Productive Coverage: Improving the Actionability of Code Coverage

Marko Ivanković
markoi@google.com
Google Switzerland GmbH
Zurich, Switzerland

Goran Petrović
goranpetrovic@google.com
Google Switzerland GmbH
Zurich, Switzerland

Yana Kulizhskaya
ykulizhskaya@google.com
Google Switzerland GmbH
Zurich, Switzerland

Mateusz Lewko
mlewko@google.com
Google Switzerland GmbH
Zurich, Switzerland

Luka Kalinovčić*
kalinovcic@gmail.com

René Just
rjust@cs.washington.edu
University of Washington
Seattle, WA, USA

Gordon Fraser
gordon.fraser@uni-passau.de
University of Passau
Passau, Germany

ABSTRACT

Code coverage is an intuitive and widely-used test adequacy measure. Established coverage measures treat each test goal (e.g., statement or branch) as equally important, and code-coverage adequacy requires every test goal to be covered. However, this is in contrast to how code coverage is used in practice. As a result, simply visualizing uncovered code is not actionable, and developers have to manually reason which uncovered code is critical to cover with tests and which code can be left untested. To make code coverage more actionable and further improve coverage in our codebase, we developed Productive Coverage — a novel approach to code coverage that guides developers to uncovered code that should be tested by (unit) tests. Specifically, Productive Coverage identifies uncovered code that is similar to existing tested and/or frequently in production executed code. We implemented and evaluated Productive Coverage for four programming languages (C++, Java, Go, and Python), and our evaluation shows: (1) The developer sentiment, measured at the point of use, is strongly positive; (2) Productive Coverage meaningfully increases test quality, compared to a strong baseline; (3) Productive Coverage has no negative effect on code authoring efficiency; (4) Productive Coverage modestly improves code-review efficiency; (5) Productive Coverage improves code quality and prevents defects from being introduced into the code.

ACM Reference Format:
Marko Ivanković, Goran Petrović, Yana Kulizhskaya, Mateusz Lewko, Luka Kalinovčić, René Just, and Gordon Fraser. 2024. Productive Coverage: Improving the Actionability of Code Coverage. In *46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3639477.3639733

*Work done while at Google

ICSE-SEIP '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0501-4/24/04.
https://doi.org/10.1145/3639477.3639733

1 INTRODUCTION

Code coverage was one of the first metrics used to quantify software testing, the idea being first published in 1963 by Miller and Maloney [13]. Many different code coverage criteria have been proposed over time [22], and many tools to measure code coverage are available [21]. Code coverage is a validation requirement in safety-critical domains [1], and it is widely used in other domains—in commercial and open-source projects alike. Code coverage has an intuitive interpretation, coverage tools are available for a wide variety of programming languages, and prior work has found that developers generally perceive code coverage as useful [9].

At a high level, there are two common use cases for computing code coverage during development: (1) computing a code coverage ratio (e.g., the ratio of covered to all statements) and comparing it to a pre-defined threshold; (2) visualizing code coverage and exposing untested code to authors and reviewers. The second use case is aligned with code-review practices commonly found in commercial and open source projects [19, 20].

To better understand to what extent code coverage actually guides developers towards writing tests and/or improves the efficiency of the code review process, we conducted a field experiment at Google, hiding code coverage information during 30k randomly selected code reviews. Our results were as follows:

- We found evidence that displaying code coverage information during code review significantly reduces the overall code review duration.
- We found *no* evidence that showing code coverage during code review improves coverage more than the improvement already provided by the code review process itself.

Overall, our findings suggest that code coverage visualization improves code review efficiency: When reviewers find untested but important code, they may ask for more tests, and the visualization can help them identify what is not tested. However, even with such visualization coverage reaches a saturation point that still leaves some critical code uncovered. While Google's codebase has a very high degree of code coverage after using code coverage automation for over a decade, we conjecture that saturation is, to some extent, a result of the lack of actionability of code coverage findings: Code coverage can expose untested code, but established coverage metrics treat each line, statement or branch of code as

equally important. This is not only in stark contrast to how developers actually use code coverage [9], but importantly code authors and reviewers need to use their intuition to separate code into "important" and "less important" to focus their testing and quality assurance efforts accordingly.

Motivated by these findings, we designed a novel approach to code coverage—termed *Productive Coverage*—that automatically generates actionable test goals which actively guide developers to untested and important code. Instead of developer intuition, Productive Coverage relies on two key bits of information to decide what is important: (1) is similar code well tested in the existing code base and (2) is similar code frequently running in production? We developed and deployed an implementation of Productive Coverage, and we evaluated it, focusing on the following research questions:

- RQ1 How do developers perceive Productive Coverage guidance?
- RQ2 Does Productive Coverage improve code coverage?
- RQ3 Does Productive Coverage affect code review duration?

Our results show that:

- RA1 Developers generally find Productive Coverage useful and report a positive experience.
- RA2 Productive Coverage improves code coverage over traditional code coverage visualization.
- RA3 Productive Coverage significantly reduces the time reviewers spend on the code review, with a small effect; it has no significant effect on the time the authors spend on the review.

The remainder of the paper is structured as follows: Section 2 provides background information and a summary of the code coverage infrastructure at Google. Section 3 details our field experiment of using traditional code coverage during modern code review. Section 4 introduces Productive Coverage and section 5 details its evaluation. Section 6 describes related work, and section 7 concludes the paper.

2 BACKGROUND

The research presented in this paper was conducted in an industrial setting at Google. The software testing and reviewing practices at Google are established and similar to what is done at other companies and in open source projects [19].

2.1 Change-based code review

A central aspect of software development at Google is the code review process [20], which is change-based (i.e., incrementally applied to all changes in the code base) and tool-assisted (i.e., information produced by automated analyses are available to authors and reviewers during the review).

At Google, tens of thousands of changes are reviewed per day. Changes are contained in a *changelist*: an atomic update to the company's centralized version control system, which consists of a list of files, the operations to be performed on these files, and the file contents to be modified or added. Once a developer is satisfied with a (local) code change, they create a changelist and send it to the code review system for review. The author and the reviewers exchange comments through the review system. In response to these comments, the author may modify the contents of the changelist. Once the author and the reviewers are satisfied with the changelist, it is merged into the version control system.

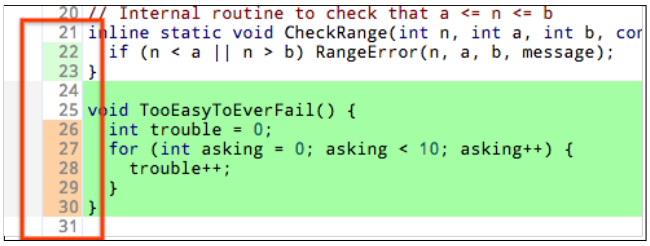

Figure 1: Baseline code coverage visualization (line numbers only): the color coding of a line number indicates whether it is covered (green) or uncovered (orange).

Developers' time is the most expensive aspect of the code review process at Google. All changelists are reviewed, and developers participate in the review process as author or reviewer multiple times per day. Around 70% of changelists have one reviewer (98% have three or fewer). Even though most changelists are small (43% of the changelists have 10 or fewer lines and 98% have fewer than 200 lines) and relatively quick to review, the total cost of the review process is significant. Optimizing the review process without compromising on quality is an important business goal; even a 1% improvement to review duration equals hundreds of developer-hours saved per day.

2.2 Code coverage infrastructure

Google has an extensive suite of analyses integrated in the code review process (e.g., linters, formatters, and automated testing) that aid authors and reviewers. Code coverage is among those analyses.

Google's coverage infrastructure supports more than ten programming languages and has been in use for about a decade. Our research focuses on the following languages: C++, Java, Python, Go, JavaScript, TypeScript and Dart. We selected these based on number of changelists and diversity of use cases (e.g., back-end vs. front-end development). Some changelists (about 3%) involve multiple languages, mostly combinations of a back-end and a front-end language (e.g., Java and JavaScript). The term *(code) coverage* in this paper refers specifically to line coverage, not counting empty lines and comments. Line coverage is automatically computed for almost all human-authored changelists, and in Google's code base, line coverage strongly correlates with statement coverage [9].

We refer readers interested in the development and evolution of this infrastructure, or how code coverage is used outside of the code review process, to the corresponding publication [9].

If coverage information is available, it is automatically displayed as part of the code review UI, as shown in Figure 1. Developers may choose to consume, and possibly act on, the *code coverage visualization*, or they can simply ignore it. Additionally, developers can click on a small "chip" in the code review UI to retrieve a *detailed code-coverage execution log*, which developers commonly use to investigate failures during coverage computation, e.g., failing tests. Note that developers can consume the code-coverage visualization without viewing this detailed report.

3 CODE COVERAGE DURING CODE REVIEW

Code coverage computation and visualization (using color-coded line numbers) have been available during code review at Google for a decade, and developers generally perceive code coverage as useful [9]. To quantify the effects of visualizing code coverage we conducted an experiment between March, 26 and June, 27 in 2019.

Our experiment sampled changelists submitted for code review and grouped them into an experiment group and a control group. The changelists were selected randomly, based on the last few digits of the changelist ID using different digit ranges for the two groups. The changelist ID is suitable for sampling: it is monotonically increasing across the code base and is not computed from any changelist data or meta data.

We modified the existing code-coverage infrastructure to fully compute the coverage information, but not display it for changelists in the experiment group: all coverage information was hidden, and any attempt to retrieve detailed code-coverage execution logs (through a report "chip") resulted in a custom message, which replaced that report and informed the developer that the changelist was part of an experiment. This custom message also allowed the developer to opt out of the experiment: a "Show coverage" link immediately displayed code coverage in the code review UI as usual.

Retrieving the detailed code-coverage execution logs is an explicit user action, which signals active interaction with the code-coverage infrastructure. Our experiment takes this into account and distinguishes between changelists for which this signal was observed and changelists for which it was not.

3.1 Datasets

As shown in Figure 2, our experiment sampled, over the duration of 3 months, 29,149 changelists for the experiment group and 28,577 changelists for the control group[1].

For the *Experiment* group, our experiment randomly selected 29,149 changelists. For 1,005 (3.4%) of these, developers attempted to retrieve the detailed code-coverage execution logs, and hence viewed the experiment notice and the opt-out instructions. We subdivided the Experiment dataset into two non-overlapping datasets:

- *Experiment (no logs)*: For 28,144 changelists, the developers did not attempt to retrieve the execution logs, and hence did not view the experiment notice.
- *Experiment (logs)*: For 398 changelists, the developers explicitly opted into the experiment: they attempted to retrieve the execution logs, viewed the experiment notice, but continued without code coverage information. For the remaining 607 changelists, the developers explicitly opted out of the experiment: they attempted to retrieve the execution logs, viewed the experiment notice, and clicked the "Show coverage" link.

For the *Control* group, our experiment randomly selected 28,557 changelists that are disjoint from the Experiment dataset, using the same sampling methodology during the same time period:

- *Control (no logs)*: For 26,915 changelists, the developers could observe the code coverage visualization but did not retrieve the execution logs.

[1]Due to confidentiality reasons, we cannot disclose all three of: (1) the experiment duration, (2) the exact sampling ratio, and (3) the total number of changelists. We opted to not disclose the exact sampling ratio, which is least important in our opinion.

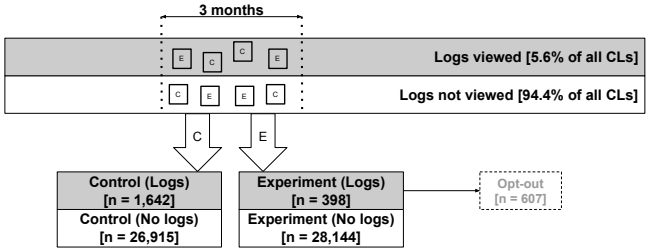

Figure 2: Summary of the datasets.
In the grey shaded "Logs" datasets, the detailed coverage execution log was viewed. In the white ones, it was not. The sizes of the boxes are not to scale.

- *Control (logs)*: For 1,642, the developers could observe the code coverage visualization and retrieved the detailed execution logs.

3.2 Measures of interest

Our research is concerned with the effects of showing code coverage during code review on the change in code coverage and the code-review duration. This section details how we measured these two variables of interest and what changelist characteristics we accounted for to minimize the risk of confounding.

We compare these measures for Control (no logs) vs. Experiment (no logs) to study differences resulting from withholding coverage information in the cases where developers did not (attempt to) view the detailed execution logs, and for Control (logs) vs. Experiment (logs) to study the differences in those cases where developers did.

3.2.1 Code coverage increase. The coverage infrastructure automatically computes coverage at the start of the code review, and repeatedly during the code review if the changelist is updated. To study the relationship between visualizing code coverage and code coverage increase, we measured the following:

(1) **Change in coverage percentage**: the difference between the coverage percentage at the end and at the start of the review. We computed this difference for all changelists for which coverage was computed more than once; for all other changelists, the difference is zero.

3.2.2 Review duration. We quantify the review duration for a changelist using three different measures:

(1) **Total wall-clock time**: the time (in minutes) between the changelist being emailed to the reviewers for the first time and it being submitted to the code base. This includes periods where no developer is in any way interacting with the changelist, for example night time.
(2) **Active shepherding time**: the time (in seconds) spent by the author of the changelist viewing and responding to reviewer comments and actively working on the changelist, between requesting the code review and merging the changelist in the code base. This includes time spent in the main code review tool and any other tools (e.g., editors) but does not include offline time, such as in-person conversations.
(3) **Active reviewing time**: the time (in seconds) spent by the reviewer in providing feedback. This includes time spent

in the main code review tool and time spent in other tools doing review related tasks, such as looking up APIs or documentation but not offline time spent on the review, such as in-person conversations or thinking about the code while drinking coffee.

These measures are available to us because the development environments and all tools used during the code review process are instrumented. The definitions of the active time measures follow the definitions described by Egelman et al. [5].

3.3 Experimental controls

Code coverage ratios and review times can be influenced by many factors. Based on feedback collected from developers and our own experience, we collected the following covariates for each changelist and used them as statistical controls:

(1) **Number of reviewers**: 70% of changelists have one reviewer and 98% have three or fewer. Changelists with more reviewers have longer total wall-clock time and unsurprisingly use more active reviewing and shepherding time.
(2) **Previous changelists with coverage**: familiarity of the author with coverage in particular and the code review process in general. To control for the fact that authors with longer tenure naturally have more changelists, only the 90 days before the changelist being studied are considered.
(3) **Previous changelists with the same reviewers**: familiarity of the reviewers with the author and vice versa.
(4) **Number of files in the changelist**: log-transformed number of files in the changelist. More files to review requires more time to review.
(5) **Instrumented lines at start of the review**: log-transformed number of instrumented lines, representing the order of magnitude of the changelist size. Instrumented lines are lines of code that have been instrumented by the coverage analysis infrastructure. Note that only executable lines can be instrumented. Comments, blank lines and other non-executable lines are not counted towards the changelist size.
(6) **Programming language**: one-hot encoded variable that indicates the primary programming language of the changelist. Since code coverage ratios for changelists vary across programming languages, we control for it.

3.4 Sampling method validation

To ensure that the Control and Experiment datasets are not systematically different we performed a logistic regression. Specifically, we modelled the group (Experiment vs. Control) as the outcome variable and used all variables listed in section 3.3 as predictors. In this model, none of predictor coefficients were statistically significant, with the lowest p-value being 0.064. Based on this analysis, we conclude that our sampling approach did not introduce bias and that the resulting datasets do not have systematically different characteristics that may confound the results.

3.5 Results

3.5.1 Change in Coverage. In total, there were 13,779 changelists with multiple coverage measurements in the Control (no logs) group and 13,870 in the Experiment (no logs) group. (There were 625 in the Control (logs) group and 229 in the Experiment (logs) group).

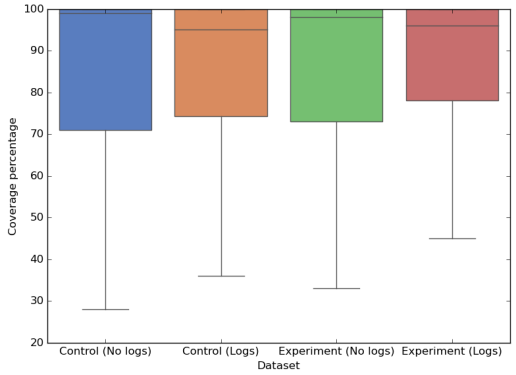

Figure 3: Code coverage percentage at the start of the code review process for all changelists per dataset.
The outliers are not plotted for readability.

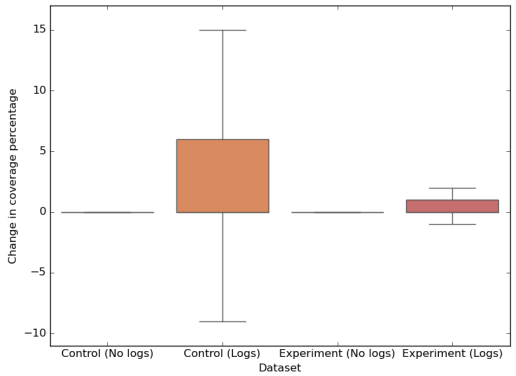

Figure 4: Change in code coverage percentage (start vs. end of the code review process) for all changelists per dataset.
The outliers are not plotted for readability.

Figure 3 shows the distribution of code coverage percentages at the start of the review. Given that code coverage is very high at the start of most changelists, with a median above 90%, the possible increase in code coverage is quite limited.

Figure 4 shows a boxplot of the change in coverage percentage. The median in all datasets is 0%, i.e., the coverage percentage did not change during the review. There are no statistically significant differences between the Control (no logs) and Experiment (no logs) datasets, nor between the Control (logs) and Experiment (logs) datasets. While Control (logs) shows higher variance in change percentage, the median in both datasets is still 0%.

This result indicates that visualizing test coverage during the code review process might not increase test coverage, beyond the increase already provided by the review process. These findings are consistent with prior work, showing that code review itself leads to improvements in both code and test quality [12].

However, the baseline coverage in our code base is very high and the code coverage visualization has been available for a decade.

Table 1: Mann-Whitney U test comparing the distributions of review duration between Experiment and Control groups.

"Execution logs" indicates whether a developer attempted to retrieve the detailed code-coverage execution logs.

Execution logs	Median duration		Difference	p-value
	Control	*Experiment*		
Total wall-clock time (min)				
No	1062	1116	5%	< 0.01
Yes	2663	3242	18%	0.35
Active shepherding time (sec)				
No	1374	1429	4%	< 0.01
Yes	4344	4409	1%	0.53
Active reviewing time (sec)				
No	584	658	11%	< 0.01
Yes	1816	1980	8%	0.56

Controlling for related learning and saturation effects would require us to consistently deny visualization to a subset of engineers, which would neither be practical nor ethical.

3.5.2 Change in review duration. We examined what effect showing code coverage has on code review duration. We hypothesized that visualizing code coverage (color-coded line numbers) speeds up the code review: When reviewing a changelist, a reviewer may want to check whether a given line is covered by tests. If code coverage is not visualized, this will take longer since the reviewer will have to inspect test files and verify that a corresponding test case exists.

Table 1 summarizes the statistical analysis of total wall-clock time in review. We can see a statistically significant difference between the Control (no logs) and Experiment (no logs) datasets with a 54 minute (5%) change in the median. The table also shows similar, significant results for active shepherding time and active reviewing time, with a change in the median of 55 seconds (4%) and 74 seconds (11%), respectively.

We found no statistically significant difference between the datasets where the developers attempted to retrieve the detailed execution logs. However, these datasets are considerably smaller and the observed differences are consistent with the other datasets.

> **Result:** In our experimental setting, visualizing line coverage does not significantly improve code coverage, beyond what is already provided by the review process. It does however make the review process more efficient.

4 ACTIONABLE CODE COVERAGE

Based on the observations from our field experiment, we developed a novel approach to code coverage—*Productive Coverage*—that aims at making code coverage more actionable. Given a changelist, it analyzes the changed code and identifies code that is uncovered but should be tested, if any such code exists. Figure 5 shows an example message posted by Productive Coverage during code review. Note the thumbs up and down buttons, which allow developers to provide feedback on each individual message, and the "Please fix" button,

Figure 5: Productive coverage message in the code review system.

which allows reviewers to request changes. (Clicking the "Please fix" button adds a comment in the reviewer's name on the same line containing prefilled text.)

We developed Productive Coverage based on two key insights: (1) Google's codebase is stored in a monolithic repository that contains the aggregate knowledge of tens of thousands of developers; (2) untested code (added or changed) should be tested if it is similar to existing code that is well tested or frequently executed in production. Developing Productive Coverage, therefore, required us to define three measures: code similarity, code testedness, and code execution frequency.

4.1 Code similarity

Defining and measuring code similarity requires a suitable abstraction over source code. We chose an abstraction based on our experience with the codebase, especially with source-code lines that are usually not tested. Here are two examples: (1) A call to `log.Info` aids debugging but is usually not subject to unit testing. (2) A statement of the form `throw new Exception` may generally be important to test, but the exception message itself (e.g., `throw new Exception("this should never happen")`) may indicate otherwise. Capturing this intuition, we chose an abstraction over identifiers and string literals, enabling Productive Coverage to identify and reason over such patterns.

Productive Coverage analyzes the abstract syntax tree (AST) and finds all nodes that represent identifiers (e.g., function calls or enum values) or string literals. It then processes these nodes as follows:

- A statement with a single identifier is used as-is (unigram).
- A statement that involves multiple identifiers is transformed into a sequence of bigrams.
- A string literal is transformed into a sequence of trigrams.

Consider the following example code (Python):

```
setContext()
x = complicated.Function(238)
y = common.Function("One two three four")
```

The first line is a function call with a single identifier (`setContext`), which is used as-is. The second line is an assignment that contains three identifiers, which are transformed into 2 bigrams: `[x, complicated])` and `[complicated, Function]`. The third line is an assignment with three identifiers, which are transformed as before, and one string literal, which is transformed into 2 trigrams: `["One", "two", "three"]` and `["two", "three", "four"]`.

Given this abstraction, a statement is represented by a set of n-grams and code similarity between two statements corresponds to the ratio of matching n-grams (exact matches).

Table 2: Code abstraction and similarity model summary

	C++	Go	Java	Python	Total
Number of n-grams	9,023	12,445	20,497	7,791	49,756
Identifiers	70.6%	61.7%	89.5%	66.8%	75.5%
Strings	29.4%	38.3%	10.5%	33.2%	24.5%

We chose this similarity metric because it scales very well to the size of our codebase. The model easily fits in the memory of a single average developer workstation and requires a very small amount of processor time to query. At the same time, the model performs well for our use case, as can be seen in Section 5.

4.2 Code testedness

For each individual n-gram (and the statement associated with it), Productive Coverage relies on code coverage and mutation testing information to compute 4 values: (1) number of times the n-gram occurs in the existing codebase, (2) number of times it was covered by existing unit tests, (3) number of times the associated statement was mutated, and (4) number of times a mutant of the associated statement was detected by existing unit tests.

Note that Productive Coverage computes these n-grams and the corresponding values across the entire codebase regularly, using historical analysis data for code coverage and mutation testing. These values are then normalized (coverage and mutation score individually to range from 0 to 1) and combined into a single testedness score per n-gram (product of coverage and mutation score). To ensure robust estimates of testedness for any given n-gram, we arbitrarily chose a threshold of 25 occurrences in the exiting code base—the testedness of an n-gram above this threshold is the combined score; the testedness of an n-gram below this threshold is set to 1, which prevents Productive Coverage from filtering it.

The testedness of an AST node is computed as the average testedness of all n-grams that form that node. The testedness of a statement is the aggregated testedness of all AST nodes that form that statement.

4.3 Code execution frequency

Google samples all code running in production. Productive Coverage scores each AST node based on the number of CPUseconds of the corresponding code: $score = \log_2(1.0 + CPUseconds)$. Small scores are set to 0; we chose 0.01 CPUseconds as the threshold. The CPU usage is propagated up the AST. For example, a node representing an `if` statement will have a CPU usage that equals the total CPU usage of all statements contained in its body.

4.4 Final scoring model

As a performance optimization and to minimize model size, we chose to only store n-grams whose testedness score is below 0.5—that is, n-grams with sufficient occurrences that are "not tested enough". During inference, an n-gram not present in the stored model is treated as "well tested" and assigned the maximum score of 1.0. This optimization also treats any never-before-seen n-gram as maximally important, which is a conservative choice to bias the model towards "code should generally be tested".

As of the writing of this paper, the code abstraction and similarity model contains 49,756 n-grams with a testedness score of less than 0.5. Table 2 summarizes the most important high-level measures of the model. Due to industrial confidentiality reasons, we cannot disclose more information about the n-grams in the model.

4.5 System architecture

After a developer creates or modifies a changelist, the existing coverage service runs tests and computes line coverage. A successful computation of line coverage is a prerequisite of Productive Coverage, which involves four main steps:

(1) Productive Coverage filters out all test and test helper code.
(2) Productive Coverage creates an AST for all changed code and scores all AST nodes.
(3) Productive Coverage filters low-scoring AST nodes.
(4) Productive Coverage returns the highest scoring nodes.

Processing a single changelist is fast and Productive Coverage is trivially parallelizable across changelists; it can be deployed as a distributed service that scales with the number of changelists.

4.5.1 Filtering tests and test helper code. Unit test code is itself code and is also reviewed as part of the code review process. However, the only way unit test code can be not covered by unit tests is if the tests did not run. While this can happen if the project is misconfigured, other much simpler tools exist to detect this problem. The same applies to test helper code, i.e., test libraries that are only used to avoid code duplication in test code. To reduce the noise in the system, Productive Coverage removes tests and test helper code from all computations. Tests are easily identified because they directly import the unit testing framework. Test helpers are identified using the dependency graph. This step eliminates entire files and is done as a significant resource optimization.

4.5.2 Scoring AST nodes. Productive Coverage parses all changed code into an AST and its model can score any AST node based on that node's similarity to other nodes corresponding to code already in the codebase. This model has a small memory footprint and can be kept in memory. Specifically, Productive Coverage uses a hash map to score all AST nodes, where nodes with multiple n-grams are scored by aggregating the scores of the individual n-grams.

4.5.3 Filtering low-scoring AST nodes. AST nodes that are already covered by tests can be trivially filtered out: the automated coverage infrastructure provides covered lines and during parsing each AST node preserves the line range of the source code it represents. Productive Coverage operates on AST nodes, but the coverage infrastructure computes coverage at the line level. However, modern parsers annotate AST nodes with their byte and line range. If the AST node represents only a single line, or part of a line, then mapping that to coverage is trivial. If the AST node represents multiple lines, we consider the node covered if all lines represented by the node are covered. The AST node is uncovered if no lines represented by the node are covered. If some lines are covered and some are not, the node is mixed. Comments, whitespace and other non-code lines are ignored. In this step, all fully covered and mixed AST nodes are filtered. Figure 6 shows the ratio of the number of comments by the number of lines of code represented by that comment. 98.5% of all comments cover fewer than 15 lines.

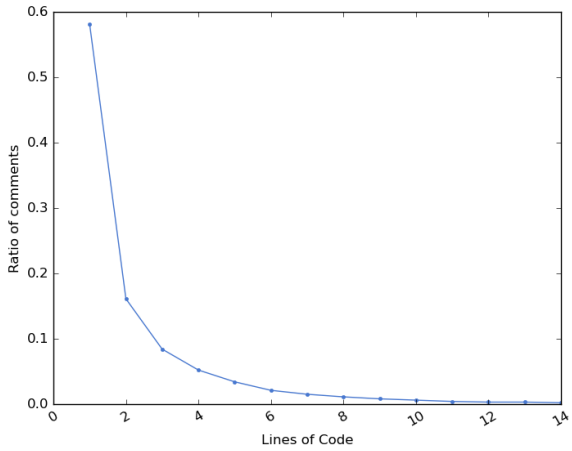

Figure 6: Number of lines of code represented by productive coverage comments.

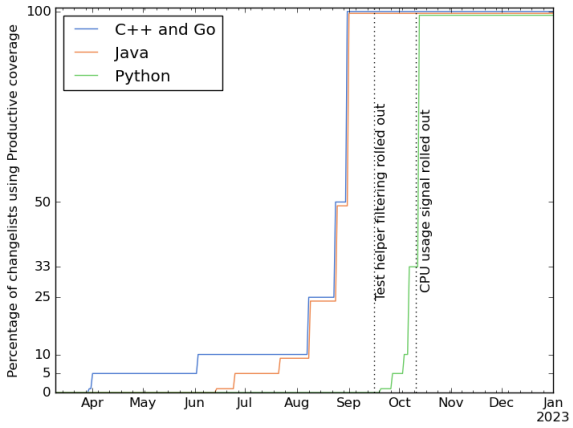

Figure 7: Timeline of Productive coverage deployment. Note that identical percentages were used for C++, Go, and Java. The lines are slightly moved to avoid visual overlap.

Remaining uncovered AST nodes are then filtered based on their testedness score—nodes with a score below 0.5 are discarded.

4.5.4 Selecting the highest scoring nodes. For each changed file in a changelist, we limit the number of lines Productive Coverage can post a message on, to prevent the developer from being overwhelmed. To that end, the AST nodes are sorted by their aggregated testedness score and only the top scoring nodes are selected.

4.6 Deployment

Productive coverage was deployed in 2022. Figure 7 shows the timeline of the deployment. The deployment was spread out over a period of 7 months to allow us to improve the quality of the tool based on user feedback. Productive Coverage was initially posting comments on a very limited set of changelists, randomly selected based on the last two digits of their ID. Two improvements that were implemented during the rollout period were the test helper filtering, implemented in September, and the code execution frequency signal, implemented in October. We chose to initially validate our idea on two languages, C++ and Go. Once the idea proved viable, Java and Python were implemented based on business priorities.

While Productive Coverage was *posting* comments only on a percentage of changes, it was computing testedness scores for all changes during the entire period and storing comments it would have posted, but did not, in a database for analysis. This set of "counterfactual comments" that were never posted offers a baseline for measuring the effect of the tool.

Productive Coverage uses our existing mutation testing system, which suppresses mutants based on arid node heuristics [16, 17]. Arid nodes are AST nodes that developers are generally not interested in testing, and hence they are never mutated [15]. This has implications for Productive Coverage's testedness score, which is low for such nodes. While a low testedness score is arguably the correct score for such AST nodes, we leave a deeper investigation into the effects of arid node suppression for future work.

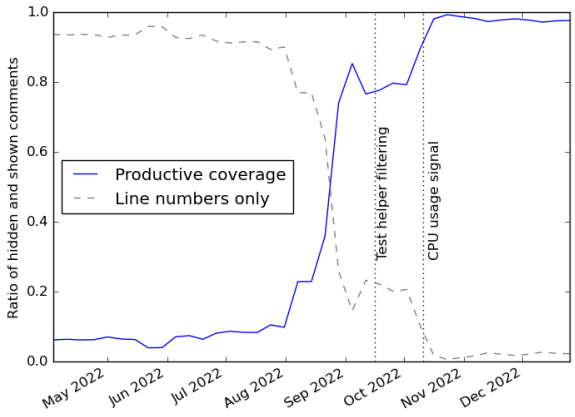

Figure 8: Ratio of hidden and shown comments.

5 EVALUATION

During deployment, 554,067 comments were shown by the tool to developers. During the roll-out, comments were posted only on a percentage of changes but the analysis was performed on all applicable changes. Figure 8 shows the ratio of hidden and shown comments per week of roll-out. Due to industrial confidentiality reasons we cannot disclose the total number of shown and hidden comments per week or month.

5.1 Developer sentiment

Figure 9 shows the ratio of positive, negative and mixed developer sentiments across the deployment period. Positive feedback means that one or more developers clicked either the "Thumbs up" or the "Please fix" buttons shown in Figure 5. Negative feedback means that one or more developers clicked the "Thumbs down" button shown in Figure 5. Mixed feedback means that the comment received both positive and negative feedback, sometimes from different people. It should be noted that the higher volatility at the start of the year is due to smaller total number of comments at those times. The overall trend of decreasing negative feedback shows the effect of

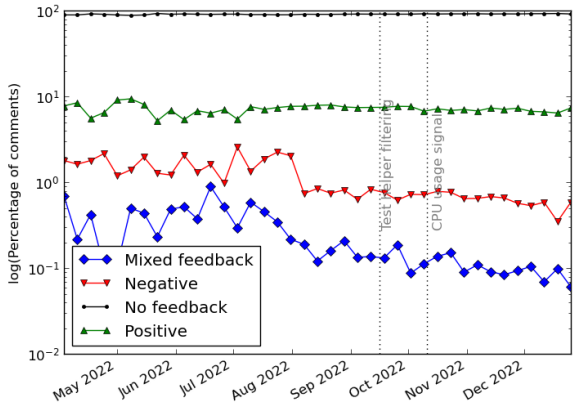

Figure 9: Percentage of comments by type of feedback they received.
Note the log-scaled y-axis. Also note that the volatility of the last data points is due to smaller populations (winter holiday period).

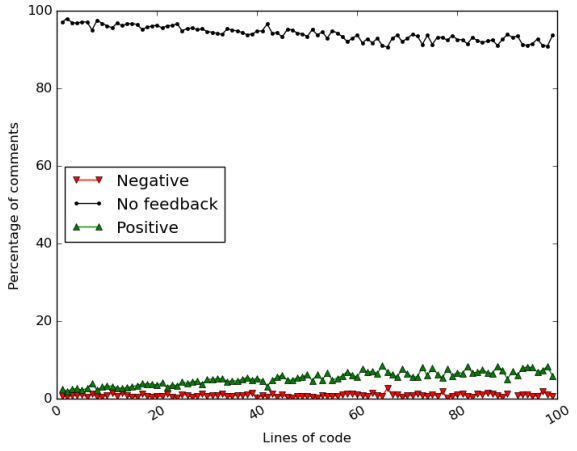

Figure 10: Percentage of negative, positive and comments without feedback by size of change.
The "Mixed feedback" category is not plotted for clarity. It is the smallest category and lacks sufficient data points (see Figure 9).

continuous improvements to arid node heuristics, the testedness model and the rollout of the execution frequency signal.

Figure 9 shows that developers are 10 times more likely to be satisfied by the comments posted by the tool than be dissatisfied by it. However, it should also be noted that for almost 90% of comments the developers did not provide any explicit feedback. Figure 10 shows one significant factor that explains this; developers are more likely to provide feedback on comments posted on larger changelists. Since most changelists are small, this leads to relatively small overall feedback rate. Note that the positive feedback rate increases with changelist size and the negative rate remains constant.

RQ1: Developers are generally satisfied with the actionability of the comments provided by Productive Coverage.

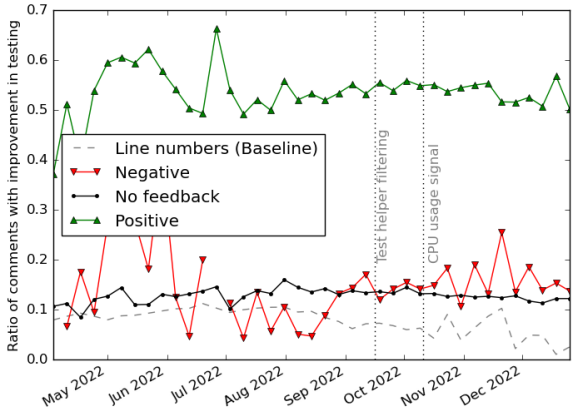

Figure 11: Ratio of comments where coverage was improved, by human feedback.
The "Mixed feedback" category is not plotted for clarity. It is the smallest category and lacks sufficient data points (see Figure 9). The high volatility of each line during some periods is due to smaller populations.

5.2 Change in coverage

During the lifetime of the changelist, tests can be added, changed or removed, changing the coverage of the lines of code represented by the AST node that any comment is attached to. For any given individual comment, we can first determine which, if any, lines at the end of the code review map to the lines represented by the comment. We then compute the coverage of both sets of lines and say that the coverage has improved if the coverage percentage of the lines represented by the comment at the end of the review is greater than the coverage percentage of the lines represented by the comment at the start of the review. Figure 11 shows the ratio of comments where the line where the comment was posted was covered by tests more at the end of the code review than at the start of the review. Note that the high volatility of each line during some periods is due to smaller populations. As can be clearly shown from the chart, when the automated comment receives positive feedback, it leads to improved coverage more than 50% of the time. If the comment does not receive any feedback the coverage will improve 12% of the time, and even comments with negative feedback still lead to improved coverage 13% of the time. The baseline, coverage improvement due to the review process itself and previous automation is 8%.

Another way to improve coverage during the review is to modify the code under test. We track the lifetime of the comment using a modified, high performance re-implementation of GumTreeDiff [6]. During the lifetime of a changelist, the code that the comment is attached to can change in 4 ways:

- **Not moved**: There are no changes to the file containing the code. The code maps to the exact same location.
- **Moved**: The code in the file was modified, moving the code represented by the comment to different line numbers, but the code representing the comment was not modified at all, or was only slightly changed. The code maps to a single location at the end of the review.

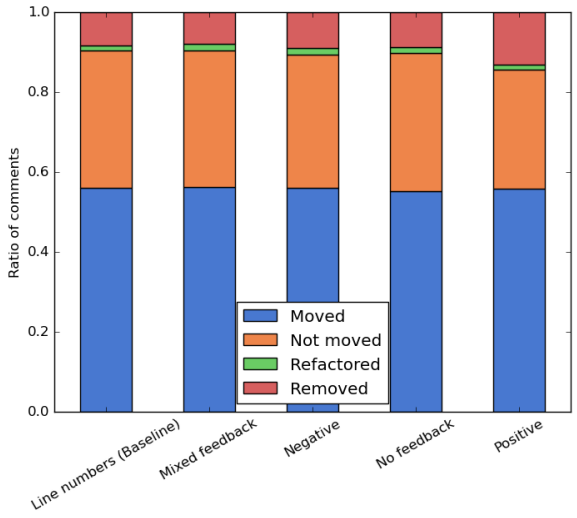

Figure 12: Location of code represented by the productive coverage comment at the end of the review.
The distribution was stable across the deployment period. Only the "Positive" feedback group significantly differs from the baseline.

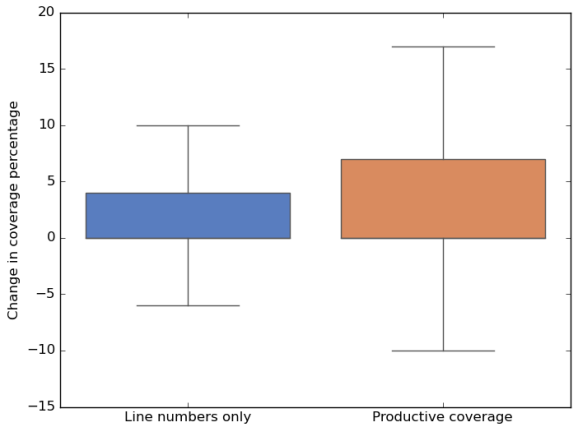

Figure 13: Change in code coverage percentage (start vs. end of the code review process), with and without Productive Coverage comments shown in the review process.
The change is 0 in 47% of the changes where the comments were hidden and 41% of changes where the comments were shown. The median in both categories is 0. The outliers are not plotted for readability.

- **Refactored**: The code represented by the comment was refactored and moved into multiple new places, but the connection between the new locations and old can still be seen; for example, if the inner code of a for loop is refactored into a separate function. The code maps to two or more code locations at the end of the review.
- **Removed**: The code was deleted or modified beyond recognition. The code maps to no location at the end of the review.

Figure 12 shows the percentage of comments in each category. The baseline, no feedback, mixed and negative have no significant differences, but the positive comments are more likely to have the lines where the comment was posted no longer present at the end of the review. We manually examined 200 randomly selected comments from the "Removed" category: 100 with positive feedback and 100 baseline counterfactuals; we found no difference between the two. From the 200 comments we examined, 4 categories emerged:

(1) The lines of code were unnecessary and should not have been included in the change. The author deleted the lines. Roughly 20% of comments were in this group.
(2) The lines of code were refactored keeping the logic of the code exactly the same, but changing the code so much that the diffing algorithm was no longer able to identify the code as moved. The most common case here is moving the code from one file to the other, since we did not even attempt cross-file diffing. Roughly 30% are in this group.
(3) The code was modified changing the behaviour of the code. Commonly, the change was in the error handling code and the way errors are propagated or not was changed. Less frequently, the code had a bug and the bug was fixed. Roughly 30% are in this group.
(4) The review stretched for so long that the codebase advanced and a sync to head was required. The author possibly reverted the change completely and started from a fresh client, or they performed a three way merge that significantly changed to code making all previous analysis obsolete. Roughly 10% are in this group.

Based on this, we conclude that Productive Coverage leads to improvements in code quality. The 5% increase in removed code in the positive feedback category (13%) compared to all other categories (8%) provides us with a lower bound for this code improvement. Given that roughly 10% of all productive coverage comments receive positive feedback we estimate that at least 1 in every 200 productive coverage comments leads to direct code improvements.

To illustrate how this happens, we provide the following comment exchange between a reviewer and an author; a real example we observed during our examination:

Reviewer: Please fix.

> Maybe add a test case that verifies that the [codename] keeps going, if it is straightforward to stage data that will cause the call above to fail?

Author: Done.

> This caught a bug here (the `continue`) so was well worth adding.

Figure 13 shows the change in coverage at the start and at the end of the code review. This chart is provided for comparison to the withholding experiment described in section 3.5.1.

Based on the observed improvements to tests and code, we conclude that:

RQ2: Productive coverage leads to a meaningful improvement in test and code quality, despite a strong baseline.

Table 3: Mann-Whitney U test comparing the distributions of review duration between the baseline code coverage visualization (line numbers only) and Productive Coverage.

Median duration		Difference	p-value
Line numbers only	*Productive Coverage*		
Total wall-clock time (min)			
1732	1728	-0.23%	0.80
Active shepherding time (sec)			
1954	1959	0.26%	0.76
Active reviewing time (sec)			
1495	1460	-2.34%	< 0.01

5.3 Change in review duration

We examined the effects of showing Productive Coverage comments on code review duration, using the same three time measures described in Section 3.2. Table 3 summarizes the results.

The active reviewing time differs significantly (two-sided Mann-Whitney U test; $p < 0.01$) between the changes where the comments were not shown (line numbers only) and those were they were (Productive Coverage). The medians of active reviewing time are 1495 and 1460 seconds when the comments are hidden and shown respectively, showing a 2.34% decrease in active review time when productive coverage is used.

In contrast to active reviewing time, we did not observe a significant difference in the total wall-clock time, nor in the active shepherding time. The differences in medians of -0.23% and 0.26%, respectively, are not statistically significant. Note that the review times in tables 1 and 3 are not directly comparable due to different populations (difference in number of programming languages and all changelists vs. coverage-eligible changelists).

> **RQ3:** Productive Coverage significantly reduces the time reviewers spend on the code review, with a small effect; it has no significant effect on the time the authors spend on the review.

5.4 Threats to validity

The primary limitation of our study, regarding external validity, is the experiment context and research setting, which is limited to a single company. While developers, programming languages, and tools all form a large and diverse sample, it may still be too company-specific. For example, the unit testing practices and high code coverage baseline may not be representative.

Regarding construct validity, our chosen proxy measures such as review time may not accurately capture the conceptual variables of interest, such as reviewer effort and process efficiency. We tried to mitigate this threat as much as possible by incorporating precise measures such as active review time.

A threat to internal validity are potential carry-over-effects. Our dataset contains very few observations of established projects choosing to adopt coverage late in the development process. Because coverage has been established at Google for many years, older projects have adopted it many years ago, and new projects are using it from the start. This means that our results do not necessarily reflect what would happen in the short term if coverage is introduced into existing projects with weak baseline coverage.

6 RELATED WORK

Code coverage has been studied extensively since it was introduced, however the focus of studies has been test suite quality. Marick [11] considers many ways in which such use can go wrong. Inozemtseva and Holmes [8] analyzed five projects with up to 724 KLOC. They found low to moderate correlation between coverage and defects, and Kochar et al. [10] analyzed two projects with 120 KLOC and found moderate to strong correlation between coverage and defects. Chen et al. [4] analyzed the relationship between a test set's size, its code coverage adequacy, and its fault detection capability. None of these studies, however, explore the effects of code coverage on any other metric, and while test suite quality is important, industrial practitioners must consider the developer workflow holistically.

Some research does analyze other metrics, often to increase the prediction power. Chen et al. [3] for example augment testing time with coverage. Antinyan et al. [2] report the results of an analysis of adequacy of unit test coverage criteria at Ericsson, and give results of a literature survey that lists 8 related works. Their analysis was performed on a single 2 MLOC project worked on by 150 engineers. They found very low correlation between coverage and defects. Piwowarski et al. [18] report on a case study from 1991, performed at IBM on a single 780 KLOC project. They found that the cost of removing errors dropped by 20% when coverage computation was added to the developer workflow and when improving coverage was set as a goal. Most of the reduction in error removal cost came as a result of fewer errors in the shipped product. Mockus et al. [14] report on two case studies from 2009 performed at Microsoft and Avaya, on two project with 40+ MLOC and 1 MLOC respectively. They found that improving coverage reduced the number of errors in the released product, and that the effort required to improve coverage increases disproportionally with coverage, with the study concluding that "the optimal levels of coverage are likely to be well short of 100%.". As shown in this paper, while 100% is indeed difficult to achieve, 'well short' is perhaps a bit too pessimistic.

Gopinath et al. [7] provide a large-scale analysis, having analyzed 1,254 open source projects. They found that statement coverage is "probably the best coverage criteria for predicting test suite quality in their context, over testing effort lifetime", but more interestingly they commented that the best criteria for research purposes differ from those for practitioners, and that the focus of practitioners is not on evaluating testing methods but on producing quality software. Our findings seem to be in line with this observation.

7 CONCLUSIONS

Code coverage is an intuitive and established test adequacy measure. However, not all parts of the code base are equally important, and hence additional testing may be critical for some uncovered code, whereas it may not be worthwhile for other uncovered code. As a result, simply visualizing uncovered code is not actionable.

This paper proposes a novel approach to code coverage — termed Productive Coverage — that aims to make code coverage findings actionable by pointing out uncovered code that should be tested.

Productive Coverage is based on the principle that uncovered code should be tested if it is similar to existing code that is well tested and/or frequently executed in production. Conversely, if uncovered code is similar to code that is rarely, if ever, tested, alerting a developer to the fact that the code is uncovered would likely result in a false-positive warning.

We implemented Productive Coverage for four programming languages (C++, Java, Go, and Python), and our evaluation shows:

- The developer sentiment towards Productive Coverage, measured at the point of use, is strongly positive.
- Productive Coverage meaningfully increases code coverage above a strong baseline—an established code review process and color-coded line numbers.
- Productive Coverage has no adverse effect on code authoring efficiency.
- Productive Coverage modestly improves the code-review efficiency, by reducing the active review time.
- In addition to improving test quality, Productive Coverage measurably improves code quality and prevents defects from being introduced into the codebase.

REFERENCES

[1] DO-178C — Software Considerations in Airborne Systems and Equipment Certification, December 2021.

[2] Antinyan, V., Derehag, J., Sandberg, A., and Staron, M. Mythical unit test coverage. *IEEE Software 35* (05 2018), 73–79.

[3] Chen, M.-H., Lyu, M. R., and Wong, W. E. Effect of code coverage on software reliability measurement. *IEEE Transactions on reliability 50*, 2 (2001), 165–170.

[4] Chen, Y. T., Gopinath, R., Tadakamalla, A., Ernst, M. D., Holmes, R., Fraser, G., Ammann, P., and Just, R. Revisiting the relationship between fault detection, test adequacy criteria, and test set size. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (2020), pp. 237–249.

[5] Egelman, C. D., Murphy-Hill, E., Kammer, E., Hodges, M. M., Green, C., Jaspan, C., and Lin, J. Predicting developers' negative feelings about code review. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)* (2020), IEEE, pp. 174–185.

[6] Falleri, J., Morandat, F., Blanc, X., Martinez, M., and Monperrus, M. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014* (2014), pp. 313–324.

[7] Gopinath, R., Jensen, C., and Groce, A. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering* (2014), ACM, pp. 72–82.

[8] Inozemtseva, L., and Holmes, R. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering* (2014), ACM, pp. 435–445.

[9] Ivanković, M., Petrović, G., Just, R., and Fraser, G. Code coverage at google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2019), ESEC/FSE 2019, ACM, pp. 955–963.

[10] Kochhar, P. S., Thung, F., and Lo, D. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)* (2015), IEEE, pp. 560–564.

[11] Marick, B. How to misuse code coverage. In *Proceedings of the International Conference on Testing Computer Systems (ICTCS)* (June 1999), pp. 16–18.

[12] McIntosh, S., Kamei, Y., Adams, B., and Hassan, A. E. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering 21*, 5 (2016), 2146–2189.

[13] Miller, J. C., and Maloney, C. J. Systematic mistake analysis of digital computer programs. *Commun. ACM 6*, 2 (Feb. 1963), 58–63.

[14] Mockus, A., Nagappan, N., and Dinh-Trong, T. T. Test coverage and post-verification defects: A multiple case study. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement* (USA, 2009), ESEM '09, IEEE Computer Society, p. 291–301.

[15] Petrović, G., Ivanković, M., Fraser, G., and Just, R. Does mutation testing improve testing practices? In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (2021), IEEE, pp. 910–921.

[16] Petrović, G., Ivanković, M., Fraser, G., and Just, R. Practical mutation testing at scale: A view from google. *IEEE Trans. Softw. Eng. 48*, 10 (oct 2022), 3900–3912.

[17] Petrović, G., Ivanković, M., Fraser, G., and Just, R. Please fix this mutant: How do developers resolve mutants surfaced during code review? In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (2023), IEEE, pp. 150–161.

[18] Piwowarski, P., Ohba, M., and Caruso, J. Coverage measurement experience during function test. In *Proceedings of the 15th international conference on Software Engineering* (1993), IEEE Computer Society Press, pp. 287–301.

[19] Rigby, P. C., and Bird, C. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2013), ESEC/FSE 2013, Association for Computing Machinery, p. 202–212.

[20] Sadowski, C., Söderberg, E., Church, L., Sipko, M., and Bacchelli, A. Modern code review: A case study at google. In *International Conference on Software Engineering, Software Engineering in Practice track (ICSE SEIP)* (2018).

[21] Yang, Q., Li, J. J., and Weiss, D. M. A survey of coverage-based testing tools. *The Computer Journal 52*, 5 (2009), 589–597.

[22] Zhu, H., Hall, P. A., and May, J. H. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR) 29*, 4 (1997), 366–427.