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

Ernest Bonginkosi Mamba and Stephen Phillip Levitt(✉)

University of the Witwatersrand, Johannesburg, South Africa
stephen.levitt@wits.ac.za

Abstract. Contemporary software development often involves the use of source control repositories which are hosted online and making incremental patches (commits) throughout the development process. Online hosting services facilitate the use of build pipelines and the integration of code coverage services into these pipelines. However, existing research into how incremental patches to software systems affect code coverage has not fully taken advantage of the data that is made available by these coverage services. This paper presents a partial replication of two previous studies on patch coverage, analysing over 50,000 builds from 46 projects obtained from two popular coverage services, Codecov and Coveralls. Data quality issues, such as missing commits, duplicate builds from Cron Jobs, and sudden coverage drops, were identified and addressed, highlighting the need for rigorous data cleaning process when mining data from coverage services. Results indicate that patches are generally either fully covered or entirely uncovered, with a majority achieving full coverage, suggesting that very seldom do engineers opt for partial coverage. There is a weak correlation (correlation coefficient: 0.23) between patch coverage and system coverage, indicating that patch coverage alone cannot be used to predict system coverage evolution. Furthermore, patch testing does not enhance patch maintainability.

Keywords: Patch Coverage · Maintainability · Software System Evolution

1 Introduction

Meir Lehman, a prominent figure in the field of software engineering argued 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 [13,14]. 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.

A. Gerber (Ed.): SAICSIT 2024, CCIS 2159, pp. 79–94, 2024.
https://doi.org/10.1007/978-3-031-64881-6_5

Central to the concept of software evolution is the source code, the centre 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 [23], play a pivotal role in ensuring the continued proper functioning of a system. One measure of determining how thorough a test suite is, is known as *code coverage*. Code coverage refers to the number of source code lines that are executed when the test suite is run.

There is a delicate relationship between the source code and its tests because as the software evolves even minor code changes and refactoring efforts can disrupt existing tests [20,21] and significantly alter code coverage [6]. This underscores the need for the continued maintenance of tests as the source code, itself, evolves.

Software typically evolves through incremental changes or modifications to source code repositories by means of commits. In this paper, these modifications are termed "patches". A modification refers to the alteration, deletion, or addition of one or more lines within one or more files. A patch can modify source code files (production or test) or non-source files such as a README file. Patch testing specifically evaluates the testing of modified code, focusing on the extent to which the altered code is tested and covered. For the remainder of the study, the terms patch and commit are used interchangeably.

Previous research efforts have extensively investigated the co-evolution of test and source code, however, this has been done through mining multiple, stable *release* versions of systems [24]. While such an approach provides information at stable points in a system's development, it does not provide insights into the development process at day-to-day level. On the other hand, studies using more fine-grained empirical data have predominantly aimed to investigate the synchronous or sequential alteration of production code and test code [15,17,25,26]. Limited work has been done to understand how test coverage is affected by incremental changes. To date, only two studies from Marinescu *et al.* [16] and Hilton *et al.* [9] have investigated how incremental changes effect test coverage.

As hosting providers have become more sophisticated over time in terms of the services they offer, new opportunities have arisen to study evolution of source code and accompanying tests. GitHub [7], for example, now allows development teams to create flexible build pipelines which can be used to take the source code in a repository through a number of different stages, including unit testing, and ultimately deploy it into production. GitHub also affords tight integration with third-party code coverage services. These services enable the development team to graphically visualise and track their coverage statistics. GitHub's built-in build pipeline functionality, together with the public APIs offered by the third-party coverage services that it integrates with, has both increased the number of open-source projects which generate coverage statistics and made these statistics accessible.

In this paper, the work of Marinescu *et al.* [16] and Hilton *et al.* [9] is extended by considering a different dataset, and specifically making use of code coverage service data, to investigate the maintainability of incremental changes (patches)

and the relationship between patch testing and its impact on incremental change maintainability.

2 Related Work

Marinescu *et al.* [16] pioneered the exploration of patch coverage in small incremental changes and subsequently established a formal definition for this concept. To investigate the co-evolution of test and production code, and patch coverage, the authors developed the COVRIG tool and conducted a study involving six open-source software (OSS) projects written in C/C++. The authors selected 250 revisions per project, iteratively checking out each revision, compiling and collecting coverage information from coverage reports. Their findings revealed that patches were either fully covered or not covered at all, with engineers seldom opting for partial patch coverage. 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. Hilton *et al.* [9] 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. Hilton *et al.* employed a mixed method to collecting coverage information by using coverage service tool Coveralls [18] along with manually compiling code and collecting coverage information. They chose 47 projects spanning different programming languages and utilised 250 revisions for projects hosted on Coveralls. Hilton *et al.* present slightly contradicting results from Marinescu *et al.*, whereby they state that patch coverage is not bimodal, rather, it varies from patch to patch with no-discernible pattern. 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.

Zaidman *et al.* [25,26] studied whether production and test code co-evolve synchronously at a commit level using two Java projects. The authors observed two patterns of evolution, synchronous - where production and test code are changed together and phased - where production and test code are changed separately. Stanislav *et al.* [15] examined sixty-one projects with a total of 240000 commits to examine the co-evolution of test and production maintenance. Their results showed that, in the majority of cases, production code changes, in particular, code fixes happens solely without modifying test code. Marsavina *et al.* [17], mined five open-source projects and used association rules to identify co-evolution patterns. Their results showed six distinct co-evolution patterns.

3 Research Questions

The study conducted here is a partial replication of the studies conducted by [9,16], and as such the following research questions are the same:

1. What is the distribution of patch sizes? This inquiry aims to assess the total number of lines affected per patch (i.e. magnitude of each patch), potentially unveiling insights into the incremental changes that occur as the system and tests evolve.
2. What is the distribution of patch coverage across the revisions of the system? Analysing patch testing activities could provide insights into testing practices within an open-source environment and help understand why the system coverage is as it appears.
3. How does individual patch coverage affect overall system coverage? The reasonable hypothesis would be that higher patch coverage implies an improved system coverage and vice versa, therefore, this question aims at validating this hypothesis.

Additionally, the following new research question is posed:

4. How maintainable is a typical patch, and how does patch maintainability vary across revisions? Maintainability is measured using the Software Improvement Group's (SIG) maintainability model [12]. This question is investigated because along with patch testing, it is important to understand how maintainability varies at the level of incremental changes.

Aside from the addition of a novel research question, this study adds value by making use of an almost entirely different dataset to the studies that it replicates. Lastly, and importantly, the methodology used here is different in that this work exclusively makes use of commercial coverage services which offer free coverage reporting for open-source projects. Using existing coverage services greatly simplifies the calculation of coverage statistics, enabling a greater breadth of projects to be covered or a more in-depth analysis of individual projects to be conducted, by considering a greater number of project builds.

4 Mining Open-Source Project Builds

Patch and system code coverage are determined by compiling and executing a project's test suite along with the production code being tested, and recording which production code statements have been hit or missed. Downloading and compiling projects can pose significant challenges due to project dependency mismatches and resource requirements, sometimes leading to compile failures for different revisions [25]. Marinescu *et al.* [16] attempted to address this using virtualisation, but acknowledged its continued resource intensity. Hilton *et al.* [9] adopted an approach in which they manually ran the test suites for some projects but used the Coveralls code coverage service for others.

4.1 Dataset Selection

All the projects that were selected for this study are hosted on GitHub [7]. GitHub serves as a central data hub for a huge number of open-source projects

due to its openness and licensing nature in contrast to proprietary source control management systems [19].

To identify possible projects, the sampling strategy employed by Pinto *et al.* [22] was adopted. In order for a repository to be considered it needed to be:

1. popular - using number of stars, forks, and contributors as measure of popularity,
2. actively maintained, and
3. stewarded by a well-respected open-source organisation, such as the Apache Foundation, or private company, such as Microsoft.

Repositories meeting the above criteria were then manually inspected to identify README files containing coverage badges from either CodeCov (eg. 

) or Coveralls (eg. 

). These badges indicate that the project in question was submitting data to the respective coverage service (with the badge giving the project's overall code coverage). Projects with diverse range of coverage, and meeting the additional criteria of having a minimum of one hundred builds over a two year period were finally selected. This last criteria ensured that each of the projects chosen has sufficient historical data for longitudinal analysis to be conducted.

It is noted that none of the projects from Marinescu *et al.* study utilised a coverage service; hence, they were excluded. Five projects appearing in the Hilton *et al.* study were selected; however, a larger number of revisions were available for analysis in this investigation.

4.2 Data Cleaning

Data cleaning is essential when there are data issues such as inconsistencies, gaps and misrepresented data. The following issues arose when analysing the data received from the coverage services' APIs:

Missing Commits - Various branching strategies can be adopted when using Git and GitHub, including Gitflow, GiHub flow, and so on. To ensure consistency, data is extracted from all branches. The rationale for mining all branches is that development on all branches will ultimately contribute and affect overall coverage. Additionally, the aim of this investigation is to focus on the incremental contributions towards building a system irrespective of which branches these contributions happen on. These contributions, if incorporated into the trunk (main branch), accurately portray how the system is being built.

However, 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. A dangling Git 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. To address the issue of dangling commits, all commits resulting in a missing commit hash error are skipped.

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

CronJobs/Cron-Builds - These define regular builds on specific branches and dates. CronJobs, which typically build the last commit on a specific branch, are not triggered by either a pull request or a push, and therefore contribute no new coverage and patch statistics because they are merely scheduled jobs. To exclude Cronjobs, builds emanating from unique commit hashes are sought within the build history, and duplicate builds are removed. Occasionally, it was observed that a slight variation in coverage occurs for builds resulting from the exact same commit hash. This is attributed to the dynamic nature of test coverage extraction, as noted by the Coveralls team [3]. Only the first build produced was analysed in situations where duplicate builds emanated from the same commit hash.

Sudden Coverage Drops - The most common data problem observed was either build failures or multi-triggered builds. These issues often lead to the coverage services' API failing to retrieve data from a build. This results in coverage spiking and/or incorrectly reported coverage. In an exchange with the Codecov team [10], 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 1 shows two consecutive builds, the first one returns correct data while the second returns 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 drop in coverage is chosen to filter out the noise in the data. An exponentially-weighted moving average (EWMA) is used to smooth the data from such outliers. Take the example of the Apache Gobblin project in Fig. 1 (top), there are numerous points where the coverage is picked-up as zero. Upon further investigation, it was discovered that these commits contained minimal changes, indicating that they could not have been the sole cause of the significant decrease in coverage from 45% to 0%.

A 30-point EWMA is applied selectively to maintain genuine coverage points, and is calculated only when a coverage value has deviated by a 30% or more from the previous value. The noise reduction that is achieved by adopting this approach can be seen in the bottom graph of Fig. 1.

4.3 Metrics and Measures Computation

Patch Coverage: Marinescu *et al.* [16] define 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

```
[    {"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,
      },
     {"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,
      },
     ... # excluded for brevity
]
```

Listing 1: Apache/Libcloud Corrupt Web API Response

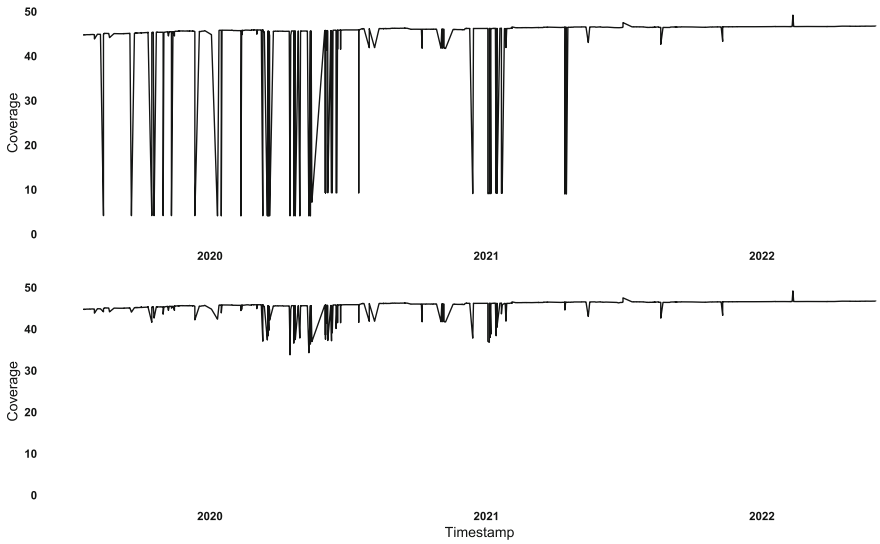

Fig. 1. Apache/Gobblin before data cleaning (top graph) and after data cleaning (bottom graph) with an EWMA

that adds the patch". In other words, unlike system coverage, patch coverage is a far more granular measure focused on newly added code changes [1,11]. 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 not necessarily be invoked. This definition excludes statements like comments as these are ignored by both interpreters and compilers.

To compute patch coverage for a given commit, the modified files, and the corresponding modified lines, are first identified. Then for each modified line, the line's coverage status is extracted from the code coverage service. The line's coverage status defines whether the line was executed or not. Coveralls represents line coverage status of executable lines as zero (line was not executed during the test run) or $N \in \{1, \ldots, \infty\}$, where N represents how many times the line was invoked during the test run. Codecov represents the line coverage status with a zero or one. Zero denotes lines that are not executed while one represents lines that are run by the tests. Codecov also has the concept of half-covered lines, termed *partial* coverage. Partially covered lines are branches with one or more omitted execution paths, and these are represented with a line coverage status of two. In this context, partially covered lines are not counted as executed (covered) lines and are added to the count of executable lines.

Patch Maintainability: The concept of fine grained maintainability was first explored by Kuipers *et al.* [12] and Heitlager *et al.* [8]. They introduced a maintainability model known as the SIG Maintainability Model (SIG-MM). SIG-MM was developed through the mapping of ISO/IEC-25010 quality characteristics to code properties and ultimately to source code metrics.

To tailor the SIG-MM for incremental changes, Di Biase *et al.* [4] refined this model to focus on patches and called this derivation the Delta Maintainability Model (DMM). Equation (1) presents the mathematical expression to calculate the DMM score of a patch and Fig. 2 depicts the mapping of the ISO/IEC-25010 quality characteristics to the code properties that are used in Eq. (1).

$$\text{DMM Score} = \frac{\text{DMM Size} + \text{DMM Unit Interface} + \text{DMM Complexity}}{\text{No. of Properties}} \quad (1)$$

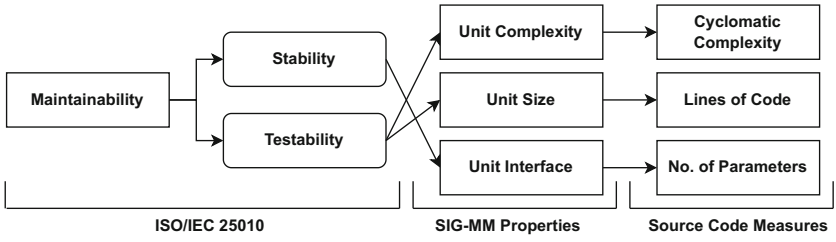

Fig. 2. Relation between SIG Maintainability Model and source code measures and ISO/IEC 25010 quality characteristics

4.4 Compute Infrastructure

Collecting and computing the above metrics required multiple RESTful API calls to the two coverage services. In addition to the API calls, the computation of the maintainability metric and the patch coverage required cloning each project repository in order to iterate over the commit history. Performing all these steps on a local machine can easily become cumbersome and inefficient due to the processing power required. Thus, a virtual machine and containers were utilised to collect and compute all the metrics. A virtual machine hosted on the Microsoft Azure cloud running Ubuntu-20.04 served as the host operating system. The Docker containerisation engine [5] was chosen for its popularity and efficiency. To ensure data collection from the coverage services was conducted in parallel, two containers were deployed with one dedicated to collecting and computing metrics using Coveralls while the other used Codecov.

5 Analyses of Patches and Patch Impacts

5.1 Project Statistics

The study data consists of projects from both Codecov (Table 1) and Coveralls (Table 2). In total 50,666 revisions across 46 projects in 10 programming languages were analysed. Tables 1 and 2 present the projects, along with the time in months denoting the difference between the first and last revision (build) dates, as well as the number of revisions during this period as well as the system coverage as measured for the first and last revisions.

5.2 Typical Patch Size

Hilton *et al.* [9] argues that smaller patches are easier for engineers to understand, while larger patches may necessitate external strategies for comprehension due to potential complexity. To investigate the size of patches, patches touching non-source files only are excluded from the analysis. This exclusion of non-source files patches led to the exclusion of projects CL{09,18} as their entire build history is based on non-source file patches. The distribution of patch sizes across revisions illustrated by the whisker box in Fig. 3 indicates that the typical size of patch is around 10 lines. It can also be observed that the upper quartile of the distribution indicates that most of the patches contain fewer than 1000 lines of code, suggesting that commits rarely exceed this threshold. These findings align with the work of Hilton *et al.* but differ slightly from Marinescu *et al.*, who reported a lower number of lines changed, ranging from 4 to 7.

A number of outliers are observed in this distribution, especially project CV10, which has a patch that has close to 100000 lines of code. Upon further examination this patch was revealed to be a merge patch that added just over 62000 lines of code from 3000 files. Further examination of the other outliers revealed that most of the commits of over 1000 LoC are actually merge commits.

Table 1. Project Key Statistics - Codecov

Identifier	Project Name	Language	Time (Months)	Revisions	Coverage (%)	
					Start	End
CV01	codecov/gazebo	Javascript/TypeScript	37	5094	16.67	97.91
CV02	apollographql/apollo-client	TypeScript	61	3034	84.15	95.29
CV03	apollographql/federation	TypeScript	43	354	90.54	68.48
CV04	TNG/property-loader	Java	29	131	86.95	87.81
CV05	alibaba/GraphScope	C++	36	967	77.26	40.91
CV06	jenkinsci/analysis-model	Java	72	2276	88.16	93.09
CV07	jenkinsci/warnings-ng-plugin	Java	62	2796	83.07	81.82
CV08	JabRef/jabref	Java	72	2844	29.68	41.47
CV09	eclipse/kapua	Java	72	2653	52.88	20.64
CV10	eclipse/mosquitto	C	24	117	81.15	80.86
CV11	ARMmbed/continuous-delivery-scripts	Python	33	476	68.81	68.91
CV12	apereo/phpCAS	PHP	41	213	43.77	43.44
CV13	Netflix/Priam	Java	30	441	36.73	47.83
CV14	zxing/zxing	Java	81	209	73.92	79.08
CV15	damianszczepanik/cucumber-reporting	Java	80	252	98.66	97.11
CV16	Netflix/genie	Java	92	1287	89.723	90.750
CV17	Netflix/conductor	Java	31	900	58.03	65.78
CV18	apache/yunikorn-core	Go	44	369	58.06	77.77
CV19	Netflix/titus-executor	Go	36	919	35.42	17.49
CV20	apache/celix	C	43	1276	68.35	88.37
CV21	facebook/metro	Javascript	72	1552	81.32	83.13
CV22	microsoft/superbenchmark	Python	35	513	0.00	85.79
CV23	google/go-github	Go	57	967	33.97	97.72
CV24	apache/apisix-ingress-controller	Go	36	627	58.15	37.24
CV25	kubernetes/ingress-nginx	Go	38	1772	36.61	56.02
CV26	microsoft/responsible-ai-toolbox	TypeScript	24	1196	89.42	89.01

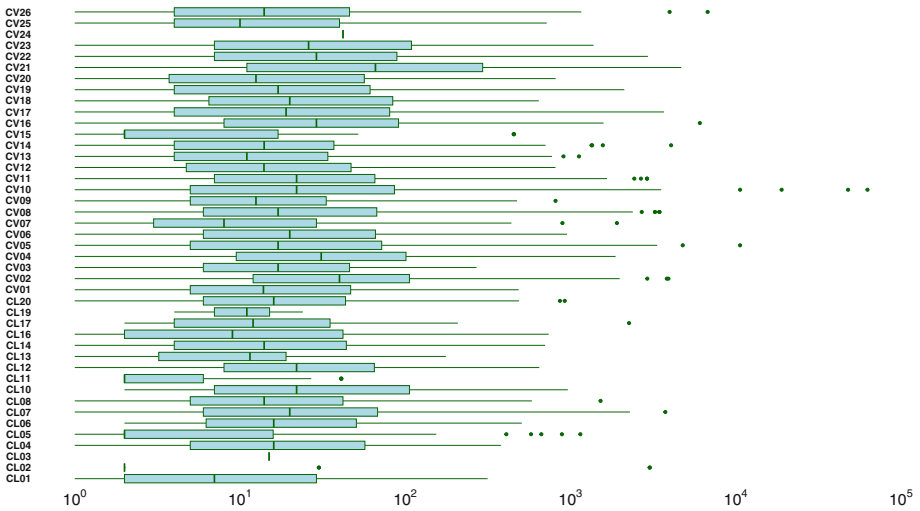

Fig. 3. Distribution of patch size: whisker/box plot representing the minimum, median and maximum patch size of patches modifying source code. The (log scale) x-axis represents the size of a patch in lines of code.

Table 2. Project Key Statistics - Coveralls

Identifier	Project Name	Language	Time (Months)	Revisions	Coverage (%)	
					Start	End
CL01	kubernetes/helm	Go	26	1258	54.579	63.541
CL02	ManageIQ/ui-components	Typescript	64	633	89.744	50.977
CL03	apache/datasketches-cpp	C++	34	180	93.854	98.938
CL04	apache/servicecomb-pack	Java	72	1315	95.278	81.703
CL05	Netflix/repokid	Python	44	171	28.782	56.235
CL06	yahoo/fluxible	Javascript	77	490	90.086	95.749
CL07	facebook/react	Javascript	44	4753	87.115	86.162
CL08	microsoft/botbuilder-python	Python	53	748	74.106	66.985
CL09	F5Networks/f5-adcaas-openstack	Python	29	279	100	94.328
CL10	Microsoft/vscode-mssql	Typescript	28	921	59.322	66.367
CL11	F5Networks/f5-openstack-agent	Python	88	772	28.848	23.441
CL12	Microsoft/sqltoolsservice	C#	42	754	20.158	76.855
CL13	yahoo/react-i13n	Javascript	61	202	86.106	90.342
CL14	HazyResearch/fonduer	Python	24	508	44.844	85.788
CL15	grpc/grpc-node	Typescript	36	514	85.336	73.630
CL16	square/ghostunnel	Go	52	436	90.548	89.655
CL17	square/go-jose	Go	70	397	97.002	90.065
CL18	eBay/ebayui-core	Typescript	36	1096	86.781	83.728
CL19	platinumazure/eslint-plugin-qunit	Javascript	100	465	100	100
CL20	dropwizard/dropwizard	Java	62	1521	55.210	85.738

5.3 Patch Coverage Analysis

To compute the coverage of patches, the definition provided by Marinescu *et al.* [16], and described earlier, is used. Patch coverage is computed 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 [9]. The 0% and 100% bins are specifically chosen to identify patches that are either completely uncovered or fully covered. A stacked bar graph is plotted to visualise the distribution for each project (Fig. 4).

Figure 4 shows that while partial coverage of patches is prevalent in almost all projects, patches are generally either entirely covered or uncovered, that is, the largest bins are the 0% or 100% bins in the majority of projects. This finding corroborates the observations of Marinescu *et al.* [16] and those of Hilton *et al.* [9].

5.4 Impact of Patches on System Coverage

A patch can either be fully, partially, or not covered. However, the consequence to the overall system coverage is not known a *priori*. To assess this impact, a positive influence is defined as an increase in system coverage, negative as a decrease, and no change (no impact) as neutral. Patches are categorised based on whether they modify source or non-source files. In answering this question, the coverage difference between two successive builds is examined. If the net coverage

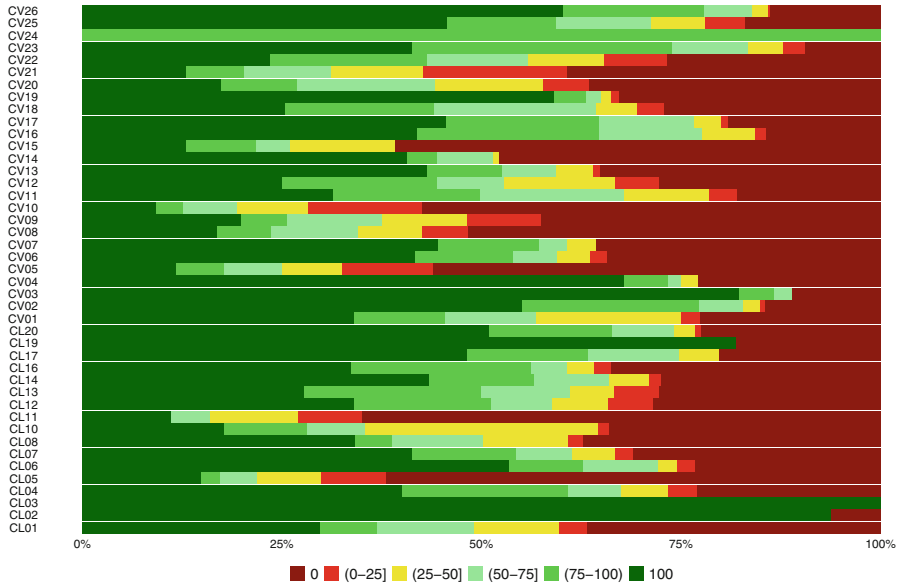

Fig. 4. Distribution of patch coverage for patches touching source files. Each colour represents the range and the size of the bar is the percentage of the patches whose coverage lies within the range.

from b1 (where b1 is the earlier build) to b2 (where b2 is the later build) is less than zero, the patch is said to have decreased (or negatively impacted) coverage, if the difference is greater than zero, the patch is said to have increased (or positively impacted) coverage, and if the difference is zero, the patch is classified as having no impact or change to overall coverage.

The majority of patches depicted in Fig. 5, particularly those altering non-source files, show a significant positive and negative impact. These findings align with Hilton *et al.*'s research [9], who asserted that patches involving non-source files can influence patch coverage. However, it is noted that such magnitudes could also be influenced by the absence of a one-to-one correspondence between commits and builds. Intermediate commits preceding a build-triggering commit may not be fully considered, potentially leading to fluctuations in coverage that are not accurately attributed. Conversely, projects CV{15,16,20,23,24} and CL19 demonstrate behavior deemed as "plausible", wherein patches modifying non-source files exhibit less influence on system coverage. Note in both modification types is the prevalence of patches with no impact/change, which aligns with the distribution of patch coverage. Interestingly, projects CL{09,18}, having build histories in which only non-source code files are modified, have some patches which increase and decrease system coverage. It is suspected that this phenomenon can be attributed to intermediate commits that did not initiate a build, or to the modification of build scripts, such as `Makefile`'s, which could cause coverage changes. To further assess the relationship between patch cov-

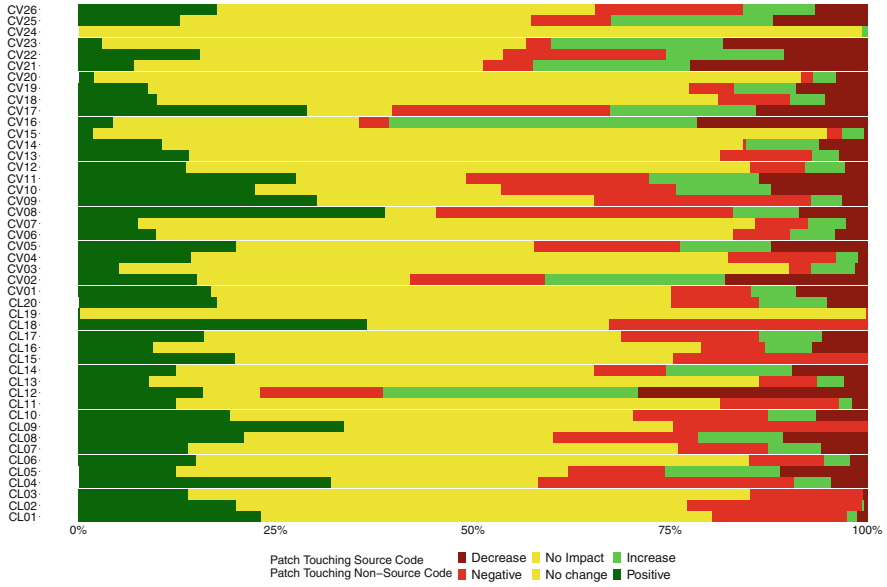

Fig. 5. Distribution of patch impact. Colour represents the range and size represents the percentage of patches whose impact lies within the range.

erage and overall coverage Kendall's coefficient is computed and Cohen's interpretation of correlation strength is employed, revealing a correlation strength of 0.239, classified by Cohen as weak [2].

5.5 Patch DMM Metric

According to Di Biase *et al.* [4], the DMM explores the changes to source code as the "ratio of good changes over the sum of good and bad changes". The resulting value of this ratio is, therefore, between 0 and 1, where zero indicates a "bad" (unmaintainable) change and one indicates "good" (maintainable) change.

Figure 6 illustrates the distribution of DMM score per patch. The DMM bins are split as 0, (0–0.25], (0.25–0.50], (0.50–0.75], (0.75–1.00) and 1.00. As in the case of patch coverage, the bins, 0 and 1.00 are specifically chosen to separate the scores of patches with DMM scores at the extremes, indicating a poor patch and good patch (from a maintainability perspective), respectively. As the DMM score is specifically designed for source code, any patches modifying non-source files will inherently have a DMM score of zero, and therefore, these patches are excluded. Note that the majority of patches have DMM scores either in the (0.50–0.75] bin or 0, which means patches are either "somewhat" maintainable or not maintainable at all. Kendall's coefficient is also computed to assess the influence of testing practices on maintainability. The relationship between patch coverage and DMM is found to be very weak, with a τ value of 0.160.

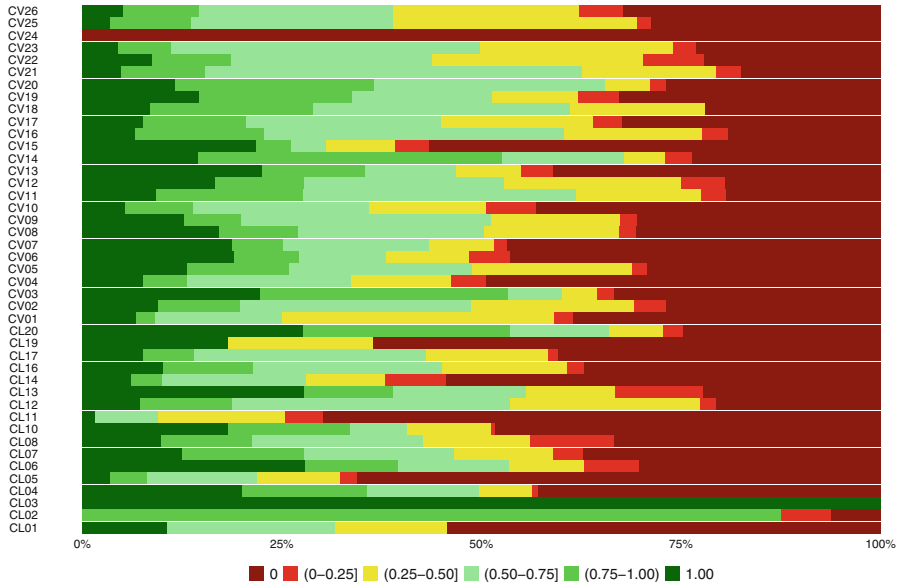

Fig. 6. Distribution of DMM: Each colour represent the range and the size of the bar is the percentage of the patches whose coverage lies within the range

5.6 Threats to Validity

In typical developer workflows, changes to a code repository involve adding/modifying files, staging changes, and pushing to the upstream repository. Developers tend to make multiple commits before pushing upstream. Coverage services are often triggered by pushes or merges to the upstream repository, leading to a lack of one-to-one mapping between commits and coverage reports. This lack of a one-to-one mapping between a commit and coverage service report necessitated using an approach that started from the coverage service end, and gathered all the commits hashes that triggered each build. This means there is no direct one-to-one mapping a between a commit and patch coverage per se. Thus, the patch coverage investigated is patch coverage in the context of triggered builds.

The reliance on coverage service data to analyse patch coverage is a critical factor. The discussion of the Apache/Gobblin data illustrates the potential pitfalls when blindly trusting coverage service data without implementing a rigorous data integrity verification process and employing robust data processing and cleaning methods. To mitigate this, a 30-point EWMA was utilised to smooth the data. Despite the EWMA's effectiveness in filtering out noise, it also introduces a limitation by potentially skewing genuine coverage decreases.

6 Conclusion

A longitudinal study examining testing and maintainability dynamics for small incremental changes is presented. By mining over 50,000 build histories from

two popular coverage services, Codecov and Coveralls, it is found that patches are generally either fully covered or entirely uncovered, with rare instances of partial coverage. Patch coverage shows a weak correlation with system coverage, indicating that this metric alone cannot be used to infer the trajectory of system coverage. This, therefore, implies for the development team that patch metrics must be used in conjunction with system metrics for a more thorough understanding of whether the system coverage is improving or not. Patch coverage is also found to have a weak correlation with patch maintainability, suggesting that patch testing may not significantly enhance maintainability. This study contributes to research on fine-grained changes to software systems by replicating two existing studies on an expanded dataset with new projects and by analysing these projects far more comprehensively through processing many more builds. Additionally, the value of leveraging coverage service data for research purposes is emphasised, albeit with a need for data scrutiny and the introduction of data cleaning processes.

References

1. Ben, S.: Patch coverage. https://seriousben.com/posts/2022-02-patch-coverage/. Accessed Feb 2022
2. Cohen, J.: Statistical Power Analysis for the Behavioral Sciences. Lawrence Erlbaum Associates (LEA), 2nd edn. (1988)
3. Coveralls: API returning same commit hash yet different coverage percentage and date (2023). https://github.com/lemurheavy/coveralls-public/issues/1648. (Personal Communication)
4. Di Biase, M., Rastogi, A., Bruntink, M., van Deursen, A.: The delta maintainability model: Measuring maintainability of fine-grained code changes. In: 2019 IEEE/ACM International Conference on Technical Debt (TechDebt), pp. 113–122 (2019). https://doi.org/10.1109/TechDebt.2019.00030
5. Docker: Make better, secure software from the start. https://www.docker.com/. Accessed Aug 2023
6. Elbaum, S., Gable, D., Rothermel, G.: The impact of software evolution on code coverage information. In: Proceedings IEEE International Conference on Software Maintenance. ICSM 2001, pp. 170–179 (2001). https://doi.org/10.1109/ICSM.2001.972727
7. GitHub: Let's build from here: The world's leading AI-powered developer platform. https://github.com/. Accessed Nov 2023
8. Heitlager, I., Kuipers, T., Visser, J.: A practical model for measuring maintainability. In: 6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007), pp. 30–39 (2007). https://doi.org/10.1109/QUATIC.2007.8
9. Hilton, M., Bell, J., Marinov, D.: A large-scale study of test coverage evolution. In: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 53–63 (2018). https://doi.org/10.1145/3238147.3238183
10. Hu, T.: Github issue: Patch coverage formula vs overall coverage ratio. https://github.com/codecov/feedback/issues/55. Accessed Aug 2023
11. Hu, T.: Why patch coverage is more important than project coverage. https://about.codecov.io/blog/why-patch-coverage-is-more-important-than-project-coverage/. Accessed Jan 2024

12. Kuipers, T., Visser, J.: Maintainability index revisited: position paper. In: Special Session on System Quality and Maintainability (SQM 2007) of the 11th European conference on software maintenance and reengineering (CSMR 2007) (2007)
13. Lehman, M.M., Belady, L.A.: Program evolution: processes of software change. Academic Press Professional, USA (1985)
14. Lehman, M.: Programs, life cycles, and laws of software evolution. Proc. IEEE **68**(9), 1060–1076 (1980). https://doi.org/10.1109/PROC.1980.11805
15. Levin, S., Yehudai, A.: 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), pp. 35–46 (2017). https://doi.org/10.1109/ICSME.2017.9
16. Marinescu, P., Hosek, P., Cadar, C.: 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, pp. 93—104. ACM (2014). https://doi.org/10.1145/2610384.2610419
17. Marsavina, C., Romano, D., Zaidman, A.: Studying fine-grained co-evolution patterns of production and test code. In: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, pp. 195–204 (2014). https://doi.org/10.1109/SCAM.2014.28
18. Merwin, N., Donahoe, L.: Coveralls: deliver better code. https://coveralls.io/. Accessed Nov 2023
19. Midha, V., Palvia, P.: Factors affecting the success of open source software. J. Syst. Softw. **85**(4), 895–905 (2012). https://doi.org/10.1016/j.jss.2011.11.010
20. Moonen, L., van Deursen, A., Zaidman, A., Bruntink, M.: On the interplay between software testing and evolution and its effect on program comprehension. In: Software Evolution, pp. 173–202. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-76440-3_8
21. Nierstrasz, O., Demeyer, S.: Object-oriented reengineering patterns. In: Proceedings of the 26th International Conference on Software Engineering, pp. 734—735. ICSE 2004, IEEE Computer Society, USA (2004)
22. Pinto, L.S., Sinha, S., Orso, A.: Understanding myths and realities of test-suite evolution. In: SIGSOFT FSE (2012). https://api.semanticscholar.org/CorpusID:9072512
23. Runeson, P.: A survey of unit testing practices. IEEE Softw. **23**(4), 22–29 (2006). https://doi.org/10.1109/MS.2006.91
24. Yu, L., Mishra, A.: An empirical study of Lehman's law on software quality evolution. Int. J. Softw. Inform. **7**, 469–481 (2013)
25. Zaidman, A., Rompaey, B., Deursen, A., Demeyer, S.: Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. Empir. Softw. Eng. **16**(3), 325–364 (2011)
26. Zaidman, A., Van Rompaey, B., Demeyer, S., van Deursen, A.: Mining software repositories to study co-evolution of production and test code. In: 2008 1st International Conference on Software Testing, Verification, and Validation, pp. 220–229 (2008). https://doi.org/10.1109/ICST.2008.47