-0.059	-0.047
CV52	0.068	0.054	0.408	0.373	0.344	0.313	-0.117	-0.09	-0.11	-0.085
CV53	-0.032	-0.025	0.52	0.438	0.438	0.354	-0.077	-0.057	-0.082	-0.062
CV54	0.132	0.107	0.219	0.196	0.135	0.125	-0.011	-0.008	-0.029	-0.022
CV55	0.074	0.054	0.522	0.419	0.434	0.356	-0.005	-0.004	0.02	0.016
CV56	-0.034	-0.024	0.451	0.386	0.349	0.301	0.096	0.076	0.164	0.133
CV57	0.255	0.198	0.56	0.478	0.442	0.365	0.065	0.05	0.038	0.029
CV58	0.387	0.295	0.38	0.318	0.149	0.116	0.022	0.014	-0.083	-0.063
CV59	0.299	0.239	0.458	0.404	0.318	0.269	-0.097	-0.073	-0.126	-0.096
CV60	-0.172	-0.136	0.394	0.357	0.326	0.296	-0.013	-0.011	0.016	0.012
CV61	-0.17	-0.136	0.389	0.364	0.231	0.216	0.063	0.05	0.107	0.085
CV62	0.104	0.078	0.533	0.444	0.407	0.311	-0.047	-0.035	-0.07	-0.053

Table F.3: Spearman and Kendall's Correlation Coefficient

Symbol	Cov v/s Patch		Patch v/s MI		Patch v/s CRAP		Cov v/s MI		Cov v/s CRAP	
	ρ	τ	ρ	τ	ρ	τ	ρ	τ	ρ	τ
CV63	-0.117	-0.098	0.105	0.096	0.113	0.095	0.014	0.015	0.112	0.091
CV64	-0.119	-0.104	0.798	0.76	0	0	-0.116	-0.094	0	0
CV65	0.088	0.069	0.348	0.303	0.273	0.223	0.212	0.163	0.194	0.151
CV66	-0.115	-0.093	0.324	0.307	0.293	0.281	0.019	0.016	0.031	0.025
CV67	-0.025	-0.02	0.352	0.334	0.244	0.236	0.197	0.158	0.229	0.186
CV68	0	0	0	0	0	0	0.148	0.124	0.139	0.116
CV69	0.372	0.302	0.452	0.419	0.328	0.304	0.036	0.027	-0.016	-0.016
CV70	0.345	0.279	0.348	0.312	0.308	0.284	0.132	0.099	0.142	0.11
CV71	0.11	0.092	0.3	0.255	0.181	0.132	-0.016	-0.007	-0.042	-0.033
CV72	0.035	0.027	0.533	0.43	0.315	0.195	-0.115	-0.085	-0.21	-0.158
CV73	0.166	0.133	0.566	0.535	0.513	0.467	-0.044	-0.034	-0.069	-0.054
CV74	-0.025	-0.019	0.683	0.633	0.564	0.521	0.012	0.01	0.007	0.006
CV75	0.053	0.041	0.559	0.512	0.426	0.367	0.087	0.074	0.049	0.037
CV76	-0.013	-0.01	0.418	0.39	0.188	0.181	-0.096	-0.07	-0.102	-0.08
CV77	-0.192	-0.187	0.699	0.699	0.699	0.699	0.024	0.024	0.024	0.024
CV78	-0.188	-0.136	0.181	0.148	0.147	0.098	0.026	0.017	-0.111	-0.082
CV79	0.101	0.075	0.236	0.19	0.129	0.096	0.198	0.144	0.27	0.208
CV80	-0.11	-0.09	0.516	0.433	0.409	0.338	-0.196	-0.149	-0.2	-0.154
CV81	-0.014	-0.008	0.647	0.509	0.527	0.418	-0.145	-0.106	-0.137	-0.103
CV82	0.038	0.038	0	0	0	0	0	0	0	0
CV83	0.15	0.114	0.33	0.279	0.223	0.158	-0.015	-0.012	-0.079	-0.059
CV84	-0.1	-0.087	0.712	0.654	0.475	0.471	-0.289	-0.232	-0.359	-0.299
CV85	-0.097	-0.077	0.522	0.449	0.429	0.379	-0.033	-0.022	-0.033	-0.022
CV86	0.214	0.17	0.512	0.45	0.414	0.361	0.08	0.062	0.096	0.073
CV87	-0.199	-0.174	0.506	0.495	0.399	0.391	-0.19	-0.166	-0.163	-0.143
CV88	-0.147	-0.126	0.515	0.495	0.424	0.413	-0.132	-0.11	-0.103	-0.086
CV89	0	0	0.577	0.57	0.504	0.495	0	0	0	0
CV90	0.104	0.102	0.046	0.044	0.05	0.049	0.081	0.076	0.039	0.037
CV91	0.165	0.133	0.076	0.069	-0.025	-0.024	0.029	0.018	-0.005	-0.004
CV92	0.529	0.433	0.211	0.19	0.144	0.131	0.061	0.046	0.034	0.026

Table F.4: Coveralls: Spearman and Kendall's Correlation Coefficient

Symbol	Cov v/s Patch		Patch v/s MI		Patch v/s CRAP		Cov v/s MI		Cov v/s CRAP	
	ρ	τ	ρ	τ	ρ	τ	ρ	τ	ρ	τ
CL01	0	0	0	0	0	0	-0.21	-0.168	-0.21	-0.173
CL02	0	0	0	0	0	0	-0.006	-0.005	0.098	0.084
CL03	0.053	0.046	0.188	0.174	0.123	0.118	-0.023	-0.017	-0.06	-0.049
CL04	0.097	0.081	0.336	0.324	0.303	0.295	0.008	0.007	-0.028	-0.023
CL05	0	0	0	0	0	0	-0.023	-0.021	-0.045	-0.04
CL06	-0.06	-0.048	0.295	0.288	0.291	0.286	-0.077	-0.064	-0.113	-0.094
CL07	0.047	0.037	0.377	0.349	0.307	0.288	-0.002	-0.002	-0.016	-0.012
CL08	0.014	0.011	1	1	1	1	0.014	0.011	0.014	0.011
CL09	0.11	0.093	0.147	0.134	0.162	0.159	-0.106	-0.085	-0.129	-0.106
CL10	-0.203	-0.169	0.638	0.624	0.683	0.658	-0.242	-0.196	-0.312	-0.26
CL11	0.274	0.225	0.625	0.603	0.566	0.542	0.294	0.239	0.283	0.231
CL12	-0.134	-0.109	0.522	0.505	0.526	0.501	-0.232	-0.19	-0.219	-0.178
CL13	0.053	0.043	0.64	0.621	0.577	0.557	0.027	0.022	0.019	0.015
CL14	-0.063	-0.051	0.477	0.448	0.295	0.276	-0.042	-0.033	-0.004	-0.005
CL15	0.25	0.205	0.212	0.203	0.109	0.106	0.129	0.1	0.119	0.096
CL16	-0.388	-0.312	0.317	0.298	0.275	0.255	-0.022	-0.017	0.009	0.007
CL17	0.032	0.025	0.703	0.647	0.517	0.487	-0.05	-0.039	-0.124	-0.1
CL18	0.078	0.065	0.712	0.71	0.712	0.71	-0.047	-0.039	-0.047	-0.039
CL19	0	0	0	0	0	0	0	0	0	0
CL20	0.108	0.087	0.762	0.725	0.733	0.683	0.074	0.058	0.077	0.06
CL21	0	0	0	0	0	0	0.016	0.013	0.044	0.035
CL22	0.045	0.033	0.217	0.183	0.055	0.023	0.024	0.02	0.045	0.033
CL23	0.311	0.256	0.442	0.421	0.399	0.369	0.09	0.071	0.118	0.091
CL24	0.206	0.166	0.515	0.494	0.494	0.466	0.279	0.213	0.271	0.212
CL25	0.074	0.061	-0.168	-0.158	-0.169	-0.161	0.068	0.051	0.062	0.049
CL26	0.101	0.082	0.515	0.505	0.366	0.354	0.209	0.174	0.144	0.123
CL27	0	0	0	0	0	0	0	0	0	0
CL28	-0.132	-0.113	0.644	0.639	0.277	0.274	-0.141	-0.123	-0.111	-0.096
CL29	-0.02	-0.017	0.697	0.696	0.697	0.696	0.047	0.038	0.047	0.038
CL30	-0.06	-0.048	0.143	0.135	0.01	0.01	0.011	0.004	0.055	0.042
CL31	-0.032	-0.027	0.126	0.11	-0.014	-0.015	0.066	0.049	0.045	0.033
CL32	-0.193	-0.161	0.615	0.595	0.466	0.465	-0.37	-0.308	-0.446	-0.377
CL33	0.395	0.318	0.332	0.286	0.181	0.165	0.108	0.067	0.066	0.055
CL34	0.271	0.242	0.338	0.336	0.324	0.32	0.113	0.1	0.107	0.097
CL35	0	0	0	0	0	0	0	0	0	0

Table F.5: Coveralls Cont: Spearman and Kendall's Correlation Coefficient

Symbol	Cov v/s Patch		Patch v/s MI		Patch v/s CRAP		Cov v/s MI		Cov v/s CRAP	
	ρ	τ	ρ	τ	ρ	τ	ρ	τ	ρ	τ
CL36	0.132	0.104	0.444	0.413	0.387	0.355	0.141	0.111	0.1	0.079
CL37	-0.065	-0.052	0.585	0.571	0.453	0.443	-0.146	-0.12	-0.148	-0.12
CL38	0.231	0.184	0.338	0.267	0.153	0.102	0.051	0.03	0.018	0.009
CL39	0.176	0.145	1	1	1	1	0.176	0.145	0.176	0.145
CL40	0.001	0.001	0.509	0.46	0.46	0.414	0.13	0.102	0.096	0.076
CL41	0.17	0.144	0	0	0	0	0	0	0	0
CL42	-0.116	-0.096	-0.044	-0.044	-0.044	-0.044	0.079	0.065	0.079	0.065
CL43	0	0	0	0	0	0	-0.261	-0.227	-0.261	-0.227
CL44	0	0	1	1	0	0	0	0	0	0
CL45	0	0	0.496	0.496	0.496	0.496	0	0	0	0
CL46	0.14	0.114	0.714	0.706	0.643	0.636	0.121	0.098	0.098	0.08
CL47	0.062	0.051	0.772	0.772	-0.002	-0.002	0.055	0.045	0.041	0.034
CL48	0.268	0.215	0.676	0.627	0.623	0.57	0.345	0.275	0.333	0.261
CL49	-0.001	0	0.529	0.502	0.441	0.408	-0.008	-0.006	-0.019	-0.015
CL50	0.116	0.095	0.627	0.612	0.607	0.594	-0.006	-0.006	0.007	0.002
CL51	0.093	0.074	0.684	0.628	0.568	0.512	0.133	0.103	0.109	0.085
CL52	-0.053	-0.041	0.433	0.353	0.213	0.157	-0.041	-0.029	-0.106	-0.077
CL53	-0.319	-0.258	0.423	0.396	0.384	0.349	-0.377	-0.305	-0.336	-0.274
CL54	-0.208	-0.163	0.693	0.644	0.587	0.486	-0.155	-0.124	-0.105	-0.084
CL55	0.232	0.182	0.382	0.329	0.279	0.227	0.184	0.14	0.191	0.147
CL56	0.119	0.098	0.555	0.545	0.522	0.513	-0.074	-0.06	-0.062	-0.051
CL57	0.195	0.149	0.57	0.491	0.41	0.352	0.012	0.01	-0.037	-0.027
CL58	0.179	0.138	0.621	0.572	0.592	0.528	0.04	0.031	0.085	0.066
CL59	-0.156	-0.139	0.618	0.601	0.458	0.442	-0.388	-0.348	-0.259	-0.239
CL60	-0.066	-0.063	0.724	0.678	0.646	0.617	-0.241	-0.219	-0.251	-0.232
CL61	0	0	0	0	0	0	-0.417	-0.416	-0.417	-0.416
CL62	0	0	0	0	0	0	0	0	0	0
CL63	0	0	0	0	0	0	0.057	0.054	0.057	0.054
CL64	0	0	0	0	0	0	-0.235	-0.196	-0.174	-0.145
CL65	-0.093	-0.072	0.374	0.295	0.105	0.024	-0.042	-0.036	-0.045	-0.033
CL66	0.009	0.009	0.135	0.132	0.134	0.132	0.024	0.024	0.023	0.023
CL67	0	0	0	0	0	0	0	0	0	0
CL68	-0.005	-0.004	0.181	0.16	0.07	0.061	0.074	0.051	0.103	0.081
CL69	0	0	0	0	0	0	0	0	0	0
CL70	0.138	0.115	0.617	0.601	0.382	0.375	0.201	0.164	0.228	0.189

Table F.6: Coveralls Cont: Spearman and Kendall's Correlation Coefficient

Symbol	Cov v/s Patch		Patch v/s MI		Patch v/s CRAP		Cov v/s MI		Cov v/s CRAP	
	ρ	τ	ρ	τ	ρ	τ	ρ	τ	ρ	τ
CL71	-0.07	-0.057	0.413	0.379	0.375	0.346	-0.219	-0.169	-0.193	-0.152
CL72	0.112	0.087	0.545	0.491	0.444	0.4	0.1	0.076	0.093	0.072
CL73	-0.107	-0.103	0.093	0.091	0.055	0.054	-0.191	-0.178	-0.197	-0.183
CL74	0.134	0.108	0.698	0.657	0.583	0.553	0.13	0.104	0.137	0.111
CL75	-0.008	-0.006	0.599	0.563	0.549	0.52	0.047	0.036	0.016	0.012
CL76	0	0	0	0	0	0	0.035	0.03	0.021	0.018
CL77	0	0	0	0	0	0	0.075	0.062	0.075	0.062
CL78	0.07	0.056	0.425	0.399	0.389	0.35	-0.036	-0.029	-0.03	-0.024
CL79	0.233	0.212	0.58	0.555	0.298	0.294	0.155	0.13	0.235	0.211
CL80	-0.301	-0.25	0.156	0.151	0.094	0.091	-0.047	-0.04	-0.034	-0.028
CL81	0	0	0	0	0	0	0.082	0.067	0.074	0.061
CL82	-0.157	-0.131	0.669	0.649	0.619	0.603	-0.031	-0.025	-0.03	-0.026
CL83	-0.058	-0.049	0.679	0.617	0.527	0.473	-0.047	-0.04	-0.079	-0.062
CL84	0	0	0	0	0	0	0	0	0	0
CL85	-0.062	-0.051	0.331	0.302	0.229	0.215	0.19	0.146	0.15	0.118
CL86	-0.179	-0.149	0.219	0.209	0.222	0.209	-0.243	-0.201	-0.244	-0.203
CL87	0	0	0	0	0	0	-0.006	-0.006	-0.006	-0.006
CL88	0.009	0.006	0	0	0	0	0	0	0	0
CL89	-0.301	-0.296	-0.025	-0.025	-0.025	-0.025	-0.257	-0.25	-0.262	-0.257
CL90	0	0	-0.012	-0.012	-0.007	-0.007	0	0	0	0
CL91	-0.074	-0.061	0.648	0.629	0.59	0.57	0.047	0.039	0.032	0.026
CL92	0.062	0.051	0.776	0.75	0.634	0.612	0.016	0.014	0.037	0.03
CL93	0	0	0	0	0	0	-0.136	-0.105	-0.169	-0.137
CL94	0	0	0	0	0	0	0.043	0.038	0.051	0.041
CL95	0	0	0	0	0	0	-0.072	-0.063	-0.057	-0.05
CL96	0	0	0	0	0	0	-0.011	-0.01	0.003	0.003

Appendix G

Project Description

Projects existing independently on Coveralls and Codecov and projects existing in both coverage service tools were mined. For analysis purposes, each project/repository is assigned an identifier based on the coverage service, CV for Codecov and CL for Coveralls respectively. Projects existing in both Coveralls and Codecov are denoted by CM. The time period, in months, between the first commit and last commit also computed to ensure at least six months period the first and last commit. The starting and ending coverage and lines of code are measured for the first and last commits respectively.

Table G.1: Codecov Project Key Statistics

Identifier	Project Name	language	Time(Months)	# Builds		LoC		Coverage(%)	
				Total	Mined	Start	End	Start	End
CV01	alibaba/fastjson2	Java	20	3378	2732	40205	85360	67.16	75.36
CV02	alibaba/GraphScope	C++/Rust	36	3758	976	4336	11343	77.26	40.91
CV03	alibaba/higress	Go	13	1281	273	4349	11199	29.87	33.91
CV04	alibaba/lowcode-engine	TypeScript	14	87	33	5497	8952	86.12	85.29
CV05	ansible/ansible-navigator	Python	23	2200	215	6796	10065	55.18	64.27
CV06	ansible/ansibullbot	Python	56	1309	443	9248	6992	33.45	36.5
CV07	ansible/receptor	Go	3	356	40	8530	7399	30.4	33.41
CV08	apache/apisix-ingress-controller	Go	36	3616	627	466	7966	58.15	37.37
CV09	apache/celix	C/C++	43	1957	1276	24995	24429	68.35	88.37
CV10	apache/yunikorn-core	Go	44	1716	368	5947	13436	58.06	77.77
CV11	apereo/phpCAS	PHP	41	134	71	2559	2539	43.77	43.44
CV12	apollographql/apollo-client	TypeScript	32	10098	3014	1981	3696	84.15	95.29
CV13	apollographql/federation	TypeScript	43	1134	354	2390	16895	90.54	68.48
CV14	ARMmbed/continuous-delivery-scripts	Python	33	288	109	1850	2551	68.81	68.91
CV15	arviz-devs/preliz	Python	19	502	145	927	4097	53.72	80.13
CV16	cfpb/cfpb-chart-builder	JavaScript	23	136	121	389	423	40.87	39.95
CV17	cfpb/hmda-platform	Scala	41	3157	2598	5088	11928	83.57	86.8
CV18	cfpb/hmda-platform-ui	Javascript	7	163	138	2123	2225	85.35	91.55
CV19	codecov/codecov-cli	Python	20	1265	438	242	2718	71.49	95.47
CV20	codecov/gazebo	Javascript	37	9075	5082	12	9669	16.67	97.91
CV21	codecov/opentelem-laravel	PHP	22	61	59	215	219	23.72	39.73
CV22	codecov/worker	Python	5	5386	163	25183	29868	98.61	98.21
CV23	containerd/nydus-snapshotter	Go	23	2009	750	1270	8287	34.57	33.52
CV24	containerd/ttrpc-rust	Rust	15	117	34	2413	2526	26.03	24.39
CV25	damianszczepanik/cucumber-reporting	Java	80	715	252	897	1212	98.66	97.11

Table G.2: Codecov Project Key Statistics Cont

Identifier	Project Name	Language	Time(Months)	# Builds		LoC		Coverage(%)	
				Total	Mined	Start	End	Start	End
CV26	didi/chameleon	Javascript	17	482	335	5976	15430	73.43	55.35
CV27	didi/cube-ui	Javascript	28	1027	623	937	3108	88.15	92.31
CV28	didi/mand-mobile	TypeScript	25	469	280	3401	2794	78.07	84.54
CV29	dotenv-linter/dotenv-linter	Rust	48	1747	351	590	2834	81.02	99.15
CV30	eBay/HomeObject	C++	4	313	93	315	1185	90.79	76.29
CV31	eclipse/kapua	Java	72	7477	2622	26362	41691	52.88	20.64
CV32	eclipse/kiso-testing	Python	22	982	211	4340	6487	77.81	98.12
CV33	eclipse/kuksa.val	Rust/C++	20	130	103	3489	5996	99.48	75.63
CV34	eclipse/mosquitto	C	24	138	117	30942	31283	81.15	80.86
CV35	elastic/apm-agent-python	Python	6	1184	104	10615	12528	88.72	94.88
CV36	Esri/solution.js	Typescript	5	184	28	7826	7861	100	100
CV37	facebook/akd	Rust	10	221	50	13740	9054	58.16	87.99
CV38	facebook/dns	Go	14	137	70	7420	8641	61.25	59.04
CV39	facebook/metro	Javascript	72	2377	1026	3475	10582	81.32	83.25
CV40	facebook/time	Go	44	788	77	1252	12103	26.84	57.97
CV41	google/go-github	Go	57	3007	976	11996	13241	33.97	97.72
CV42	intel/ad-rss-lib	C++	53	200	44	2329	34794	90.47	82.04
CV43	JabRef/jabref	Java	72	9889	2841	53993	63046	29.68	41.47
CV44	jenkinsci/analysis-model	Java	72	2798	2271	2297	6603	88.16	93.09
CV45	jenkinsci/warnings-ng-plugin	Java	62	3501	2791	3692	5358	83.07	81.82
CV46	JuliaLang/JuliaSyntax.jl	Julia	10	336	151	3776	4159	92.9	96.56
CV47	jupyter/nbdime	TypeScript	71	672	395	4637	10410	50.49	64.64
CV48	jupyter/nbgrader	Python	44	494	372	4610	6949	90.2	76.95
CV49	jupyter/papyri	Python	24	648	492	3841	5723	73.18	62.31
CV50	kubernetes/ingress-nginx	Go	38	4300	1766	4838	6589	36.61	56.06
CV51	linkedin/avro-util	Java	40	955	225	1659	28259	53.53	46.17
CV52	microsoft/autogen	Python	3	2678	743	1764	4085	36.68	34.71
CV53	microsoft/onnxscript	Python	15	3417	933	4753	15631	79.63	78.77
CV54	microsoft/responsible-ai-toolbox	TypeScript	24	4384	1196	3177	7731	89.42	88.84
CV55	microsoft/superbenchmark	Python	35	2557	512	3	6912	100	85.79
CV56	moby/swarmkit	Go	63	197	86	24611	31139	62.05	61.68
CV57	Netflix/conductor	Java	31	1635	892	11113	19293	58.03	65.78
CV58	Netflix/genie	Java	66	898	267	6587	14976	84.67	90.75
CV59	Netflix/Priam	Java	30	236	147	6599	7270	36.73	47.83
CV60	Netflix/titus-executor	Go	36	1879	919	7165	22852	35.42	17.49
CV61	nukeop/nuclear	TypeScript	26	763	331	5072	6706	61.28	71.41
CV62	nushell/nushell	Rust	3	1335	293	98492	102281	55.08	68.55

Table G.3: Codecov Project Key Statistics Cont

Identifier	Project Name	Language	Time(Months)	# Builds		LoC		Coverage(%)	
				Total	Mined	Start	End	Start	End
CV63	nushell/reedline	Rust	3	153	29	7805	7911	49.8	49.26
CV64	openstreetmap/mod_tile	C++	5	129	32	4565	4601	44.93	49.16
CV65	openSUSE/osc	Python	56	575	263	16422	17758	22.99	31.05
CV66	php-censor/php-censor	PHP	71	1378	930	12652	10214	16.9	23.68
CV67	phpmyadmin/sql-parser	PHP	80	1052	755	3998	5024	100	96.44
CV68	puppetlabs/puppetlabs-stdlib	Ruby	15	242	120	5238	5220	5.46	3.01
CV69	pytest-dev/pytest-django	Python	78	1031	354	2007	2300	96.26	96.48
CV70	python-babel/babel	Python	71	670	237	4040	4444	90.05	91
CV71	pytorch/torcheval	Python	14	366	136	6020	10699	94.24	95.35
CV72	redis/jedis	Java	36	1741	547	9891	14965	58.83	75.44
CV73	scrapy/scrapyd	Python	75	276	157	833	1883	66.03	87.2
CV74	servo/rust-url	Rust	77	245	144	3534	3547	81.24	81.7
CV75	square/FetchRequests	Swift	40	186	83	5112	5049	93.45	93.29
CV76	square/Valet	Swift	41	247	96	1173	986	72.8	82.56
CV77	square/web-sdk	TypeScript	29	221	43	58	58	82.76	56.9
CV78	tensorflow/adanet	Python	18	210	145	1682	4032	93.88	85.27
CV79	tikv/agatedb	Rust	9	160	32	6228	8838	83.86	89.6
CV80	tikv/migration	Go	20	580	68	19838	20230	53.46	61.74
CV81	tikv/raft-engine	Rust	25	760	136	5508	12457	83.59	98.21
CV82	TNG/property-loader	Java	29	205	131	406	402	86.95	87.81
CV83	twitter/finagle	Scala	52	2098	437	23237	27030	66.41	75.32
CV84	vercel/edge-runtime	TypeScript	5	729	21	1698	2434	90.4	82.58
CV85	videojs/http-streaming	Javascript	33	976	186	8905	10731	86.1	86.07
CV86	vuejs/vue-eslint-parser	TypeScript	80	403	226	490	3397	95.71	89.9
CV87	vuejs/vuefire	TypeScript	80	1612	839	99	3850	98.99	74.65
CV88	yelp/swagger-gradle-codegen	Kotlin	23	179	121	873	576	27.72	70.83
CV89	yelp/swagger-spec-compatibility	Python	46	81	26	1724	2128	100	100
CV90	yelp/venv-update	Python	34	82	60	1902	1871	97.95	100
CV91	zulip/python-zulip-api	Python	76	1420	320	7805	8967	83.36	84.18
CV92	zxing/zxing	Java	81	392	209	12807	15648	73.92	79.1

Table G.4: Coveralls Project Key Statistics

Identifier	Project Name	Language	Time(Months)	# Builds		LoC		Coverage(%)	
				Total	Mined	Start	End	Start	End
CL01	aerogear/aerogear-js-sdk	TypeScript	9	656	164	878	576	62.13	64.946
CL02	aerogear/keycloak-connect-graphql	TypeScript	22	557	119	221	168	68.671	99.342
CL03	aerogear/mobile-developer-console	JavaScript	5	548	241	691	1454	68.452	60.523
CL04	aerogear/mobile-security-service-operator	Go	6	696	172	578	2494	78.028	62.791
CL05	aerogear/voyager-server	TypeScript	21	628	116	312	183	49.673	63.194
CL06	apache/commons-collections	Java	71	610	296	13011	13474	85.097	90.137
CL07	apache/commons-math	Java	81	468	270	42689	33502	90.103	90.562
CL08	apache/datasketches-cpp	Java	34	530	180	5190	16193	93.854	98.938
CL09	apache/incubator-hivemall	Java	14	865	100	19579	28636	34.838	42.104
CL10	apache/libcloud	Python	15	275	63	55916	7018	89.878	22.67
CL11	apache/servicecomb-pack	Java	72	2224	1315	1313	4957	95.278	81.824
CL12	ARMmbed/mbed-ls	Python	34	496	282	685	1058	38.686	82.136
CL13	chef/supermarket	Ruby	12	3156	1159	441	2564	97.506	98.674
CL14	containers/virtcontainers	Go	14	1488	519	2967	6590	62.588	61.32
CL15	coreos/alb-ingress-controller	Go	14	322	202	1949	2864	0.77	29.504
CL16	data-8/datascience	Python	95	752	460	1099	2468	74.431	94.935
CL17	docker/distribution	Go	2	707	272	6020	7295	54.585	56.532
CL18	docker/libkv	Go	39	329	74	848	1384	61.91	84.032
CL19	docker/swarm	Go	2	627	132	2429	2640	58.172	48.902
CL20	dropwizard/dropwizard	Java	62	3720	1521	4597	8528	55.21	85.741
CL21	eBay/ebayui-core	TypeScript	36	3833	1096	416	1886	86.781	83.728
CL22	eclipse/orion.client	JavaScript	8	273	178	6377	37900	50.924	34.862
CL23	elastic/elasticsearch-py	Python	33	161	126	6327	70392	38.865	57.16
CL24	elastic/logstash	Java	11	513	403	4059	5053	70.978	72.155
CL25	elixir-lang/ex_doc	Elixir	28	793	231	643	879	95.956	95.108
CL26	Esri/openssl	C	1	117	72	210670	183584	66.405	65.008
CL27	F5Networks/f5-adcaas-openstack	Python	29	1787	279	104	2975	100	94.328
CL28	F5Networks/f5-openstack-agent	Python	88	799	772	4697	9944	28.848	23.441
CL29	facebook/AsyncDisplayKit	Objective-C++	3	741	110	8082	2361	57.473	56.756
CL30	facebook/ds2	C++	17	1098	209	7735	10041	42.986	51.1
CL31	facebook/react	Javascript	44	5822	4752	3803	20696	87.115	86.162
CL32	gavv/roc-go	Go	11	111	26	249	806	76.305	95.037
CL33	google/OpenSK	Rust	29	843	70	6056	13747	41.955	96.319
CL34	gpodder/mygpo-auth	Python	59	483	140	282	730	100	94.521
CL35	grpc/grpc-node	TypeScript	36	1586	514	2449	3297	85.336	73.65
CL36	HazyResearch/fonduer	Python	24	1506	508	5847	4440	44.844	85.788
CL37	IBM/pytorchpipe	Python	7	276	89	3476	5916	38.176	36.832
CL38	ilovepi/Compiler	C#	29	221	193	271	970	53.137	96.495
CL39	intel/govmm	Go	31	337	87	1140	2150	70.526	80.698
CL40	intel/multus-cni	Go	31	779	274	780	1616	45	70.73
CL41	JuliaLang/HDF5.jl	Julia	20	291	76	1269	1088	90.859	72.702
CL42	jupyter/qtconsole	Python	63	581	222	4972	4690	21.319	61.834
CL43	kiegroup/kogito-serverless-operator	Go	14	968	37	182	2747	64.286	32.399

Table G.5: Coveralls Project Key Statistics Cont

Identifier	Project Name	Language	Time(Months)	# Builds		LoC		Coverage(%)	
				Total	Mined	Start	End	Start	End
CL44	Kinto/kinto-elasticsearch	Python	46	681	120	72	266	100	100
CL45	Kinto/kinto-facebook	Python	39	644	196	155	155	100	100
CL46	kubernetes/helm	Go	26	1271	1258	2697	12868	54.579	63.747
CL47	kubernetes/ingress	Go	10	2093	731	2655	8234	33.258	43.745
CL48	linkedin/gobblin	Java	15	698	244	22458	1420	42.181	56.901
CL49	linuxfoundation/cii-best-practices-badge	Ruby	7	815	677	415	781	73.494	96.543
CL50	ManageIQ/ui-components	TypeScript	64	1341	633	819	2201	89.744	50.977
CL51	microsoft/botbuilder-python	Python	53	2956	746	4808	13512	74.106	66.985
CL52	Microsoft/sqltoolsservice	C#	42	2316	754	7729	31129	20.158	76.822
CL53	Microsoft/vscode-mssql	TypeScript	28	1302	921	3567	3671	59.322	65.759
CL54	MITLibraries/topichub	Scala	14	370	188	3856	6022	18.62	65.294
CL55	mitmproxy/netlib	Python	26	484	309	1076	2691	99.164	96.284
CL56	mozilla/zamboni	Python	3	795	335	52428	56707	96.214	96.223
CL57	Netflix/genie	Java	80	2685	1020	3610	14940	89.723	93.876
CL58	Netflix/repokid	Python	44	1808	171	747	2061	28.782	56.235
CL59	npm/wombat-cli	Javascript	13	541	158	27	252	100	77.778
CL60	nskins/goby	Ruby	36	177	113	1798	2659	89.711	100
CL61	ntedgi/express-body-parser-error-handler	Javascript	24	551	411	12	10	71.429	100
CL62	openSUSE/snapper	C++	11	229	159	8393	8317	36.995	38.511
CL63	opsdroid/opsdroid-audio	Python	49	794	257	297	327	10.774	14.067
CL64	osmlab/osm-request	Javascript	32	134	105	2	252	100	67.342
CL65	pingcap/tidb-binlog	Go	6	375	72	10595	11511	67.702	75.484
CL66	platinumazure/eslint-plugin-qunit	Javascript	100	3111	462	147	878	100	100
CL67	puppetlabs/cpp-hocon	C++	25	251	68	6346	6724	63.379	87.909
CL68	reactjs/react-modal	Javascript	53	704	300	270	380	76.296	84.737
CL69	reactjs/reselect	Typescript	29	372	182	41	15	100	100
CL70	rust-lang/crates.io	Rust	9	375	223	3022	3738	85.87	82.343
CL71	Semantic-Org/Semantic-UI	Javascript	18	786	624	3971	1639	14.002	89.262
CL72	sirixdb/sirix	Java	9	736	571	26791	30610	58.8	64.028
CL73	spatie/laravel-enum	PHP	8	121	67	27	1041	100	100
CL74	square/ghostunnel	Go	52	802	436	857	1479	90.548	89.655
CL75	square/go-jose	Go	70	825	397	1668	3251	97.002	90.065
CL76	sveltejs/svelte-eslint-parser	TypeScript	9	472	135	2193	2633	90.693	91.454
CL77	sympy/sympy	Python	51	703	30	2876	451	98.748	88.248
CL78	tensorflow/lucid	Python	37	499	249	431434	2951	25.846	63.063
CL79	terasolunaorg/terasoluna-gfw	Java	1	59	23	2610	2610	99.004	99.004

Table G.6: Coveralls Project Key Statistics Cont

Identifier	Project Name	Language	Time(Months)	# Builds		LoC		Coverage(%)	
				Total	Mined	Start	End	Start	End
CL80	tessel/t2-cli	Javascript	39	531	173	3376	3113	73.993	85.898
CL81	tinyhttp/tinyhttp	TypeScript	29	259	148	1341	3282	92.84	98.062
CL82	twitter/secureheaders	Ruby	42	606	354	300	2042	100	96.572
CL83	twitter/twitter-cldr-rb	Ruby	53	481	67	9572	12142	64.574	96.014
CL84	videojs/conventional-changelog-videojs	Javascript	38	115	80	76	76	94.737	94.737
CL85	vuejs/vue-validator	Javascript	17	348	290	108	1427	100	94.184
CL86	webdriverio/webdrivercss	Javascript	22	190	117	147	217	89.796	99.078
CL87	wemake-services/dotenv-linter	Python	15	522	292	455	456	98.12	98.123
CL88	wemake-services/kira	Elixir	23	386	56	124	165	88.71	90.909
CL89	wemake-services/remark-lint-are-links-valid	Javascript	30	286	70	88	62	90.909	100
CL90	wemake-services/stylelint-config-strict-scss	Javascript	11	365	214	1	4	100	100
CL91	yahoo/fluxible	Javascript	77	980	490	232	1529	90.086	95.749
CL92	yahoo/react-i13n	Javascript	61	863	202	547	673	86.106	90.342
CL93	Zooz/api-schema-builder	Javascript	12	236	86	348	555	96.629	97.133
CL94	Zooz/express-ajv-swagger-validation	Javascript	26	341	129	184	71	99.329	99.213
CL95	Zooz/prometheus-api-metrics	Javascript	22	564	469	38	68	87.273	99.209
CL96	zulip/zulipbot	Javascript	38	235	62	225	356	0.444	59.27

Appendix H

Additional Figures

The following results are patch coverage results of builds modifying source code files and only none-source code files.

H.1 Patch Coverage - Code and Non-Code Builds

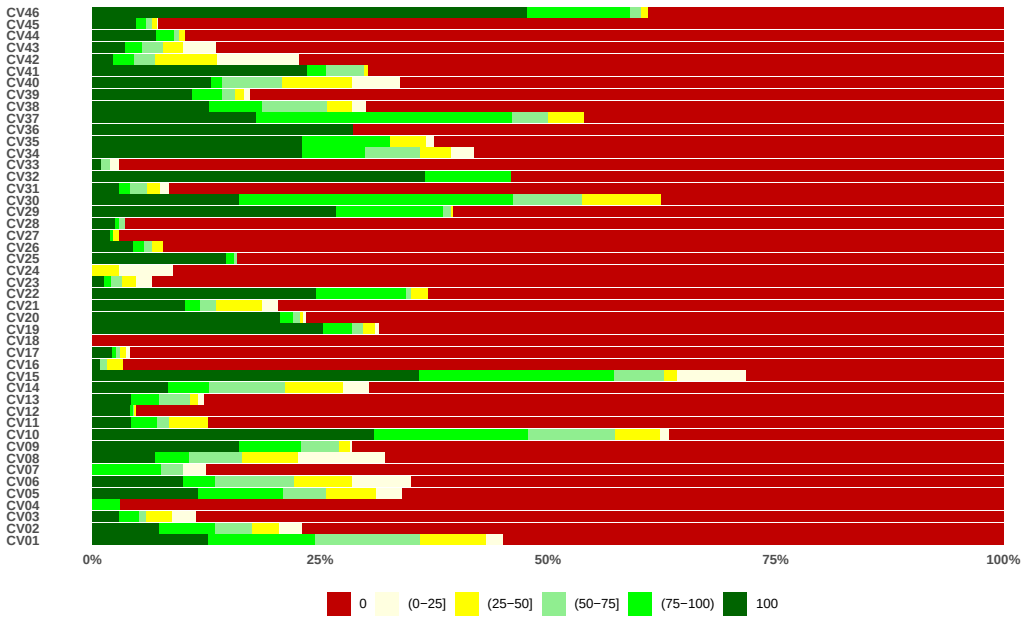


Figure H.1: Codecov Patch Coverage distribution: The bar chart illustrates the distribution of Patch Coverage in patches affecting Code and Non-Code files, with each bar representing the percentage of patches falling within a specific coverage range.

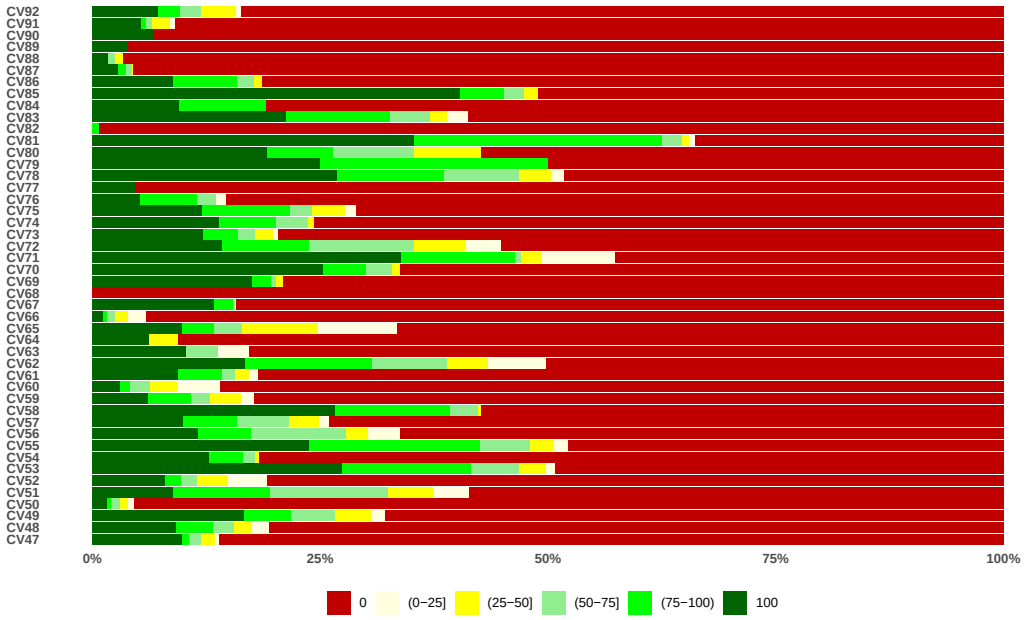


Figure H.2: Codecov Patch Coverage distribution Cont: The bar chart illustrates the distribution of Patch Coverage in patches affecting Code and Non-Code files, with each bar representing the percentage of patches falling within a specific coverage range.

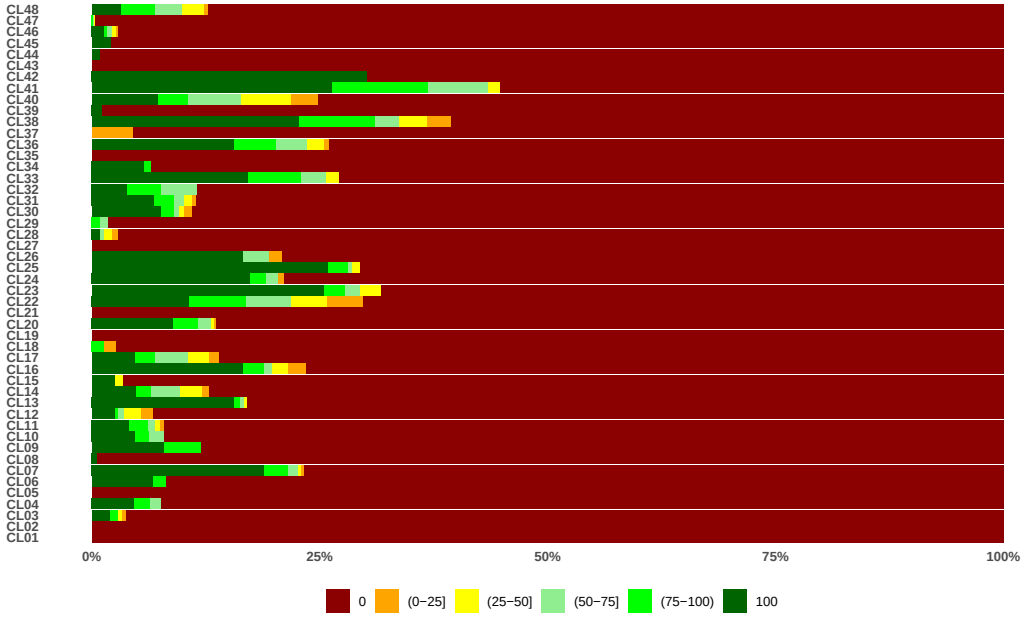


Figure H.3: Coveralls Patch Coverage distribution: The bar chart illustrates the distribution of Patch Coverage in patches affecting Code and Non-Code files, with each bar representing the percentage of patches falling within a specific coverage range.

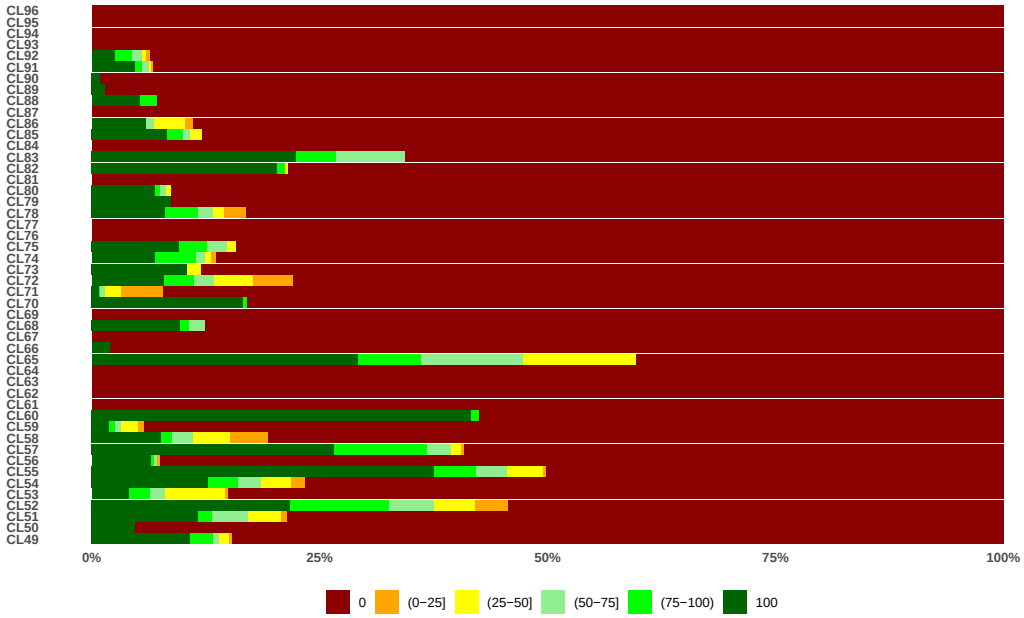


Figure H.4: Coveralls Patch Coverage distribution Cont: The bar chart illustrates the distribution of Patch Coverage in patches affecting Code and Non-Code files, with each bar representing the percentage of patches falling within a specific coverage range.