Modern Code Review: A Case Study at Google

Caitlin Sadowski, Emma Söderberg,
Luke Church, Michal Sipko
Google, Inc.
{supertri,emso,lukechurch,sipkom}@google.com

Alberto Bacchelli
University of Zurich
bacchelli@ifi.uzh.ch

ABSTRACT

Employing lightweight, tool-based code review of code changes (aka *modern code review*) has become the norm for a wide variety of open-source and industrial systems. In this paper, we make an exploratory investigation of modern code review at Google. Google introduced code review early on and evolved it over the years; our study sheds light on why Google introduced this practice and analyzes its current status, after the process has been refined through decades of code changes and millions of code reviews. By means of 12 interviews, a survey with 44 respondents, and the analysis of review logs for 9 million reviewed changes, we investigate motivations behind code review at Google, current practices, and developers' satisfaction and challenges.

CCS CONCEPTS

- **Software and its engineering → Software maintenance tools**;

ACM Reference format:
Caitlin Sadowski, Emma Söderberg,
Luke Church, Michal Sipko and Alberto Bacchelli. 2018. Modern Code Review: A Case Study at Google. In *Proceedings of 40th International Conference on Software Engineering: Software Engineering in Practice Track, Gothenburg, Sweden, May 27-June 3, 2018 (ICSE-SEIP '18),* 10 pages.
DOI: 10.1145/3183519.3183525

1 INTRODUCTION

Peer code review, a manual inspection of source code by developers other than the author, is recognized as a valuable tool for improving the quality of software projects [2, 3]. In 1976, Fagan formalized a highly structured process for code reviewing—code inspections [16]. Over the years, researchers provided evidence on the benefits of code inspection, especially for defect finding, but the cumbersome, time-consuming, and synchronous nature of this approach hindered its universal adoption in practice [37]. Nowadays, most organizations adopt more lightweight code review practices to limit the inefficiencies of inspections [33]. Modern code review is (1) informal (in contrast to Fagan-style), (2) tool-based [32], (3) asynchronous, and (4) focused on reviewing code changes.

An open research challenge is understanding which practices represent valuable and effective methods of review in this novel context. Rigby and Bird quantitatively analyzed code review data from software projects spanning varying domains as well as organizations and found five strongly convergent aspects [33], which they conjectured can be prescriptive to other projects. The analysis of Rigby and Bird is based on the value of a *broad* perspective (that analyzes multiple projects from different contexts). For the development of an empirical body of knowledge, championed by Basili [7], it is essential to also consider a *focused and longitudinal* perspective that analyzes a single case. This paper expands on work by Rigby and Bird to focus on the review practices and characteristics established at Google, *i.e.*, a company with a multi-decade history of code review and a high-volume of daily reviews to learn from. This paper can be (1) prescriptive to practitioners performing code review and (2) compelling for researchers who want to understand and support this novel process.

Code review has been a required part of software development at Google since very early on in the company's history; because it was introduced so early on, it has become a core part of Google culture. The process and tooling for code review at Google have been iteratively refined for more than a decade and is applied by more than 25,000 developers making more than 20,000 source code changes each workday, in dozens of offices around the world [30].

We conduct our analysis in the form of an exploratory investigation focusing on three aspects of code review, in line with and expanding on the work by Rigby and Bird [33]: (1) The motivations driving code review, (2) the current practices, and (3) the perception of developers on code review, focusing on challenges encountered with a specific review (*breakdowns* in the review process) and satisfaction. Our research method combines input from multiple data sources: 12 semi-structured interviews with Google developers, an internal survey sent to engineers who recently sent changes to review with 44 responses, and log data from Google's code review tool pertaining to 9 million reviews over two years.

We find that the process at Google is markedly lighter weight than in other contexts, based on a single reviewer, quick iterations, small changes, and a tight integration with the code review tool. Breakdowns still exist, however, due to the complexity of the interactions that occur around code review. Nevertheless, developers consider this process valuable, confirm that it works well at scale, and conduct it for several reasons that also depend on the relationship between author and reviewers. Finally, we find evidence on the use of the code review tool beyond collaborative review and corroboration for the importance of code review as an educational tool.

2 BACKGROUND AND RELATED WORK

We describe the review processes investigated in literature, then we detail convergent code review practices across these processes [33].

2.1 Code Review Processes and Contexts

Code Inspections. Software inspections are one of the first formalized processes for code review. This highly structured process involves planning, overview, preparation, inspection meeting, reworking, and follow-up [16]. The goal of code inspections is to find defects during a synchronized inspection meeting, with authors and reviewers sitting in the same room to examine code changes. Kollanus and Koskinen compiled the most recent literature survey on code inspection research [25] . They found that the vast majority of studies on code inspections are empirical in nature. There is a consensus about the overall value of code inspection as a defect finding technique and the value of reading techniques to engage the inspectors. Overall, research on code inspection has declined from 2005, in line with the spread of the internet and the growth of asynchronous code review processes.

Asynchronous review via email. Until the late 2000s, most large OSS projects adopted a form of remote, asynchronous reviews, relying on patches sent to communication channels such as mailing lists and issue tracking systems. In this setting, project members evaluate contributed patches and ask for modifications through these channels. When a patch is deemed of high enough quality, core developers commit it to the codebase. Trusted committers may have a commit-then-review process instead of doing pre-commit reviews [33]. Rigby *et al.* were among the first to do extensive work in this setting; they found that this type of review "*has little in common* [with code inspections] *beyond a belief that peers will effectively find software defects*" [34]. Kononenko *et al.* analyzed the same setting and find that review response time and acceptance are related to social factors, such as reviewer load and change author experience [26], which were not present in code inspections.

Tool-based review. To bring structure to the process of reviewing patches, several tools emerged in OSS and industrial settings. These tools support the *logistics* of the review process: (1) The author of a patch submits it to the code review tool, (2) the reviewers can see the diff of the proposed code change and (3) can start a threaded discussion on specific lines with the author and other reviewers, and then (4) the author can propose modifications to address reviewers' comments. This feedback cycle continues until everybody is satisfied or the patch is discarded. Different projects adapted their tools to support their process. Microsoft uses **CodeFlow**, which tracks the state of each person (author or reviewer) and where they are in the process (signed off, waiting, reviewing); CodeFlow does not prevent authors from submitting changes without approval [33] and supports chats in comment threads [4]. Google's Chromium project (along with several other OSS projects) relies on the externally-available **Gerrit** [17]; in Chromium, changes are only merged into the master branch after explicit approval from reviewers and automated verification that the change does not break the build [12]. In Gerrit, unassigned reviewers can also make comments. VMware developed the open-source **ReviewBoard**, which integrates static analysis into the review process; this integration relies on change authors manually requesting analysis and has been shown to improve code review quality [5]. Facebook's code review system, **Phabricator** [29], allows reviewers to "take over" a change and commit it themselves and provides hooks for automatic static analysis or continuous build/test integration.

In the context of tool-based reviews, researchers have investigated the relationship between code change acceptance or response time and features of the changed code and authors [41], as well as the agreement among reviewers [22]. Qualitative investigations have been also conducted to define what constitutes a good code review according to industrial [11] and OSS developers [26].

Pull-based development model. In the GitHub pull request process [18] a developer wanting to make a change forks an existing git repository and then makes changes in their fork. After a pull request has been sent out, it appears in the list of pull requests for the project in question, visible to anyone who can see the project. Gousios *et al.* qualitatively investigated work practices and challenges of pull-request integrators [21] and contributors [20], finding analogies to other tool-based code reviews.

Table 1: Convergent review practices [33].

id	Convergent Practice
CP_1	Contemporary peer review follows a lightweight, flexible process
CP_2	Reviews happen early (before a change is committed), quickly, and frequently
CP_3	Change sizes are small
CP_4	Two reviewers find an optimal number of defects
CP_5	Review has changed from a defect finding activity to a group problem solving activity

2.2 Convergent Practices in Code Review

Rigby and Bird presented the first and most significant work that tries to identify convergent practices across several code review processes and contexts [33]. They considered OSS projects that use email based reviews, OSS projects that use Gerrit, an AMD project that uses a basic code review tool, and Microsoft with CodeFlow. They analyzed the process and the data available from these projects to describe several angles, such as iterative development, reviewer selection, and review discussions. They identified five practices of modern code review to which all the considered projects converged (Table 1). We will refer to these practices using their id, *e.g.*, CP_1. Substantially, they found an agreement in terms of a quick, lightweight process (CP_1,CP_2,CP_3) with few people involved (CP_4) who conduct group problem solving (CP_5).

3 METHODOLOGY

This section describes our research questions and settings; it also outlines our research method and its limitations.

3.1 Research Questions

The overall goal of this research is to investigate modern code review at Google, which is a process that involves thousands of developers and that has been refined over more than a decade. To this aim, we conduct an exploratory investigation that we structure around three main research questions.

RQ$_1$: What are the motivations for code review at Google? Rigby and Bird found the motivations to be one of the converging traits of modern code review (CP$_5$). Here we study what motivations and expectations drive code reviews at Google. In particular, we consider both the historical reasons for introducing modern code review (since Google is one of the first companies that used modern code review) and the current expectations.

RQ$_2$: What is the practice of code review at Google? The other four convergent practices found by Rigby and Bird regard how the process itself is executed, in terms of flow (CP$_1$), speed and frequency (CP$_2$), size of the analyzed changes (CP$_3$), and number of reviewers (CP$_4$). We analyze these aspects at Google to investigate whether the same findings hold for a company that has a longer history of code review, an explicit culture, and larger volume of reviews compared to those analyzed in previous studies [4, 33].

RQ$_3$: How do Google developers perceive code review? Finally, in our last research question, we are interested in understanding how Google developers perceive modern code review as implemented in their company. This exploration is needed to better understand practices (since perceptions drive behavior [39]) and to guide future research. We focus on two aspects: *breakdowns* of the review process developers experience during specific reviews and whether developers are *satisfied* with review despite these challenges.

3.2 Research Setting

We briefly describe our research setting for context on our methodology. See Section 5.1 for a comprehensive description of Google's code review process and tooling.

Most software development at Google occurs in a monolithic source repository, accessed via an internal version control system [30]. Since code review is required at Google, every commit to Google's source control system goes first through code review using Critique: an internally developed, centralized, web-based code review tool. The development workflow in Google's monolithic repository, including the code review process, is very uniform. As with other tools described in Section 2, Critique allows reviewers to see a diff of the proposed code change and start a threaded discussion on specific lines with the author and other reviewers. Critique offers extensive logging functionalities; all developer interactions with the tool are captured (including opening the tool, viewing a diff, making comments, and approving changes).

3.3 Research Method

To answer our research questions, we follow a mixed qualitative and quantitative approach [13], which combines data from several sources: semi-structured interviews with employees involved in software development at Google, logs from the code review tool, and a survey to other employees. We use the interviews as a tool to collect data on the diversity (as opposed to frequencies [23]) of the motivations for conducting code review (RQ$_1$) and to elicit developers' perceptions of code review and its challenges (RQ$_3$). We use Critique logs to quantify and describe the current review practices (RQ$_2$). Finally, we use the survey to confirm the diverse motivations for code review that emerged from the interviews (RQ$_1$) and elicit the developers' satisfaction with the process.

Interviews. We conducted a series of face-to-face semi-structured [13] interviews with selected Google employees, each taking approximately 1 hour.The initial pool of possible participants was selected using snowball sampling, starting with developers known to the paper authors. From this pool, participants were selected to ensure a spread of teams, technical areas, job roles, length of time within the company, and role in the code review process. The interview script (in appendix [1]) included questions about perceived motivations for code review, a recently reviewed/authored change, and best/worst review experiences. Before each interview we reviewed the participant's review history and located a change to discuss in the interview; we selected these changes based on the number of interactions, the number of people involved in the conversation and whether there were comments that seemed surprising. In an observation part of the interview, the participant was asked to think-aloud while reviewing the pending change and to provide some explicit information, such as the entry point for starting the review. The interviews continued until saturation [19] was achieved and interviews were bringing up broadly similar concepts. Overall, we conducted 12 interviews with staff who had been working at Google from 1 month to 10 years (median 5 years), in Software Engineering and Site Reliability Engineering. They included technical leads, managers and individual contributors. Each interview involved three to four people: The participant and 2-3 interviewees (two of which are authors of this paper). Interviews were live-transcribed by one of the interviewers while another one asked questions.

Open Coding on Interview Data. To identify the broad themes emerging from the interview data we performed an open coding pass [27]. Interview transcripts were discussed by two authors to establish common themes, then converted into a coding scheme. An additional author then performed a closed coding of the notes of the discussions to validate the themes. We iterated this process over one of the interviews until we had agreement on the scheme. We also tracked in what context (relationship between reviewer and author) these themes were mentioned. The combination of the design of the questions and the analysis process means that we can discuss stable themes in the results, but cannot meaningfully discuss relative frequencies of occurrence [23].

Analysis of Review Data. We analyzed quantitative data about the code review process by using the logs produced by Critique. We mainly focus on metrics related to convergent practices found by Rigby and Bird [33]. To allow for comparison, we do not consider changes without any reviewers, as we are interested in changes that have gone through an explicit code review process. We consider a "reviewer" to be any user who approves a code change, regardless of whether they were explicitly asked for a review by the change author. We use a name-based heuristic to filter out changes made by automated processes. We focus exclusively on changes that occur in the main codebase at Google. We also exclude changes not yet committed at the time of study and those for which our diff tool reports a delta of zero source lines changed, *e.g.*, a change that only modifies binary files.
On an average workday at Google, about 20,000 changes are committed that meet the filter criteria described above. Our final dataset includes the approximately 9 million changes created by more than 25,000 authors and reviewers from January 2014 until July 2016 that meet these criteria, and about 13 million comments collected from all changes between September 2014 and July 2016.

Survey. We created an online questionnaire that we sent to 98 engineers who recently submitted a code change. The code change had already been reviewed, so we customized the questionnaire to ask respondents about how they perceived the code review for *their specific recent change*; this strategy allowed us to mitigate recall bias [35], yet collect comprehensive data. The survey consisted of three Likert scale questions on the value of the received review, one multiple choice on the effects of the review on their change (based on the expectations that emerged from the interviews) with an optional 'other' response, and one open-ended question eliciting the respondent's opinion on the received review, the code review tool, and/or the process in general. We received 44 valid responses to our survey (45% response rate, which is considered high for software engineering research [31]).

3.4 Threats to Validity and Limitations

We describe threats to validity and limitations of the results of our work, as posed by our research method, and the actions that we took to mitigate them.

Internal validity - Credibility. Concerning the quantitative analysis of review data, we use heuristics to filter out robot-authored changes from our quantitative analysis, but these heuristics may allow some robot authored changes; we mitigated this as we only include robot-authored changes that have a human reviewer. Concerning the qualitative investigation, we used open coding to analyze the interviewees' answers. This coding could have been influenced by the experience and motivations of the authors conducting it, although we tried to mitigate this bias by involving multiple coders. The employees that decided to participate in our interviews and the survey freely decided to do so, thus introducing the risk of *self-selection bias*. For this reason, results may have been different for developers who would not choose to participate; to try to mitigate this issue, we combine information from both interviews and survey. Moreover, we used a snowball sampling method to identify engineers to interview, this is at the risk of *sampling bias*. Although we attempted to mitigate this risk by interviewing developers with a range of job roles and responsibilities, there may be other factors the developers we interviewed share that would not apply across the company. To mitigate *moderator acceptance bias*, the researchers involved in the qualitative data collection were not part of the Critique team. Social desirability bias may have influenced the answers to align more favorably to Google culture; however, at Google people are encouraged to criticize and improve broken workflows when discovered, thus reducing this bias. Finally, we did not interview research scientists or developers that interact with specialist reviewers (such as security reviews), thus our results are biased towards general developers.

Generalizability - Transferability. We designed our study with the stated aim of understanding modern code review within a specific company. For this reason, our results may not generalize to other contexts, rather we are interested in the diversity of the practices and breakdowns that are still occurring after years and millions of reviews of refinement. Given the similarity of the underlying code review mechanism across several companies and OSS projects, it is reasonable to think that should a review process reach the same level of maturity and use comparable tooling, developers would have similar experiences.

4 RESULTS: MOTIVATIONS

In our first research question, we seek to understand motivations and expectations of developers when conducting code review at Google, starting by investigating what led to the introduction of this process in the first place.

4.1 How it All Started

Code review at Google was introduced early on by one of the first employees; the first author of this paper interviewed this employee (referred to as E in the following) to better understand the initial motivation of code review and its evolution. E explained that the main impetus behind the introduction of code review was *to force developers to write code that other developers could understand*; this was deemed important since code must act as a teacher for future developers. E stated that the introduction of code review at Google signaled the transition from a research codebase, which is optimized towards quick prototyping, to a production codebase, where it is critical to think about future engineers reading source code. Code review was also perceived as capable of ensuring that more than one person would be familiar with each piece of code, thus increasing the chances of knowledge staying within the company.

E reiterated on the concept that, although it is great if reviewers find bugs, the foremost reason for introducing code review at Google was to improve code understandability and maintainability. However, in addition to the initial educational motivation for code review, E explained that three additional benefits soon became clear to developers internally: checking the consistency of style and design; ensuring adequate tests; and improving security by making sure no single developer can commit arbitrary code without oversight.

4.2 Current expectations

By coding our interview data, we identified four key themes for what Google developers expect from code reviews: **education**, **maintaining norms**, **gatekeeping**, and **accident prevention**. Education regards either teaching or learning from a code review and is in line with the initial reasons for introducing code review; norms refer to an organization preference for a discretionary choice (*e.g.*, formatting or API usage patterns); gatekeeping concerns the establishment and maintenance of boundaries around source code, design choices or another artifact; and accidents refer to the introduction of bugs, defects or other quality related issues.

These are the main themes *during* the review process, but code review is also used *retrospectively* for **tracking history**. Developers value the review process after it has taken place; code review enables the ability to browse the history of a code change, including what comments occurred and how the change evolved. We also noticed developers using code review history to understand how bugs were introduced. Essentially, code review enables future auditing of changes.

In our survey, we further validated this coding scheme. We asked what authors found valuable about code review comments; they could select one or more of the four themes and/or write in their own. Each of the four themes identified earlier was selected by between 8 and 11 respondents in the context of a particular code review, thus providing additional confidence that the coding scheme described above aligns with developers' perception of the value of the code review.

Although these expectations can map over those found previously at Microsoft [4], the main focus at Google, as explained by our participants, is on education as well as code readability and understandability, in line with the historical impetus for review. For this reason, the focus does not align with that found by Rigby and Bird (*i.e.*, a group problem solving activity) [33].

> ***Finding 1***. *Expectations for code review at Google do not center around problem solving. Reviewing was introduced at Google to ensure code readability and maintainability. Today's developers also perceive this educational aspect, in addition to maintaining norms, tracking history, gatekeeping, and accident prevention. Defect finding is welcomed but not the only focus.*

As stated earlier, when coding interview transcripts we also tracked the review context in which a theme was mentioned. We found that the relative importance of these different themes depends on the relationship between the author and

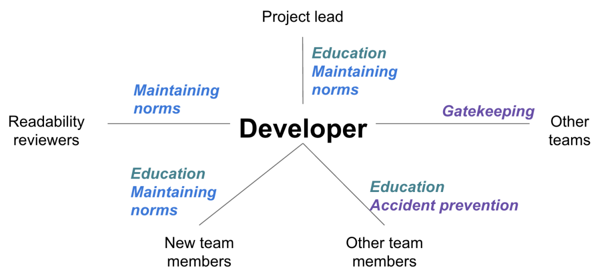

Figure 1: Relationship diagram describing which themes of review expectations appeared primarily within a particular author/reviewer context.

reviewer (Figure 1). For example, maintaining norms came up between an engineer and those with a different seniority (project leads, expert readability reviewers or "new" team members) but less so with their peers or other teams, where instead gatekeeping and accident prevention are primary. The educational benefits are broadly valued and encompass several different relationships.

> ***Finding 2***. *Expectations about a specific code review at Google depend on the work relationship between the author and reviewers.*

5 RESULTS: PRACTICES

In our second research question, we describe the code review process and compare its quantitative aspects with the convergent practices found in previous work [33].

5.1 Describing The Review Process

The code review at Google is linked to two concepts: *ownership* and *readability*. We first introduce them, then we describe the flow of the review process, and we conclude with the distinct characteristics of the internal review tool, Critique, that serve in contrast with other review tools.

Ownership. The Google codebase is arranged in a tree structure, where each directory is explicitly *owned* by a set of people. Although any developer can propose a change to any part of the codebase, an owner of the directory in question (or a parent directory) must review and approve the change before it is committed; even directory owners are expected to have their code reviewed before committing.

Readability. Google defines a concept called *readability*, which was introduced very early on to ensure consistent code style and norms within the codebase. Developers can gain readability certification in a particular language. To apply for readability, a developer sends changes to a set of readability reviewers; once those reviewers are confident the developer understands the code style and best practices for a language, the developer is granted readability for that language. Every change must be either authored or reviewed by someone with a readability certification for the language(s) used.

Code Review Flow. The flow of a review is tightly coupled with Critique and works as follows:

1. *Creating:* Authors start modifying, adding, or deleting some code; once ready, they create a change.
2. *Previewing:* Authors then use Critique to view the diff of the change and view the results of automatic code analyzers (e.g. from Tricorder [36]). When they are ready, authors mail the change out to one or more reviewers.
3. *Commenting:* Reviewers view the diff in the web UI, drafting comments as they go. Program analysis results, if present, are also visible to reviewers. *Unresolved* comments represent action items for the change author to definitely address. *Resolved* comments include optional or informational comments that may not require any action by a change author.
4. *Addressing Feedback:* The author now addresses comments, either by updating the change or by replying to comments. When a change has been updated, the author uploads a new snapshot. The author and reviewers can look at diffs between any pairs of snapshots to see what changed.
5. *Approving:* Once all comments have been addressed, the reviewers approve the change and mark it 'LGTM' (Looks Good To Me). To eventually commit a change, a developer typically must have approval from at least one reviewer. Usually, only one reviewer is required to satisfy the aforementioned requirements of ownership and readability.

We attempt to quantify how "lightweight" reviews are (CP_1). We measure how much back-and-forth there is in a review, by examining how many times a change author mails out a set of comments that resolve previously unresolved comments. We make the assumption that one iteration corresponds to one instance of the author resolving some comments; zero iterations means that the author can commit immediately. We find that over 80% of all changes involve at most one iteration of resolving comments.

Suggesting Reviewers. To identify the best person to review a change, Critique relies on a tool that analyzes the change and suggests possible reviewers. This tool identifies the smallest set of reviewers needed to fulfill the review requirements for all files in a change. Note that often only one reviewer is required since changes are often authored by someone with ownership and/or readability for the files in question. This tool prioritizes reviewers that have recently edited and/or reviewed the included files. New team members are explicitly added as reviewers since they have not yet built up reviewing/editing history. Unassigned reviewers can also make comments on (and can potentially approve) changes. Tool support for finding reviewers is typically only necessary for changes to files beyond those for a particular team. Within a team, developers know who to send changes to. For changes that could be sent to anyone on the team, many teams use a system that assigns reviews sent to the team email address to configured team members in a round-robin manner, taking into account review load and vacations.

Code Analysis Results. Critique shows code analysis results as comments along with human ones (although in different colors). Analyzers (or reviewers) can provide suggested edits, which can be both proposed and applied to the change via Critique. To vet changes before they are committed, development at Google also includes pre-commit hooks: checks where a failure requires an explicit override by the developer to enable a commit. Pre-commit checks include things like basic automated style checking and running automated test suites related to a change. The results of all pre-commit checks are visible in the code review tool. Typically, pre-commit checks are automatically triggered. These checks are configurable such that teams can enforce project-specific invariants and automatically add email lists to changes to raise awareness and transparency. In addition to pre-commit results, Critique displays the result of a variety of automated code analyses through Tricorder [36] that may not block committing a change. Analysis results encompass simple style checks, more complicated compiler-based analysis passes and also project-specific checks. Currently, Tricorder includes 110 analyzers, 5 of which are plugin systems for hundreds of additional checks, in total analyzing more than 30 languages.

***Finding 3**. The Google code review process is aligned with the convergent practice of being lightweight and flexible. In contrast to other studied systems, however, ownership and readability are explicit and play a key role. The review tool includes reviewer recommendation and code analysis results.*

5.2 Quantifying The Review Process

We replicate quantitative analysis that Rigby and Bird conducted that led to the discovery CP_{2-4} so as to compare these practices to the traits that Google has converged to.

Review frequency and speed. Rigby and Bird found that fast-paced, iterative development also applies to modern code review: In their projects, developers work in remarkably consistent short intervals. To find this, they analyzed review frequency and speed.

At Google, in terms of frequency, we find that the median developer authors about 3 changes a week, and 80 percent of authors make fewer than 7 changes a week. Similarly, the median for changes reviewed by developers per week is 4, and 80 percent of reviewers review fewer than 10 changes a week. In terms of speed, we find that developers have to wait for initial feedback on their change a median time of under an hour for small changes and about 5 hours for very large changes. The overall (all code sizes) median latency for the *entire* review process is under 4 hours. This is significantly lower than the median time to approval reported by Rigby and Bird [33], which is 17.5 hours for AMD, 15.7 hours for Chrome OS and 14.7, 19.8, and 18.9 hours for the three Microsoft projects. Another study found the median time to approval at Microsoft to be 24 hours [14].

Review size. Rigby and Bird argued that quick review time could only be enabled by smaller changes to review and subsequently analyzed review sizes. At Google, over 35% of the changes under consideration modify only a single file and about 90% modify fewer than 10 files. Over 10% of changes modify only a single line of code, and the median number of lines modified is 24. The median change size is significantly lower than reported by Rigby and Bird for companies such as AMD (44 lines), Lucent (263 lines), and Bing, Office and SQL Server at Microsoft (somewhere between those boundaries), but in line for change sizes in open source projects [33].

Number of reviewers and comments. The optimal number of reviewers has been controversial among researchers, even in the deeply investigated code inspections [37]. Rigby and Bird investigated whether the considered projects converged to a similar number of involved reviewers. They found this number to be two, remarkably regardless of whether the reviewers were explicitly invited (such as in Microsoft projects, who invited a median of up to 4 reviewers) or whether the change was openly broadcasted for review [33].
At Google, by contrast, fewer than 25% of changes have more than one reviewer,[1] and over 99% have at most five reviewers with a median reviewer count of 1. Larger changes tend to have more reviewers on average. However, even very large changes on average require fewer than two reviewers.
Rigby and Bird also found a "minimal increase in the number of comments about the change when more [than 2] reviewers were active" [33], and concluded that two reviewers find an optimal number of defects. At Google, the situation is different: A greater number of reviewers results in a greater average number of comments on a change. Moreover, the average number of comments per change grows with the number of lines changed, reaching a peak of 12.5 comments per change for changes of about 1250 lines. Changes larger than this often contain auto-generated code or large deletions, resulting in a lower average number of comments.

Finding 4. *Code review at Google has converged to a process with markedly quicker reviews and smaller changes, compared to the other projects previously investigated. Moreover, one reviewer is often deemed as sufficient, compared to two in the other projects.*

6 RESULTS: DEVELOPERS' PERCEPTIONS

Our last research question investigates challenges and satisfaction of developers with code review at Google.

6.1 Code Review Breakdowns at Google

Previous studies investigated challenges in the review process overall [4, 26] and provided compelling evidence, also corroborated by our experience as engineers, that understanding code to review is a major hurdle. To broaden our empirical body of knowledge, we focus here on challenges encountered in specific reviews ("breakdowns"), such as delays or disagreements.

[1] 30% of changes have comments by more than one commenter, meaning that about 5% of changes have additional comments from someone that did not act as an approver for the change.

Five main themes emerged from the analysis of our interview data. The first four themes regard breakdowns concerning aspects of the *process*:

Distance: Interviewees perceive distance in code review from two perspectives: geographical (*i.e.*, the physical distance between the author and reviewers) and organizational (*e.g.*, between different teams or different roles). Both these types of distance are perceived as the cause of delays in the review process or as leading to misunderstandings.

Social interactions: Interviewees perceive the communication within code review as possibily leading to problems from two aspects: *tone* and *power*. Tone refers to the fact that sometimes authors are sensitive to review comments; sentiment analysis on comments has provided evidence that comments with negative tone are less likely to be useful [11]. Power refers to using the code review process to induce another person to change their behavior; for example, dragging out reviews or withholding approvals. Issues with tone or power in reviews can make developers uncomfortable or frustrated with the review process.

Review subject: The interviews referenced disagreements as to whether code review was the most suitable context for reviewing certain aspects, particularly *design reviews*. This resulted in mismatched expectations (*e.g.*, some teams wanted most of the design to be complete before the first review, others wanted to discuss design in the review), which could cause friction among participants and within the process.

Context: Interviewees allowed us to see that misunderstandings can arise based on not knowing what gave rise to the change; for example, if the rationale for a change was an urgent fix to a production problem or a "nice to have" improvement. The resulting mismatch in expectations can result in delays or frustration.

The last theme regarded the *tool* itself:

Customization: Some teams have different requirements for code review, *e.g.*, concerning how many reviewers are required. This is a technical breakdown, since arbitrary customizations are not always supported in Critique, and may cause misunderstandings around these policies. Based on feedback, Critique recently released a new feature that allows change authors to require that all reviewers sign off.

6.2 Satisfaction and Time Invested

To understand the significance of the identified concerns, we used part of the survey to investigate whether code review is overall perceived as valuable.

We find (Table 2) that code review is broadly seen as being valuable and efficient within Google – all respondents agreed with the statement that code review is valuable. This sentiment is echoed by internal satisfaction surveys we conducted on Critique: 97% of developers are satisfied with it.

Within the context of a specific change, the sentiment is more varied. The least satisfied replies were associated with changes that were very small (1 word or 2 lines) or with changes which were necessary to achieve some other goal, *e.g.*, triggering a process from a change in the source code.

	Strongly disagree				Strongly agree
For this change, the review process was a good use of my time	2	4	14	11	13
Overall, I find code review at Google valuable	0	0	0	14	30

	Too little				Too much
For this change, the amount of feedback was	2	2	34	5	0

Table 2: User satisfaction survey results

However, the majority of respondents felt that the amount of feedback for their change was appropriate. 8 respondents described the comments as not being helpful, of these 3 provided more detail, stating that the changes under review were small configuration changes for which code review had less impact. Only 2 respondents said the comments had found a bug.

To contextualize the answers in terms of satisfaction, we also investigate the time developers spend reviewing code. To accurately quantify the reviewer time spent, we tracked developer interactions with Critique (*e.g.*, opened a tab, viewed a diff, commented, approved a change) as well as other tools to estimate how long developers spend reviewing code per week. We group sequences of developer interactions into blocks of time, considering a "review session" to be a sequence of interactions related to the same uncommitted change, by a developer other than the change author, with no more than 10 minutes between each successive interaction. We sum up the total number of hours spent across all review sessions in the five weeks that started in October 2016 and then calculate the average per user per week, filtering out users for whom we do not have data for all 5 weeks. We find that developers spend an average of 3.2 (median 2.6 hours a week) reviewing changes. This is low compared to the 6.4 hours/week of self-reported time for OSS projects [10].

***Finding 5**. Despite years of refinement, code review at Google still faces breakdowns. These are mostly linked to the complexity of the interactions that occur around the reviews. Yet, code review is strongly considered a valuable process by developers, who spend around 3 hours a week reviewing.*

7 DISCUSSION

We discuss the themes that have emerged from this investigation, which can inspire practitioners in the setting up of their code review process and researchers in future investigations.

7.1 A Truly Lightweight Process

Modern code review was born as a lighter weight alternative to the cumbersome code inspections [4]; indeed Rigby and Bird confirmed this trait (CP_1) in their investigation across systems [33]. At Google, code review has converged to an even more lightweight process, which developers find both valuable and a good use of their time.

Median review times at Google are much shorter than in other projects [14, 33, 34]. We postulate that these differences are due to the culture of Google on code review (strict reviewing standards and expectations around quick turnaround times for review). Moreover, there is a significant difference with reviewer counts (one at Google vs. two in most other projects); we posit that having one reviewer helps make reviews fast and lightweight.

Both low review times and reviewer counts may result from code review being a required part of the developer workflow; they can also result from small changes. The median change size in OSS projects varies from 11 to 32 lines changed [34], depending on the project. At companies, this change size is typically larger [33], sometimes as high as 263 lines. We find that change size at Google more closely matches OSS: most changes are small. The size distribution of changes is an important factor in the quality of the code review process. Previous studies have found that the number of useful comments decreases [11, 14] and the review latency increases [8, 24] as the size of the change increases. Size also influences developers' perception of the code review process; a survey of Mozilla contributors found that developers feel that size-related factors have the greatest effect on review latency [26]. A correlation between change size and review quality is acknowledged by Google and developers are strongly encouraged to make small, incremental changes (with the exception of large deletions and automated refactoring). These findings and our study support the value of reviewing small changes and the need for research and tools to help developers create such small, self-contained code changes for review [6, 15].

7.2 Software Engineering Research in Practice

Some of the practices that are part of code review at Google are aligned with practices proposed in software engineering research. For example, a study of code ownership at Microsoft found that changes made by minor contributors should be received with more scrutiny to improve code quality [9]; we found that this concept is enforced at Google through the requirement of an approval from an owner. Also, previous research has shown that typically one reviewer for a change will take on the task of checking whether code matches conventions [4]; readability makes this process more explicit. In the following, we focus on the features of Critique that make it a '*next generation code review tool*' [26].

Reviewer recommendation. Researchers found that reviewers with prior knowledge of the code under review give more useful comments [4, 11], thus tools can add support for reviewer selection [11, 26]. We have seen that reviewer recommendation is tool-supported, prioritizing those who recently edited/reviewed the files under review. This corroborates recent research that frequent reviewers make large contributions to the evolution of modules and should be included along with frequent editors [40]. Detecting the right reviewer does not seem problematic in practice at Google, in fact the model of recommendation implemented is straightforward since it can programmatically identify owners. This is in contrast with other proposed tools that identify reviewers who have reviewed files with similar names [43] or take into account such features as the number of comments included in reviews [42]. At Google, there is also a focus on dealing with reviewers' workload and temporary absences (in line with a study at Microsoft [4]).

Static Analysis integration. A qualitative study of 88 Mozilla developers [26] found that static analysis integration was the most commonly-requested feature for code review. Automated analyses allow reviewers to focus on the understandability and maintainability of changes, instead of getting distracted by trivial comments (*e.g.*, about formatting). Our investigation at Google showed us the practical implications of having static analysis integration in a code review tool. CRITIQUE integrates feedback channels for analysis writers [36]: Reviewers have the option to click "Please fix" on an analysis-generated comment as a signal that the author should fix the issue, and either authors or reviewers can click "Not useful" in order to flag an analysis result that is not helpful in the review process. Analyzers with high "Not useful" click rates are fixed or disabled. We found that this feedback loop is critical for maintaining developer trust in the analysis results.

A Review Tool Beyond Collaborative Review. Finally, we found strong evidence that CRITIQUE's uses extend beyond reviewing code. Change authors use CRITIQUE to examine diffs and browse analysis tool results. In some cases, code review is part of the development process of a change: a reviewer may send out an unfinished change in order to decide how to finish the implementation. Moreover, developers also use CRITIQUE to examine the history of submitted changes long after those changes have been approved; this is aligned with what Sutherland and Venolia envisioned as a beneficial use of code review data for development [38]. Future work can investigate these unexpected and potentially impactful non-review uses of code review tools.

7.3 Knowledge spreading

Knowledge transfer is a theme that emerged in the work by Rigby and Bird [33]. In an attempt to measure knowledge transfer due to code review, they built off of prior work that measured expertise in terms of the number of files changed [28], by measuring the number of distinct files changed, reviewed, and the union of those two sets. They find that developers know about more files due to code review.

At Google, knowledge transfer is part of the educational motivation for code review. We attempted to quantify this effect by looking at comments and files edited/reviewed. As developers build experience working at Google, the average number of comments on their changes decreases (Figure 2).

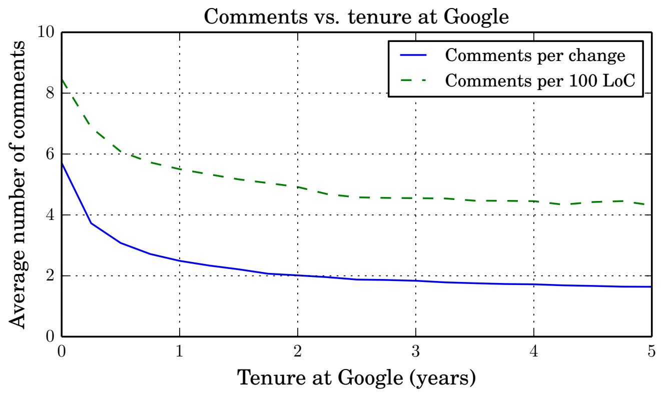

Figure 2: Reviewer comments vs. author's tenure at Google

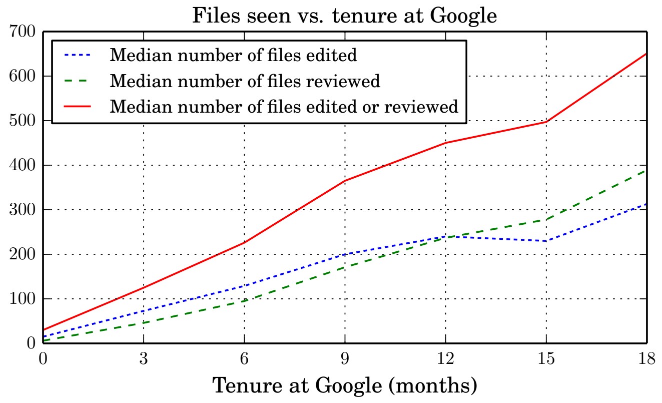

Figure 3: The number of distinct files seen (edited or reviewed, or both) by a full-time employee over time.

Developers at Google who have started within the past year typically have more than twice as many comments per change. Prior work found that authors considered comments with questions from the reviewer as not useful, and the number of not useful comments decreases with experience [11]. We postulate that this decrease in commenting is a result of reviewers needing to ask fewer questions as they build familiarity with the codebase and corroborates the hypothesis that the educational aspect of code review may pay off over time. Also, we can see that the number of distinct files edited and reviewed by engineers at Google, and the union of those two sets, increase with seniority (Figure 3) and the total number of files seen is clearly larger than the number of files edited. For this graph, we bucket our developers by how long they have been at the company (in 3-month increments) and then compute the number of files they have edited and reviewed. It would be interesting in future work to better understand how reviewing files impacts developer fluency [44].

8 CONCLUSION

Our study found code review is an important aspect of the development workflow at Google. Developers in all roles see it as providing multiple benefits and a context where developers can teach each other about the codebase, maintain the integrity of their teams' codebases, and build, establish, and evolve norms that ensure readability and consistency of the codebase. Developers reported they were happy with the requirement to review code. The majority of changes are small, have one reviewer and no comments other than the authorization to commit. During the week, 70% of changes are committed less than 24 hours after they are mailed out for an initial review. These characteristics make code review at Google lighter weight than the other projects adopting a similar process. Moreover, we found that Google includes several research ideas in its practice, making the practical implications of current research trends visible.

ACKNOWLEDGMENTS

The authors would like to thank Elizabeth Kemp for her assistance with the qualitative study, Craig Silverstein for context, and past and present members of the Critique team for feedback, including in particular Ben Rohlfs, Ilham Kurnia, and Kate Fitzpatrick. A. Bacchelli gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529.

REFERENCES

[1] 2017. Appendix to this paper. https://goo.gl/dP4ekg. (2017).
[2] A.F. Ackerman, L.S. Buchwald, and F.H. Lewski. 1989. Software inspections: An effective verification process. *IEEE Software* 6, 3 (1989), 31–36.
[3] A.F. Ackerman, P.J. Fowler, and R.G. Ebenau. 1984. Software inspections and the industrial production of software. In *Symposium on Software validation: inspection-testing-verification-alternatives*.
[4] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *ICSE*.
[5] Vipin Balachandran. 2013. Reducing Human Effort and Improving Quality in Peer Code Reviews Using Automatic Static Analysis and Reviewer Recommendation. In *ICSE*.
[6] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K Lahiri. 2015. Helping developers help themselves: Automatic decomposition of code review changesets. In *ICSE*.
[7] V.R. Basili, F. Shull, and F. Lanubile. 1999. Building knowledge through families of experiments. *IEEE Trans. on Software Eng.* 25, 4 (1999), 456–473.
[8] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W Godfrey. 2013. The influence of non-technical factors on code review. In *WCRE*.
[9] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 2011. Don't touch my code!: examining the effects of ownership on software quality. In *FSE*.
[10] Amiangshu Bosu and Jeffrey C Carver. 2013. Impact of peer code review on peer impression formation: A survey. In *ESEM*.
[11] Amiangshu Bosu, Michaela Greiler, and Christian Bird. 2015. Characteristics of useful code reviews: an empirical study at Microsoft. In *MSR*.
[12] chromium 2016. Chromium developer guidelines. https://www.chromium.org/developers/contributing-code. (August 2016).
[13] J.W. Creswell. 2009. *Research design: Qualitative, quantitative, and mixed methods approaches* (3rd ed.). Sage Publications.
[14] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. 2015. Code Reviews Do Not Find Bugs: How the Current Code Review Best Practice Slows Us Down. In *ICSE (SEIP)*.
[15] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. 2015. Untangling fine-grained code changes. In *SANER*.
[16] M.E. Fagan. 1976. Design and code inspections to reduce errors in program development. *IBM Systems Journal* 15, 3 (1976), 182–211.
[17] gerrit 2016. https://www.gerritcodereview.com/. (August 2016).
[18] githubpull 2016. GitHub pull request process. https://help.github.com/articles/using-pull-requests/. (2016).
[19] Barney G Glaser and Anselm L Strauss. 2009. *The discovery of grounded theory: Strategies for qualitative research*. Transaction Books.
[20] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. 2016. Work Practices and Challenges in Pull-Based Development: The Contributor's Perspective. In *ICSE*.
[21] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie van Deursen. 2015. Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective. In *ICSE*.
[22] Toshiki Hirao, Akinori Ihara, Yuki Ueda, Passakorn Phannachitta, and Ken-ichi Matsumoto. 2016. The Impact of a Low Level of Agreement Among Reviewers in a Code Review Process. In *IFIP International Conference on Open Source Systems*.
[23] Harrie Jansen. 2010. The logic of qualitative survey research and its position in the field of social research methods. In *Forum Qualitative Sozialforschung*, Vol. 11.
[24] Yujuan Jiang, Bram Adams, and Daniel M German. 2013. Will my patch make it? and how fast?: Case study on the linux kernel. In *MSR*.
[25] Sami Kollanus and Jussi Koskinen. 2009. Survey of software inspection research. *The Open Software Engineering Journal* 3, 1 (2009), 15–34.
[26] Oleksii Kononenko, Olga Baysal, and Michael W Godfrey. 2016. Code review quality: How developers see it. In *ICSE*.
[27] Matthew B Miles and A Michael Huberman. 1994. *Qualitative data analysis: An expanded sourcebook*. Sage.
[28] Audris Mockus and James D Herbsleb. 2002. Expertise browser: a quantitative approach to identifying expertise. In *ICSE*.
[29] phabricator 2016. Meet Phabricator, The Witty Code Review Tool Built Inside Facebook. https://techcrunch.com/2011/08/07/oh-what-noble-scribe-hath-penned-these-words/. (August 2016).
[30] Rachel Potvin and Josh Levenburg. 2016. Why Google Stores Billions of Lines of Code in a Single Repository. *Commun. ACM* (2016).
[31] Teade Punter, Marcus Ciolkowski, Bernd Freimut, and Isabel John. 2003. Conducting on-line surveys in software engineering. In *Empirical Software Engineering*.
[32] P. Rigby, B. Cleary, F. Painchaud, M.A. Storey, and D. German. 2012. Open Source Peer Review–Lessons and Recommendations for Closed Source. *IEEE Software* (2012).
[33] Peter C Rigby and Christian Bird. 2013. Convergent software peer review practices. In *FSE*.
[34] Peter C. Rigby, Daniel M. German, Laura Cowen, and Margaret-Anne Storey. 2014. Peer Review on Open-Source Software Projects: Parameters, Statistical Models, and Theory. *IEEE Transactions on Software Engineering* 23, 4, Article 35 (2014).
[35] David L Sackett. 1979. Bias in analytic research. *Journal of chronic diseases* 32, 1-2 (1979), 51–63.
[36] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: building a program analysis ecosystem. In *ICSE*.
[37] F. Shull and C. Seaman. 2008. Inspecting the History of Inspections: An Example of Evidence-Based Technology Diffusion. *IEEE Software* 25, 1 (2008), 88–90.
[38] Andrew Sutherland and Gina Venolia. 2009. Can peer code reviews be exploited for later information needs?. In *ICSE*.
[39] William Isaac Thomas and Dorothy Swaine Thomas. 1928. The methodology of behavior study. *The child in America: Behavior problems and programs* (1928), 553–576.
[40] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. 2016. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *ICSE*.
[41] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. 2017. Review participation in modern code review. *Empirical Software Engineering* 22, 2 (2017), 768–817.
[42] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. 2015. Who should review my code? A file location-based code-reviewer recommendation approach for modern code review. In *SANER*.
[43] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. 2016. Automatically recommending peer reviewers in modern code review. *IEEE Transactions on Software Engineering* 42, 6 (2016), 530–543.
[44] Minghui Zhou and Audris Mockus. 2010. Developer Fluency: Achieving True Mastery in Software Projects. In *FSE*.