Code Coverage at Google

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

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

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 a measure of the degree to which a test suite exercises a software system. Although coverage is well established in software engineering research, deployment in industry is often inhibited by the perceived usefulness and the computational costs of analyzing coverage at scale. At Google, coverage information is computed for one billion lines of code daily, for seven programming languages. A key aspect of making coverage information actionable is to apply it at the level of changesets and code review.

This paper describes Google's code coverage infrastructure and how the computed code coverage information is visualized and used. It also describes the challenges and solutions for adopting code coverage at scale. To study how code coverage is adopted and perceived by developers, this paper analyzes adoption rates, error rates, and average code coverage ratios over a five-year period, and it reports on 512 responses, received from surveying 3000 developers. Finally, this paper provides concrete suggestions for how to implement and use code coverage in an industrial setting.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**; **Software testing and debugging**; **Empirical software validation**; *Collaboration in software development*.

KEYWORDS

coverage, test infrastructure, industrial study

ACM Reference Format:
Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 2019. Code Coverage at Google. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3338906.3340459

1 INTRODUCTION

Code coverage is a well established concept in computer science, and code coverage criteria such as statement coverage, branch coverage, and modified condition/decision coverage (MC/DC [11]) are well-known measures for test suite adequacy. For example, MC/DC-adequacy is required for safety-critical systems (RTCA DO-178). Code coverage has been known and applied for decades. For example, Piwowarski et al. [25] mention that IBM performed code coverage measurements in the late 1960s, and Elmendorf [13] provided a reasonably robust strategy for using branch coverage in testing operating systems as early as 1969.

While code coverage is commonly used in software engineering research, its effectiveness at improving testing in practice is still a matter of debate [14, 18] and its adoption in industry is not universal. A large survey at IBM on their use of code coverage [25] revealed that once code coverage was adopted, it led to an increase in test suite quality; however, ease of use and scalability are primary factors in whether code coverage is adopted in the first place. Existing industry reports on code coverage are typically based on code bases of manageable size with one or two programming languages. However, even at that scale, these reports raise concerns about the cost-effectiveness of code coverage [20]. In particular, they raise concerns that both the machine resources required to compute coverage as well as the developer time required to process the results are too costly compared to the increase in test suite quality.

Addressing and overcoming these concerns, Google has spent more than a decade refining its coverage infrastructure and implementing and validating the academic approaches. Google's infrastructure supports seven programming languages and scales to a codebase of one billion lines of code that receives tens of thousands commits per day. The paper details Google's code coverage infrastructure and discusses the many technical challenges and design decisions. This coverage infrastructure is integrated at multiple points in the development workflow. This paper describes this integration, and reports on the adoption and perceived usefulness of code coverage at Google, by analyzing five years of historical data and the 512 responses, received from surveying 3000 developers.

In summary, this paper makes the following contributions:

- It demonstrates that it is possible to implement a code coverage infrastructure based on existing, well-established libraries, seamlessly integrate it into the development workflow, and scale it to an industry-scale code base.
- It details Google's infrastructure for continuous code coverage computation and how code coverage is visualized.
- It details adoption rates, error rates, and average code coverage ratios over a five-year period.
- It reports on the perceived usefulness of code coverage, analyzing 512 responses from surveyed developers at Google.

The remainder of this paper is structured as follows. Section 2 provides background information and terminology. Section 3 details Google's code coverage infrastructure. Section 4 describes the adoption of code coverage and its usefulness, as perceived by Google developers. Section 5 discusses related work. Finally, Section 6 concludes and discusses future work.

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08.
https://doi.org/10.1145/3338906.3340459

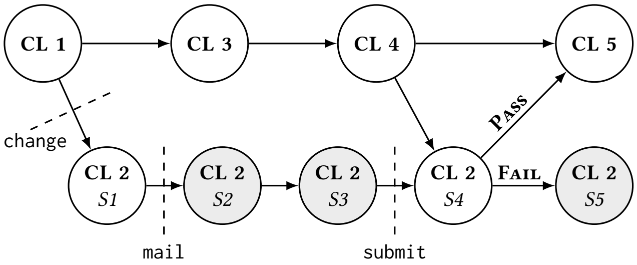


Figure 1: The life cycle of a changelist (CL).

2 BACKGROUND

Code coverage is a well established concept, but also a broad term that requires a more precise definition when applied in practice. This section defines important terms and concepts used throughout the paper.

2.1 Projects and Changelists

The following two concepts are required to understand the development workflow at Google and its integration of coverage:

- **Project**: A project is a collection of code that the project owners declare in a project definition file. The project definition contains a set of **code paths** (i.e., patterns describing a set of files) and a set of test suites and organizational metadata such as a contact email address.
- **Changelist**: A changelist is an atomic update to Google's centralized version control system. It consists of a list of files, the operations to be performed on these files, and possibly the file contents to be modified or added.

Figure 1 illustrates the life cycle of a changelist using an example. CL 1 represents the starting state of the codebase. Suppose that at this point a developer issues a `change` command, which creates a new changelist, CL 2. Initially, this new changelist is in a "pending" state and the developer can edit its contents. As the developer edits CL 2, it progresses through so called "snapshots" (represented in the diagram as *S1* to *S5*). Once the developer is satisfied with the contents of a changelist, they issue a `mail` command which initiates the code review process. At this point, automated analyses including code coverage are started and a notification is emailed to the reviewers. The changelist is now said to be "in review". Through the review process the contents may change, leading to different snapshots. When the reviewers and the author approve of a changelist, the author issues a `submit` command. In the example above, this command first merges CL 2 with the current submitted state of the codebase (CL 4) and, if the merge is successful, runs more automated tests. Only if the merge is successful and all tests pass is the changelist submitted to the codebase. To ensure that changelist numbers in the codebase are monotonically increasing, CL 2 is also renamed to CL 5 on submit. (CL 2 becomes a pointer and anyone accessing it is forwarded to CL 5.) If the submit command fails, the changelist returns back to the "in review" state and the CL author can continue to change its contents, usually addressing the failure.

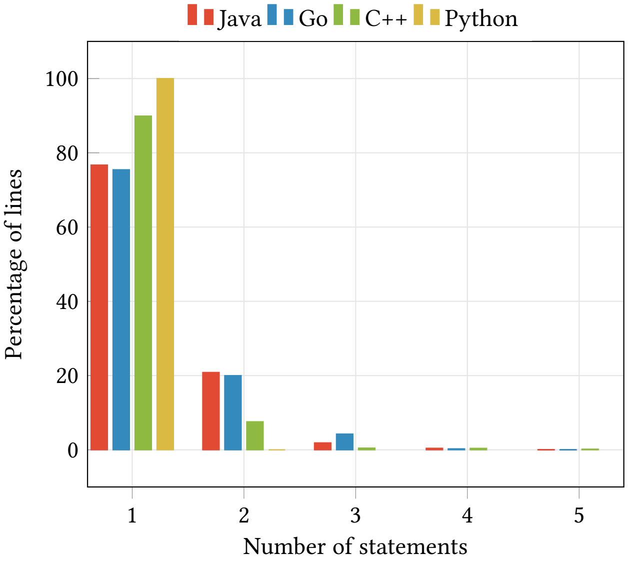


Figure 2: Distribution of number of statements per line for C++, Java, Go, and Python in Google's codebase.
Multi-statement lines are predominantly `for` loops and idiomatic returns. Lines with more than 5 statements are removed for clarity; in total, they contribute less than 1%.

2.2 Code Coverage

Google's code coverage infrastructure measures **line coverage**, which is the percentage of lines of source code executed by a set of tests. Note that blank lines and comments do not count as lines of code, and hence do not contribute towards the denominator. Because Google enforces style guides for all major languages [15], line coverage strongly correlates with statement coverage in Google's codebase. Figure 2 gives the distribution of number of statements per line. Most lines contain a single statement, and lines with more than one statement predominantly contain loops (initialization, condition, update) and non-trivial, nested return statements.

For C++, Java, and Python, Google's code coverage infrastructure also measures **branch coverage**, which is the percentage of branches of conditional statements (e.g., if statements) executed by a set of tests. Branch coverage is equivalent to edge coverage for all conditional edges in the control flow graph [27].

This paper distinguishes between two coverage *scopes*:

- **Project coverage**: Line coverage of a project's test suites over that project's code paths. Project coverage is computed once per day for a given project.
- **Changelist coverage**: The union of line coverage of all test suites of all projects that are affected by the changelist—that is, all projects whose code paths appear in the changelist diff. Changelist coverage is computed one or more times for a given changelist. Within the changelist coverage scope, this paper distinguishes between:
  - **Overall coverage**: The code coverage of the whole contents of the files in a changelist.
  - **Delta coverage**: The code coverage of only the lines that were modified or added in a changelist. (Deleted lines no longer exist, and hence cannot be covered.)

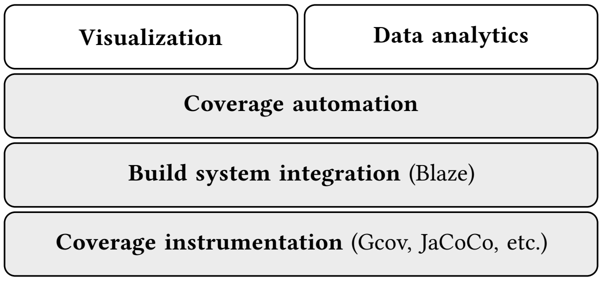

Visualization
Data analytics
Coverage automation
Build system integration (Blaze)
Coverage instrumentation (Gcov, JaCoCo, etc.)

Figure 3: Google's code coverage infrastructure.

2.3 Unit Tests

This paper considers only code coverage of tests that execute on a single machine, usually in a single process. At Google, these are colloquially referred to as *unit tests*. This paper does not consider code coverage of more complex (i.e., integration or system) tests that span multiple machines. Since integration tests focus on integration points between (sub)systems and not the internals of each (sub)system, line coverage is not the best suited coverage measure for these tests.

3 INFRASTRUCTURE

This section describes Google's code coverage infrastructure, which has been actively developed since 2012. It has evolved since to improve compatibility and address many challenges. Figure 3 gives a high-level architectural overview, showing the four main layers of the infrastructure, which are described in the subsequent sections.

3.1 Coverage Instrumentation

Google's code coverage infrastructure leverages a combination of well established code coverage libraries and internally developed solutions to support coverage computation for a variety of programming languages. Specifically, the infrastructure uses:

- **C++**: `gcov` [7].
- **Java**: `JaCoCo` [5] and additional, internal support infrastructure for Android.
- **Python**: `Coverage.py` [2].
- **Go**: Go has built in coverage support (`go -cover`) [6].
- **JavaScript**: `Istanbul` [4] and an internally developed coverage library, plus significant support infrastructure built around them.
- **Dart**: Dart-lang has built in coverage support [3].
- **TypeScript**: Re-uses much of the JavaScript infrastructure.

To address the structural challenges of integrating all of these libraries and unifying their output, external libraries were modified or built on top. In particular, all coverage libraries were either modified to produce `lcov`-formatted output, or custom infrastructure was developed to transform the output of a library into `lcov` format. The `lcov` format is the text-based tracefile format that `gcov` [7] and it's frontend `lcov` use to store coverage information. Google's code coverage infrastructure uses a slightly modified version of the format with slightly different branch coverage notation that does not require internal GCC IDs to identify the branch.

Table 1: Percentage of developers manually invoking `blaze coverage` at least once in the given time period. All numbers are the average value in 2018.

FREQUENCY	%
Daily manual invocation of `blaze coverage`	0.5%
Quarterly manual invocation of `blaze coverage`	10%
Yearly manual invocation of `blaze coverage`	30%

On the conceptual side, `lcov` supports line and branch coverage, but other types of code coverage can be simulated; for example, function coverage can be simulated by only marking the first line of each function as instrumented. Focusing on line coverage proved to be a good practical choice, because it strongly correlates with statement coverage (recall Figure 2) and is easy to visualize and comprehend; Section 3.4 provides details.

3.2 Build System Integration

Google uses *Blaze* as its core build system. Blaze is an internal version of the Bazel build system [17] and supports building binaries, running tests, and collecting code coverage information.

One way to trigger coverage computation is to invoke the `coverage` command for a set of code paths, using the Blaze command-line interface. Blaze automatically handles the coverage computation, abstracting over the implementation details of the underlying instrumentation libraries. Table 1 shows that manual invocations are rare. For example, on a daily basis only about 0.5% of developers manually invoke `blaze coverage`.

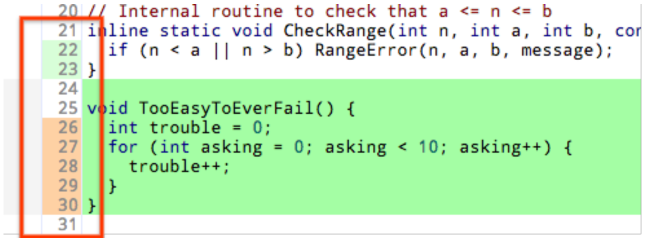
3.3 Coverage Automation

Google's code coverage infrastructure provides further automation on top of Blaze to integrate coverage computation into the regular development workflow. A developer can easily configure automated coverage computation for a project by setting a single boolean option (`enable_coverage=true`) in the project definition.

Once coverage is enabled for a project, an automated system computes project coverage for it once per day. The automated system also computes changelist coverage for all changelists that directly change the code in that project. For each changelist, at least one coverage computation is run the very first time the changelist is sent for review. If the changelist contents change during the review process, at least one more coverage computation is run after the changelist is finally submitted to the codebase.

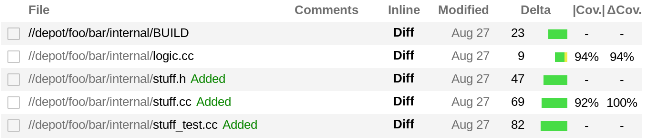
Any developer can request a coverage computation for a changelist at any point with a single push of a button in the code review UI. In practice, the author or reviewer of a changelist request a new computation if its content changes significantly during the review process. If a changelist modifies code in multiple projects, the results are combined.

The coverage automation infrastructure consumes the `lcov`-formatted output of the coverage libraries and individual test runs, and produces a merged coverage report. Instead of storing the individual `lcov` outputs, only two (run-length encoded) sequences are stored for each file—one that encodes whether a line was instrumented and one that encodes whether an instrumented line



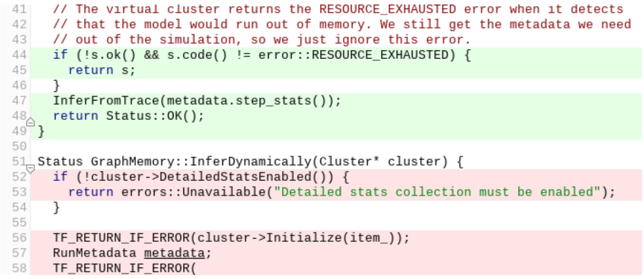
```
// Internal routine to check that a <= n <= b
inline static void CheckRange(int n, int a, int b, cor
  if (n < a || n > b) RangeError(n, a, b, message);
}

void TooEasyToEverFail() {
  int trouble = 0;
  for (int asking = 0; asking < 10; asking++) {
    trouble++;
  }
}
```

Figure 4: Visualizing line coverage during code review. The *line numbers* (highlighted with the red rectangle) are colored to visualize coverage information. A line number is colored (1) green if that line is covered, (2) orange if that line is not covered, and (3) white if that line is not instrumented. Note that the *lines* themselves are also colored. This is, however, not related to coverage but rather to code changes. A line is colored with a different shade of green if it was added in this changelist; it is colored red if it was deleted. Visualizing both coverage and code-change information using colors is tricky. We found that using green and red for both coverage and code changes confused developers—using green and orange for coverage worked much better.



	File	Comments	Inline	Modified	Delta		|Cov.|	ΔCov.
□	//depot/foo/bar/internal/BUILD		**Diff**	Aug 27	23	🟩	-	-
□	//depot/foo/bar/internal/logic.cc		**Diff**	Aug 27	9	🟩	94%	94%
□	//depot/foo/bar/internal/stuff.h Added		**Diff**	Aug 27	47	🟩	-	-
□	//depot/foo/bar/internal/stuff.cc Added		**Diff**	Aug 27	69	🟩	92%	100%
□	//depot/foo/bar/internal/stuff_test.cc Added		**Diff**	Aug 27	82	🟩	-	-

Figure 5: Visualizing coverage aggregates during code review. For each file, |Cov.| gives the overall coverage and Δ Cov. gives the delta coverage (i.e., coverage only for added or modified lines in that file).



```
  // The virtual cluster returns the RESOURCE_EXHAUSTED error when it detects
  // that the model would run out of memory. We still get the metadata we need
  // out of the simulation, so we just ignore this error.
  if (!s.ok() && s.code() != error::RESOURCE_EXHAUSTED) {
    return s;
  }
  InferFromTrace(metadata.step_stats());
  return Status::OK();
}

Status GraphMemory::InferDynamically(Cluster* cluster) {
  if (!cluster->DetailedStatsEnabled()) {
    return errors::Unavailable("Detailed stats collection must be enabled");
  }

  TF_RETURN_IF_ERROR(cluster->Initialize(item_));
  RunMetadata metadata;
  TF_RETURN_IF_ERROR(
```

Figure 6: Visualizing line coverage during code search. A line is colored (1) green if it is covered, (2) red if it is not covered, and (3) white if it is not instrumented. Note that unlike Figure 4, this visualization does not provide any other color coded information, so green and red for coverage works well.

was covered. The raw lcov outputs are then discarded. This compression is an important resource optimization and addresses a major scalability challenge. Any other more verbose format would be prohibitively expensive to store at scale. In the seven years that the two-sequence encoding has been used, it has never limited the usability of the infrastructure; more expressive formats are likely not necessary.

A further technical challenge lies in ensuring successful coverage computations. To improve the success rate, project coverage is computed only on submitted changelists for which the Google Test Automation Platform (TAP [12]) already determined that all tests pass. For changelist coverage, the infrastructure waits a preconfigured amount of time (currently 10 minutes) for TAP to determine if any test fails, in which case coverage will not be computed. While this avoids futile attempts to compute code coverage in the presence of failing tests, it does not guarantee that the coverage computation succeeds. Executing tests with coverage instrumentation enabled requires more resources and changes the execution environment. Addressing these differences to further improve the success rate required hundreds of specific fixes that target individual corner cases within the project code or configuration.

3.4 Visualization and Data Analytics

Google's code coverage infrastructure visualizes coverage information in several different ways: Changelist coverage is visualized in Critique, Google's code review system. Project coverage is visualized in CodeSearch [1] and in the developers' IDEs. Finally, the infrastructure provides a trend visualization for project coverage and the coverage data can be queried for data analytics.

Visualization In Critique, changelist coverage is shown for each project that is affected by the change and reported in a summary. Line coverage is surfaced visually for each line, as shown in figure 4, and aggregated coverage is reported numerically for each file and for the whole changelist, as shown in figure 5. Branch coverage is provided for languages for which the underlying libraries support it, visually in the code and as an aggregate.

An internal version of CodeSearch [1] helps Google developers navigate the repository, which contains over 1 billion lines of code. CodeSearch makes it easy to navigate files, cross-reference symbols, and see the version history. It is also convenient for rendering line coverage, as shown in Figure 6. The underlying data comes from the daily project coverage computations and is visualized only for files that have not changed since the last computation.

Code editors and IDEs are another natural choice for visualizing code coverage. Using the same data source as for CodeSearch, we provide custom support for many editors. For example, our metaplugin for Vim is open sourced [8].

Because line numbers are present in almost all tools that deal with code, line coverage is a good choice for visualization. Although the coverage integration is conceptually the same for all programming languages, there are nuanced differences between programming languages. For example, whether the opening curly brace of a block counts towards line coverage varies between languages, sometimes even within a single language based on the code before the brace. However, in our experience these conceptual differences do not seem to be important: developers interpret the visualization in context and simply ignore such minor idiosyncrasies.

Data Analytics In addition to visualizing changelist and project coverage, the coverage infrastructure supports trend visualization and data analysis. For example, a developer can track project coverage over a longer period of time, using a pre-built trend visualization UI. Likewise, a developer can issue specific queries, using the Dremel [22] query system. All coverage data is stored in a central database for interactive analysis. This is mostly used by developers or researchers, who are analyzing large batches of coverage data, similar to the numbers reported in this paper.

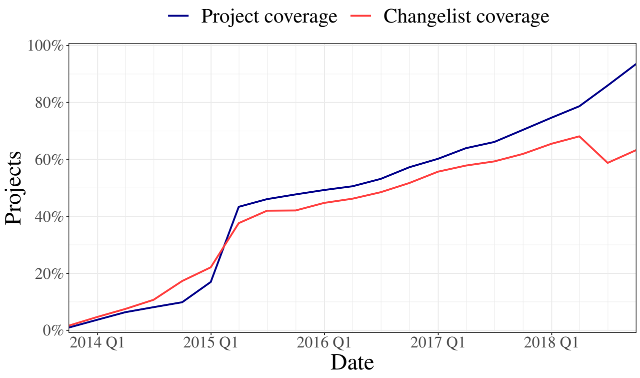


Figure 7: Projects actively using coverage automation. If a project that has coverage automation enabled does not actively use it (e.g., no daily project coverage computation is successful), it does not contribute towards the active count.

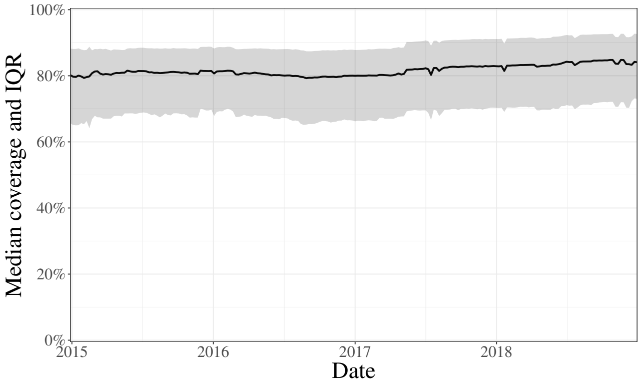


Figure 8: Median weekly project coverage. The weekly project coverage is the median of the daily project coverages in a given week. If a project had no successful coverage computation in a given week, it is excluded from the aggregation for that week. Note that we exclude data older than 2015 because fewer than 20% of projects were using coverage before 2015 (see Figure 7).

4 COVERAGE ADOPTION AND USEFULNESS

This section reports on the results of analyzing multi-year, historical data and surveying 3000 developers at Google. Specifically, this section shows (1) how Google's developers adopted and integrated code coverage into their workflow, (2) how code coverage ratios evolved over time at Google, (3) and what the perceived usefulness of code coverage is.

Code coverage has been used at Google for at least 13 years. To the best of our knowledge, the oldest internally recorded plans for an automated code coverage infrastructure date back to 2006.

The infrastructure described in Section 3, which is the basis for the reported results, is the latest version and has been in continuous use since 2012. Between 2012 and 2018, it collected approximately 13,000,000 daily project coverage measurements and 14,000,000 changelist coverage measurements.

4.1 Adoption of Code Coverage

Code coverage computation is not mandatory across Google. However, projects (or groups of projects) can choose to mandate the use of code coverage in their teams. Further, individual teams can opt into automated code coverage computation, and an individual developer can interactively invoke a code coverage computation.

Recall Table 1, which shows the percentage of developers manually invoking `blaze coverage` interactively. A typical use case for this is to compute code coverage before a changelist is created. Only 0.5% of developers manually invoke code coverage computation on a daily basis. In contrast, 33.5% of developers invoke `blaze test` (i.e., running tests without coverage computation) on the command line at least once on any given day. This shows that developers treat coverage and binary pass/fail information differently in their workflow, in particular they check coverage with much lower frequency.

For every changelist, code coverage is computed automatically. Google's code review tool surfaces the results to the benefit of the changelist author and reviewers. Reviewers can use code coverage to ask the author to improve the tests for the change, or authors can do so proactively.

Figure 7 shows the percentage[1] of projects that actively use Google's Test Automation Platform and actively measure project and changelist coverage. The graph shows all data since the introduction of the code coverage automation. The spike in the first quarter of 2015 is a result of an advertisement on Google's famous internal "Testing on the Toilet" [16]. We conjecture that if the infrastructure had been better at that time, the slope of the adoption curve would have continued to be steep after the first quarter.

February 2015 until now is a period of progressively refining and improving the coverage infrastructure to allow more projects to use it. Many projects that were not using code coverage expressed interest, but their code base was not supported by the infrastructure (e.g., coverage instrumentation causing test failures). From our experience (as owners and maintainers of the coverage infrastructure), an unsupported project usually has only a few test failures caused by the coverage instrumentation, but each project has a different root cause. Section 4.3 provides more details.

We do not expect changelist coverage usage to reach 100% for technical reasons: As a resource optimization, the coverage infrastructure only computes changelist coverage if all tests for that changelist pass. This means that some particularly passive projects might not get a successful changelist coverage computation at all in a given time period. Therefore, while the maximum is theoretically 100%, a realistic maximum is likely lower. In contrast, project coverage usage is more likely to eventually reach 100%. Note that the drop in changelist coverage usage in the second quarter of 2018 is not currently understood.

4.2 Code Coverage Ratios

Figure 8 shows the median project coverage as well as the interquartile range (IQR) for all projects for the time period between 2015 and 2018. The chart shows the aggregated weekly project coverage

[1] We give percentages because the absolute number of projects varies greatly over the period of time, and it is a confidential value.

Table 2: Coverage levels and corresponding thresholds. Many projects voluntarily set these thresholds as their goal.

Level	Threshold
Level 1	Coverage automation disabled
Level 2	Coverage automation enabled
Level 3	Project coverage at least 60%; Changelist coverage at least 70%
Level 4	Project coverage at least 75%; Changelist coverage at least 80%
Level 5	Project coverage at least 90%; Changelist coverage at least 90%

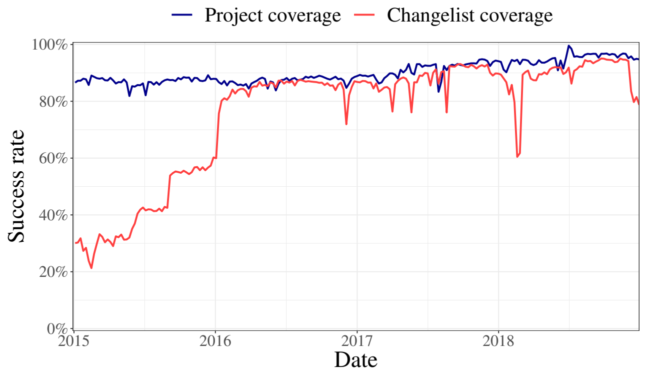


Figure 9: Coverage infrastructure success rates. The success rate indicates the percentage of all automated coverage computations that successfully completed per week. Note that we exclude data older than 2015 because fewer than 20% of projects were using coverage before 2015 (see figure 7).

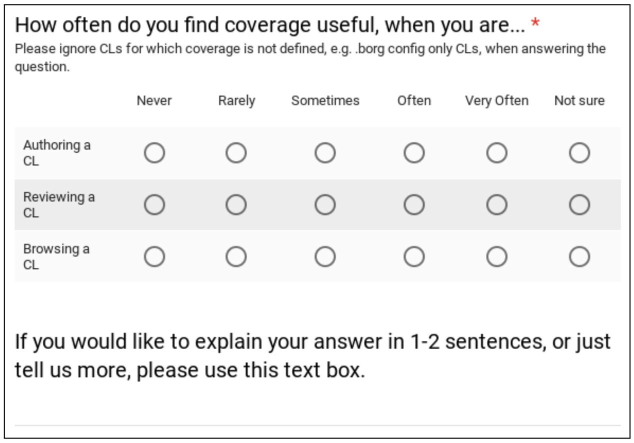
(i.e., the median daily project coverage for a given week). Reporting on weekly project coverage smoothes out outliers caused by intermittent test or infrastructure failures.

Google does not enforce any code coverage thresholds across the entire codebase. Projects (or groups of projects) are free to define their own thresholds and goals. Many projects opt-into a centralized voluntary alerting system that defines five levels of code coverage thresholds. Table 2 shows the criteria for each level. We suspect that the gradual increase in project coverage since mid 2016 reflects a gradual adoption of these levels, which we publicized in 2016.

Safety-critical systems, of course, have stricter requirements. To ensure high quality, these projects use, e.g., mutation testing [23, 24], which provides a more rigorous adequacy criterion [19].

4.3 Reasons for Failed Code Coverage Computations

The coverage infrastructure computes project coverage once per day and changelist coverage for each changelist (Section 3.3 gives more details). Figure 9 shows the success rates of the coverage infrastructure for both project and changelist coverage computations. This section describes the most common reasons for failed coverage computations.



How often do you find coverage useful, when you are... *

Please ignore CLs for which coverage is not defined, e.g. .borg config only CLs, when answering the question.

	Never	Rarely	Sometimes	Often	Very Often	Not sure
Authoring a CL	○	○	○	○	○	○
Reviewing a CL	○	○	○	○	○	○
Browsing a CL	○	○	○	○	○	○

If you would like to explain your answer in 1-2 sentences, or just tell us more, please use this text box.

Figure 10: Screenshot of the coverage usefulness survey.

A common reason for a failed coverage computation is a test failure that only manifests when coverage instrumentation of the tested code is enabled. Coverage instrumentation is relatively expensive, and performance failures are the most common cause of failed coverage computations. In most programming languages, coverage instrumentation prevents at least some compiler optimizations. This results in longer test run times, which leads to more timeouts. Coverage instrumentation also results in larger binaries, which leads to more out-of-memory errors. The second most common cause is test flakiness. Coverage instrumentation can worsen flakiness of non-deterministic tests. A simple example would be any test that uses a `sleep` command to allow the code under test to complete it's work can suddenly become flaky when instrumented simply because the code is slower. The progressive increase in the project coverage success rate is a result of a long series of infrastructure improvements to counter failed coverage computations.

Note that the changelist coverage computation was initially implemented along side TAP, which explains the lower success rate. The current coverage infrastructure awaits TAP's results to avoid attempting coverage computation in the presence of failing tests. The noticeable, sharp drops in Figure 9 for the changelist coverage success rate are coverage infrastructure failures, usually resolved within a week. These failures show the need for maintaining the coverage infrastructure, and resources should be provisioned for such maintenance.

4.4 Perceived Usefulness of Code Coverage

Since the coverage computation and visualization is completely automated and integrated into the developer workflow, we wished to understand to what extent developers consume the provided data and what the perceived usefulness of code coverage is. To that end, we conducted a survey in February 2019, which consisted of three Likert-scale questions and one open-ended question. Figure 10 shows a screenshot of the survey.

The survey asked the following question: "How often do you find coverage useful, when you are..." with three scenarios (Authoring,

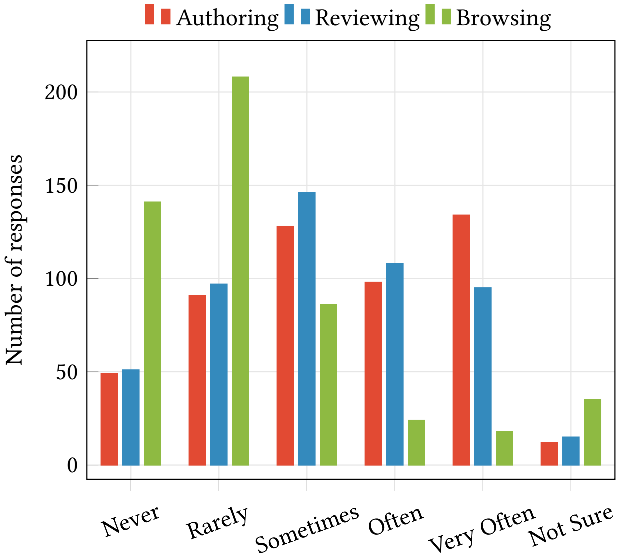


Figure 11: Self-reported usefulness of changelist coverage The survey was sent to 3000 contributors to Google's codebase out of which 512 (17%) responded. Each participant was asked to rate the usefulness in the 3 most common use cases: when authoring a code change, when reviewing a code change and when browsing a code change (i.e. neither reviewing nor authoring).

Table 3: Sentiment towards coverage expressed in the write-in text box in the changelist coverage usefulness survey.

Sentiment	Responses	Percentage
Positive	173	60%
Neutral	67	24%
Negative	46	16%
Total	286	100%

Reviewing, and Browsing a CL). Each scenario used the same five-point Likert scale. We sent the survey to 3000 randomly chosen developers at Google out of which 512 responded (17%). We randomly selected participants from the set of all people who have committed at least one line to Google's codebase. While this set contains mostly developers, it also includes people in non-engineering roles.

Figure 11 shows the self-reported usefulness of changelist coverage for the three different scenarios. When authoring a changelist, 45% of respondents reported that they use coverage "Often" or "Very Often" and 25% use it "Sometimes". When reviewing a changelist, 40% respondents use coverage "Often" or "Very Often" and 28% use it "Sometimes". 10% of respondents never use coverage. Overall, a substantial number of developers do use code coverage on a regular basis and find value in it.

Anecdotal evidence from the open-ended text box provides additional insights. A small number of respondents were not developers but still contribute changes to the codebase, e.g. technical writers. Such respondents chose "Never" or "Not sure". On the opposite side of the scale, some respondents chose to explain the "Very Often" response with statements such as "I use coverage numbers when authoring to make sure I have remembered tests. I find it more important when reviewing as it is hard to see if edge cases are covered quickly without coverage".

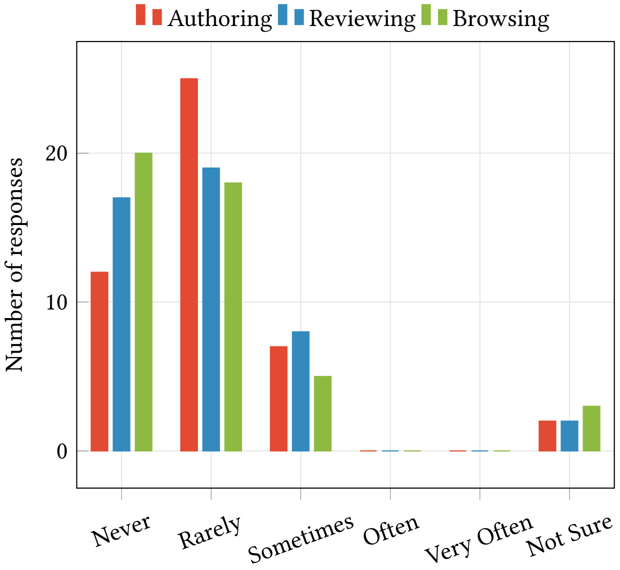


Figure 12: Self-reported usefulness of changelist coverage by the 46 survey participants who expressed generally negative sentiment towards code coverage.

We analyzed the sentiment of all open ended text box answers and categorized them into three categories:

- Positive: Answers that were generally positive towards the concept of code coverage, even if they raised some problems with the infrastructure.
- Negative: Answers that were generally negative towards the concept of code coverage.
- Neutral: Answers that did not convey the general attitude towards coverage. These were mostly reporting problems with the infrastructure or comments on the survey itself, e.g. "I suspect I'm too senior for the purposes of your survey. I haven't written or seriously reviewed code in >12 months."

Table 3 shows the distribution of the answers in these three categories. 60% of responses were generally positive towards coverage, 16% were negative, 24% were neutral. One interesting observation is that the percentage of respondents who expressed negative sentiment about code coverage is larger than the percentage of projects not using code coverage. Figure 12 shows that some respondents who expressed generally negative sentiment towards coverage, still self-reported that they use it "Sometimes" or "Rarely".

5 RELATED WORK

Code coverage has been known since at least the 1960s. Elmendorf [13] gives a reasonably robust strategy for using branch coverage in testing operating systems in 1969. Piwowarski, Ohba and Caruso [25] mention that coverage measurements were performed within IBM in late 1960.

Yang et al. [26] conducted a survey of 17 different coverage tools. The survey provides a good overview of the area and contrasts the tools based on features. It does, however, not analyze how to best integrate different tools in a coherent workflow.

Piwowarski et al. [25] describe a large survey of IBM test organizations and their use of coverage in an industrial context. They found that ease of use was the primary factor in coverage measurement adoption, and that once coverage measurement was adopted, it led to increase in test suite quality. Woo Kim [20] describes using coverage to test a 19.8 kLOC product in an industrial setting. They conclude that code coverage is useful, but that detailed code coverage analysis is not cost effective, because most defects are localized in a relatively small percentage of error-prone modules, and because there is strong correlation between complex and simple coverage criteria. While these industry reports are very valuable, they often report results based on limited code base size, most importantly they tend to analyze only one or two programming languages.

Adler et al. at IBM proposed substring hole analysis to analyze code coverage data which makes it cost-effective for large systems tests [9]. This approach enables reasoning about large amounts of coverage information intuitively. To avoid problems inherent in capturing the system test coverage, Chen et al. proposed an automated approach to estimate code coverage with high accuracy using execution logs [10].

Li et al. studied existing Java branch coverage tools, finding that none measure all the branching structures correctly and that bytecode instrumentation, which is used my most tools, is not a valid approach to measure branch coverage on source code [21].

6 CONCLUSIONS AND FUTURE WORK

This paper describes Google's code coverage infrastructure, how the computed code coverage information is visualized, and how it is integrated into the developer workflow. The code coverage infrastructure is designed to overcome technical challenges such as scale and programming language diversity, but our experience also shows that a further important technical challenge in practice is dealing with failed coverage computations. A considerable multi-year effort was required to sufficiently address these issues at Google. Usage data and a survey of the developers suggest that developers are generally positive towards the idea of code coverage. They view it as a valuable addition to their daily workflows and use it, if it is available. The key points to achieve this positive sentiment are usability and low overhead from the developer point of view.

We hope that the ideas and experiences described in this paper motivate and help others to develop similar infrastructure. Based on the lessons reported in this paper, we recommend the following:

- Measure coverage automatically at critical points in the development workflow.
- Display coverage information within the tools developers commonly use.
- Expect to invest effort in dealing with errors caused by coverage instrumentation.
- Rely on existing, well established libraries for computing coverage; these are powerful enough for daily use.
- Integrate coverage computation for all programming languages used under a single, uniform interface.

We plan to further investigate usage data and developer opinions in order to better understand how coverage is used and how it could be better used to the benefit of developers. In particular, we will broaden the scope of our survey and ask for more self-reported data related to coverage, and will correlate this data with objective behavioral data. Ultimately, it would be nice to determine whether adoption of code coverage leads to better software quality, but there is is no single, reliable proxy measurement of code quality. For example, the number of changelists may be influenced by code quality, but it might just as well be a result of general development or programmer habits. Furthermore, code quality is not the only benefit that should arise from using code coverage. Therefore, some open questions we would like to address in future work are:

- Does the *perceived* usefulness differ from *actual* usefulness? For example, does showing coverage during code review actually speed up the review process?
- Does the perceived and actual usefulness depend on the size of the changelist? For example, we conjecture that when authoring full features, like a new class accompanied by a test suite, developers care more about code coverage than when authoring a very small change.
- Does showing coverage during code review have an effect on the final code coverage ratio—that is, will tests be added more often if coverage is shown than when it is not?

ACKNOWLEDGMENTS

We would like to thank Matt Staats, Roger Fleig, Jeff Cox and Mikhail Rybalkin for their comments and suggestions.

REFERENCES

[1] Codesearch. https://cs.chromium.org/.

[2] Coverage.py. https://coverage.readthedocs.io/en/v4.5.x/.

[3] Dart - Coverage. https://github.com/dart-lang/coverage.

[4] Istanbul Code Coverage. https://github.com/istanbuljs.

[5] JaCoCo Java Code Coverage Library. https://www.eclemma.org/jacoco/.

[6] The Go Blog - The cover story. https://blog.golang.org/cover.

[7] Using the GNU Compiler Collection (GCC): Gcov. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[8] vim-coverage. https://github.com/google/vim-coverage.

[9] Adler, Y., Behar, N., Raz, O., Shehory, O., Steindler, N., Ur, S., and Zlotnick, A. Code coverage analysis in practice for large systems. In *2011 33rd International Conference on Software Engineering (ICSE)* (2011), IEEE, pp. 736–745.

[10] Chen, B., Song, J., Xu, P., Hu, X., and Jiang, Z. M. J. An automated approach to estimating code coverage measures via execution logs. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (New York, NY, USA, 2018), ASE 2018, ACM, pp. 305–316.

[11] Chilenski, J. J., and Miller, S. P. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal 9*, 5 (September 1994), 193–200.

[12] Elbaum, S., Rothermel, G., and Penix, J. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2014), FSE 2014, ACM, pp. 235–245.

[13] Elmendorf, W. R. Controlling the functional testing of an operating system. *IEEE Transactions on Systems Science and Cybernetics 5*, 4 (1969), 284–290.

[14] 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.

[15] Inc., G. Google C++ Style Guide. https://google.github.io/styleguide/cppguide.html.

[16] Inc., G. Introducing "Testing on the Toilet". https://testing.googleblog.com/2007/01/introducing-testing-on-toilet.html, Jan. 2007.

[17] Inc., G. Bazel build system. https://bazel.io/, 2015.

[18] 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.

[19] Just, R., Jalali, D., Inozemtseva, L., Ernst, M. D., Holmes, R., and Fraser, G. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)* (Nov. 2014), pp. 654–665.

[20] Kim, Y. W. Efficient use of code coverage in large-scale software development. In *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research* (2003), IBM Press, pp. 145–155.

[21] Li, N., Meng, X., Offutt, J., and Deng, L. Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools (experience report). In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)* (2013), IEEE, pp. 380–389.

[22] Melnik, S., Gubarev, A., Long, J. J., Romer, G., Shivakumar, S., Tolton, M., and Vassilakis, T. Dremel: Interactive analysis of web-scale datasets. In *Proc. of the 36th Int'l Conf on Very Large Data Bases* (2010), pp. 330–339.

[23] Petrović, G., and Ivanković, M. State of mutation testing at Google. In *Proceedings of the International Conference on Software Engineering—Software Engineering in Practice (ICSE SEIP)* (May 2018).

[24] Petrović, G., Ivanković, M., Kurtz, B., Ammann, P., and Just, R. An industrial application of mutation testing: Lessons, challenges, and research directions. In *Proceedings of the International Workshop on Mutation Analysis (Mutation)* (Apr. 2018), pp. 47–53.

[25] 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.

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

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