(will be inserted by the editor)

Does Code Review Speed Matter for Practitioners?

Gunnar Kudrjavets · Ayushi Rastogi

To all the unsung heroes who commit and review code every day.

Received: date / Accepted: date

Abstract Increasing code velocity is a common goal for a variety of software projects. The efficiency of the code review process significantly impacts how fast the code gets merged into the final product and reaches the customers. We conducted a survey to study the code velocity-related beliefs and practices in place. We analyzed 75 completed surveys from 39 participants from the industry and 36 from the open-source community. Our critical findings are (a) the industry and open-source community hold a similar set of beliefs, (b) quick reaction time is of utmost importance and applies to the tooling infrastructure and the behavior of other engineers, (c) time-to-merge is the essential code review metric to improve, (d) engineers have differing opinions about the benefits of increased code velocity for their career growth, and (e) the controlled application of the commit-then-review model can increase code velocity. Our study supports the continued need to invest in and improve code velocity regardless of the underlying organizational ecosystem.

Keywords Code review, code velocity, developer productivity, time-to-merge

1 Introduction

Traditional software development methodologies, such as the waterfall model, focus on a rigid and highly predictable development and deployment schedule. With the introduction of the Agile Manifesto in 2001, the focus of modern approaches to software development has shifted to continuous deployment of incremental code changes (Martin, 2002). As a result, Continuous Integration (CI) and Continuous Deployment (CD) (Fowler, 2006) have become default practices for most of the projects in the industry and open-source software. A critical objective in the software industry is making code changes reach the production environment fast. The code review process

G. Kudrjavets • A. Rastogi

Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence

University of Groningen, 9712 CP Groningen, Netherlands

Tel.: +31-50-363-3939 Fax.: +31-50-363-3800

E-mail: {g.kudrjavets,a.rastogi}@rug.nl

is a time-consuming part of evaluating the quality of code changes and approving their deployment. With the wide adoption of Modern Code Review (Bacchelli and Bird, 2013; Sadowski et al., 2018) principles, engineers are now under more pressure than ever to review and deploy their code promptly. One of the aspects of Modern Code Review that negatively impacts software production is the perception that the code review process is time-consuming (Cunha et al., 2021b).

In this paper, we define *code velocity* as "the time between making a code change and shipping the change to customers" (Microsoft Research, 2019). We focus on the total duration of code review completion and merge time. We include a detailed description of related terminology in Section 2.2.

Based on our experiences in the last two decades with commercial and open-source software development, we have witnessed various practices and beliefs related to increasing code velocity. The opinions related to increasing code velocity range from the willingness to deploy code faster, even if it means increased defect density (Kononenko et al., 2016; Kushner, 2011), to taking as much time as necessary to "get the code right" (The Linux Foundation, 2022). These opposing views raise questions about the developer community's prevalent attitudes and beliefs toward code velocity.

Researchers have investigated the different aspects of the Modern Code Review process in-depth (Nazir et al., 2020; Weißgerber et al., 2008; Bacchelli and Bird, 2013; Czerwonka et al., 2015). We are unaware of any studies focusing specifically on the beliefs, challenges, and trade-offs associated with increasing the code velocity. The primary goal of this study is to search for the beliefs and practices about code velocity as-is and the context in which they hold. We target a variety of experienced practitioners who contribute or review code for commercial and open-source software.

We formulate the following research questions:

RQ1: What beliefs and convictions are related to code velocity for industry and open-source software developers?

 $\mathbf{RQ2}$: What compromises are engineers willing to make to increase code velocity?

RQ3: What are the essential suggestions from practitioners to increase code velocity?

To gather field data, we survey engineers who either submit or perform code reviews as part of their daily work. We describe the recruitment of survey participants in Section 3.2. Out of 75 respondents, we classify 39 individuals as industry participants and 36 as open-source software contributors. We asked survey participants various Likert-style questions related to the essence of our research inquiries. Finally, we solicited free-form suggestions about how to increase code velocity.

Our critical findings are that (a) respondents working on both commercial and open-source software respond similarly on Likert-type items (out of 24 items, only 4 have a statistically significant difference between these two groups), (b) while there is strong opposition to abandoning the code review process, using the *commit-then-review* model under some conditions can be acceptable, and (c) respondents mainly focused on the development process, infrastructure and tooling support, response time, and the need to schedule pre-allocated time for code reviews to increase code velocity.

Our study suggests that the maximum acceptable size of the code review on the median is 800 source lines of code (SLOC). This number is an order of a magnitude

larger than developer folklore and existing code review guidelines suggest. We find that the metric associated with code review periods that engineers find the most useful is time-to-merge, followed by time-to-accept and time-to-first-response. That finding confirms what previous studies and grey literature have documented (Izquierdo-Cortazar et al., 2017; Tanna, 2021).

Issues of concern include a need for more conviction that increased code velocity benefits an engineer's career growth, slow response times from either authors or code reviewers, and faster fault detection from various parts of the infrastructure.

2 Background and related work

2.1 Motivation for the study

Developer velocity plays a significant role in software projects' success and the overall job satisfaction of developers. The topic is important enough for Microsoft and GitHub to have a joint research initiative called Developer Velocity Lab (Microsoft Research, 2023). According to Microsoft, "[i]mproving developer velocity is critical to continued satisfaction, iteration, and innovation in software teams" (McMartin, 2021). GitLab considers the reduction in the code review time as the primary metric that describes the success of the code review process (Armstrong, 2022). Data from Meta shows "a correlation between slow diff review times (P75) and engineer dissatisfaction" (Riggs, 2022). In this context, a diff is a Meta-specific term equivalent to a code review, pull request, or patch (used mainly in open-source software).

In industry, the drive to increase the code velocity is significant enough even to warrant the development of unique bots and tools. These tools periodically remind either an author or reviewer that they block the completion of a code review. For example, at Microsoft, a bot that periodically *nudges* developers "was able to reduce pull request resolution time by 60% for 8,500 pull requests," with 73% of these notifications being resolved as positive (Maddila et al., 2022). Meta considers developer velocity as one of its critical investments during an economic downturn (Vanian, 2022). The company gives engineers a general incentive to "[m]ove fast and break things" (Kushner, 2011). In Meta's development philosophy, engineers expect certain defects to appear if it results in a faster product deployment (Feitelson et al., 2013). The startup culture practiced by many software companies encourages releasing new features "as fast as possible, for the sake of fuelling growth" (Frenkel and Kang, 2021).

Another critical point in our inquiry is the differences in opinions about code velocity between industry and open-source software developers. Fundamentally, the industry and open-source software development process is motivated by different incentives. However, the attitudes vary even in the context of open-source software. For the Linux kernel, "[t]he goal is to get the code right and not rush it in," according to the official kernel development guide from the Linux Foundation (The Linux Foundation, 2022). However, we notice the desire for increased code velocity in Mozilla. Based on a study about code review practices in Mozilla, it is sometimes acceptable to be less thorough during code reviews if it speeds up the code review process (Kononenko et al., 2016).

Corporate policies are not necessarily dictated by what individual engineers think but by business needs. Research shows that developers and managers have different views about productivity and quality (Storey et al., 2022). To discover the ground truth, we need to understand what engineers think is "right." The topic of code velocity can

surface strong emotions in engineers ("... I just hate it when things are unreviewed for days") (Söderberg et al., 2022). The survey mechanism that we use provides engineers with anonymity. That anonymity enables engineers to freely share their opinions even if they contradict the company's or project's official policies related to code velocity.

Empirical software engineering involves making daily trade-offs between various project characteristics. The trade-off between increasing code velocity and product quality has severe consequences because "poorly-reviewed code has a negative impact on software quality in large systems using modern reviewing tools" (McIntosh et al., 2015). We want to study what attributes or values engineers are willing to compromise to achieve higher code velocity.

2.2 Terminology and metrics

Most commercial organizations share a similar goal: reduce the duration of code reviews and consequently increase the code velocity.

The term code velocity can have different meanings depending on the context. A customer-centric definition of code velocity is "the time between making a code change and shipping the change to customers" (Microsoft Research, 2019). As a quantifier characterizing code churn, it is defined "as the average number of commits per day for the past year of commit activity" (Tsay, 2017). In this paper, our definition focuses on the total duration of code review completion and merge time. We use time-to-merge as the period from publishing the code review to when accepted code changes are merged to the target branch (Izquierdo-Cortazar et al., 2017). Terms like review time (from publishing the patch until its acceptance) (Tan and Zhou, 2019) and resolve time (Zhu et al., 2016) that is defined as "the time spent from submission to the final issue or pull request status operation (stopped at committed, resolved, merged, closed) of a contribution" are used as well. Using the lifetime of a code review coupled with merging time matches the period that the DevOps platforms such as GitLab optimize. The formal definition for GitLab's metric is "duration from the first merge request version to merged" (Armstrong, 2022).

Different commercial software companies measure various code review periods. Google's code review guidance states that "... it is the response time that we are concerned with, as opposed to how long it takes a CL to get through the whole review and be submitted" (Google, 2023b). The term CL in Google's nomenclature means "one self-contained change that has been submitted to version control or which is undergoing code review" (Google, 2023a). A study about code velocity from Microsoft finds that critical points in time for engineers are the first comment or sign-off from a reviewer and when the code review has been marked as completed (Bird et al., 2015). A paper that investigates the performance of code review in the Xen hypervisor project finds that time-to-merge is the metric to optimize for (Izquierdo-Cortazar et al., 2017). Anecdotal evidence from grey literature about the challenges in Capital One's code review process presents a similar finding—"the most important metric was to calculate the cycle time—that is, how long it takes from a PR being raised to it being merged (or closed)" (Tanna, 2021). Meta tracks a Time In Review metric, defined as "a measure of how long a diff is waiting on review across all of its individual review cycles" (Riggs, 2022). One of the findings from Meta is that "[t]he longer someone's slowest 25 percent of diffs take to review, the less satisfied they were by their code review process."

2.3 Expectations related to code velocity

We find that expected code review response times between industry and various opensource software projects differ by order of magnitude. Google sets an expectation that "we expect feedback from a code review within 24 (working) hours" (Winters et al., 2020). Findings from Meta confirm that "reviews start to feel slow after they have been waiting for around 24 hour review" (Chen et al., 2022). Guidance from Palantir is that "code reviews need to be prompt (on the order of hours, not days)" and "[i]f you don't think you can complete a review in time, please let the committer know right away so they can find someone else" (Palantir, 2018). Existing research into code review practices at AMD, Google, and Microsoft similarly converges on 24 hours (Rigby and Bird, 2013). The published guidelines and studies match our industry experience of 24 hours being a de facto expected period for a response.

The requirements for open-source software are less demanding than in the industry. The code review guidelines from the LLVM project (LLVM Foundation, 2023b) that focuses on various compiler and toolchain technologies set up an expectation that "code reviews will take longer than you might hope." Similarly, the expectations for Linux contributors are set to "it might take longer than a week to get a response" during busy times (The Linux Foundation, 2022). The guidance for Mozilla is to "strive to answer in a couple of days, or at least under a week" (Mozilla, 2023). For Blender, the expectation is that "[d]evelopers are expected to reply to patches [in] 3 working days" (Blender, 2022).

The etiquette for handling stale code reviews in open-source software differs from the industry. According to the guidelines from various projects, approaches like the Nudge bot are unacceptable (Maddila et al., 2022). The guidance for inactive code reviews for the Linux kernel is to "wait for a minimum of one week before requesting a response" (The Linux Foundation, 2022). The LLVM project also recommends waiting for a week in case of inactivity before reminding the reviewers (LLVM Foundation, 2023b). The FreeBSD commit guidelines state clearly that "the common courtesy ping rate is one week" (The FreeBSD Documentation Project, 2022).

3 Methodology

3.1 Survey design

The main goals of our survey are to collect data about the beliefs and experiences about code velocity, the trade-offs engineers are willing to make to increase the code velocity, and suggestions from practitioners about how to increase it. Our target audience was both commercial and open-source software developers.

The survey consists of 16 questions, with one additional asking for consent. All the questions except the one that determined the participants' consent were *optional*. At the end of the survey, we asked participants if they wanted to share their contact information with researchers. The first eight questions related to participants' demographics, such as experience with software development, code reviews, their role in the software development process, and the application domain they used to answer the survey questions. This question block is followed by questions that ask participants to rank time-to-first-response, time-to-accept, and time-to-merge in order of importance

to optimize the code velocity. After that, we present four questions related to the benefits of code velocity and potential compromises related to increasing code velocity. We inquire about the possibility of using either a post-commit review model (Rigby and German, 2006) or no code review process.

Questions Q11, Q12, Q13, and Q14 contain multiple *Likert-type items* (Clason and Dormody, 1994). Because of the survey's limited number of questions, we do not use classic Likert scales. Likert scales typically contain four or more Likert-type items combined to measure a single character or trait. For example, researchers may have 4–5 questions about the timeliness of responses to a code review. Researchers then merge the results from these questions into a single composite score. The categories we inquire about come from our combined subjective experiences with software development in the industry and open-source community. Some choices, such as "Career growth" and "Job satisfaction," are apparent. Others, such as "Diversity, equity, and inclusion," reflect the changing nature of societal processes.

The survey ends with asking participants about the maximum acceptable code review size and waiting period, followed by an open-ended question about how the code velocity can be improved. We indicated to participants that the survey would take 5–7 minutes. The complete list of survey questions is publicly accessible. ¹

3.2 Survey participants

Ethical considerations Ethical solicitation of participants for research that involves human subjects is challenging (Felderer and Horta Travassos, 2020). There is no clear consensus in the academic community about sampling strategy to solicit survey participants. The topic is under active discussion (Baltes and Diehl, 2016). In 2021, researchers from the University of Minnesota experimented with the Linux kernel (Feitelson, 2023). The approach the researchers used caused significant controversy and surfaced several issues surrounding research ethics on human subjects. The fallout from the "hypocrite commits" experiment (Wu and Lu, 2021) forced us to approach recruiting the survey participants with extreme caution.

A typical approach is to use e-mail addresses mined from software repositories (e.g., identities of commit authors) to contact software developers. Based on the recent guidance about ethics in data mining (Gold and Krinke, 2021; Gonzalez-Barahona, 2020), we decided not to do this. While research related to code reviews has in the past utilized an existing relationship with a single specific developer community such as Mozilla (Kononenko et al., 2016) or Shopify (Kononenko et al., 2018), we wanted to reach out to a broader audience. We also wanted to avoid incentive-based recruitment mechanisms that offer rewards. Existing findings suggest that monetary incentives increase survey response rates (Smith et al., 2019). However, they do not necessarily reduce the non-response bias (Groves, 2006). As a potential incentive, we promised that participants who share their contact information with us would be the first to receive the paper's preprint.

Recruitment strategy Our recruitment strategy used social media (Facebook, LinkedIn, Medium, Reddit, and Twitter) to spread the message to our network of professional software engineers, researchers, and students involved in the industry and open-source

¹ https://doi.org/10.5281/zenodo.7312098

software communities. We started by composing a Medium post in a non-academic style, making the content reachable to a broader audience. That post contained a link to our survey to make participation easier. In addition, we contacted several individuals in commercial software companies and various open-source software projects to ask their permission to share the survey invite with their developer community. We received responses from Blender, Gerrit, FreeBSD, and NetBSD. These projects gave us explicit permission to post in a project's mailing list, or our contact person circulated the survey internally.

Survey summary statistics The survey was published on September 14, 2022, and closed on October 14, 2022. The survey system received a total of 110 responses. Out of all the respondents, 76 participants completed the survey, with 75 agreeing to the consent form and answering the questions. Of 75 individuals who answered the questions, 25 discovered the survey using social media and 50 via anonymous survey link. For our analysis, we only used the surveys that participants fully completed.

3.3 Survey data analysis

The methods used to analyze data from Likert-style items are controversial without a clear scientific consensus (Brown, 2011; Carifio and Perla, 2007; Chen and Liu, 2020). This paper treats Likert-style items as ordinal measurements and uses descriptive statistics to analyze them (Allen and Seaman, 2007; Boone, Jr. and Boone, 2012). We do not treat ordinal values as metric because it can lead to errors (Liddell and Kruschke, 2018). For Likert-type items, we define three general categories: negative ("Strongly disagree," "Disagree"), neutral ("Neither agree nor disagree," "I don't know"), and positive ("Agree," "Strongly agree"). We added the choice of "I don't know" based on the feedback from the pilot tests that we used to refine the survey questions.

We use the content analysis to analyze the answers to Q17 ("In your opinion, how can code velocity be improved for your projects?"). Two researchers independently manually coded all the responses to Q17. Once the coding process was finished, the researchers compared their results and tried to achieve a consensus. In case of disagreements, a third researcher acted as a referee. Several themes and categories emerged as part of the open coding process. We repeated the coding process till we classified all the responses under 7 ± 2 labels (Miller, 1956).

Numerical data, such as the maximum acceptable size of the code review, was analyzed using custom code written in R. Similarly, the statistical tests conducted in Section 4.3 and Section 4.4 to evaluate the differences between various groups were implemented in R.

4 Results

4.1 Demographics

Most of the respondents to our survey are experienced software engineers. We find that:

- A 34% of respondents identified themselves as having 3–10 years of experience.
- A 62% stated that they have more than 10 years of experience.

Consequently, this experience translates to the time spent reviewing other people's code.

- A 63% of survey participants have over 5 years of experience reviewing code.
- A 36% of respondents submit more than 10 code reviews per month, and 43% submit 3–10 code reviews.
- A 54% of respondents conduct more than 10 code reviews per month and 38% conduct 3–10 code reviews.

For the type of software the survey participants work on, 58% identified as developers working on application software such as mobile or desktop applications. 27% of respondents stated that they work on systems software such as device drivers or kernel development. Regarding different code review environments, 95% of respondents use a code collaboration tool. That tool may be public, such as Gerrit or GitHub, or the private instance of a company-specific tool, such as Google's Critique. Nearly every respondent writes or reviews code as part of their role. Only one individual stated that their role does not require reviewing code, and they only submit patches. Out of the respondents, 89% of developers author new code changes. The rest of the survey participants have a different role, such as only reviewing the code.

4.2 Grouping of respondents

We divide our participants into two groups based on how they self-identify as a response to Q8 ("What type of software developer are you?"). We use the following proxy to understand the difference between industry and open-source software developers. If a participant chose only "I work on closed-source and get paid" as a response, we classify them as "Industry." If one of the choices by participants was either "I work on open-source and get paid" or "I work on open-source and do not get paid," then we classify them as "OSS." Based on that division, we ended up with 39 participants from the industry and 36 respondents from open-source software.

In Q9 ("Choose an application domain you are most experienced with for the remaining questions?"), we asked participants what type of software is their primary area of expertise. We chose not to divide participants based on the abstraction level of the software. We base that decision on the number of respondents and the varying size of the different groups. Out of 75 respondents, 42 identified as someone working on application software, 20 on systems software, 3 on real-time or critical software, 8 on other types of software, and 2 chose not to answer the question.

4.3 RQ1: beliefs and convictions related to code velocity

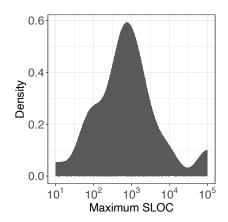
4.3.1 Expectations for the size and velocity of code reviews

Our industry experience related to code review size is highly variable. We have participated in projects where code reviews that contained thousands of SLOC were standard. Similarly, we have experience with projects where engineers required that authors split any review more extensive than 20–30 SLOC into separate reviews. Existing research suggests that the size of code reviews impacts their quality and speed (Jiang et al.,

2013; Weißgerber et al., 2008). Based on these findings, we investigate what engineers consider a maximum acceptable size for a code review.

Most open-source software projects do not have fixed guidelines for an upper bound for a code review. Very little quantitative guidance exists for the size of code reviews. Most guidelines use qualifiers such as "isolated" (LLVM Foundation, 2023a), "reasonable" (Chromium, 2023), and "small" (PostgreSQL, 2019; Phabricator, 2021; MacLeod et al., 2018b). For stable releases of the Linux kernel, the guidance is "[i]t cannot be bigger than 100 lines ..." (Linux, 2023). Google engineering practices specify some bounds: "100 lines is usually a reasonable size for a CL, and 1,000 lines is usually too large" (Google, 2023a). The acronym CL means "one self-contained change that has been submitted to version control or which is undergoing code review" (Google, 2023a). As anecdotal evidence, a respondent to a survey about code review practices states that "[a]nything more than 50 lines of changes, and my brain doesn't have the capacity to do a good code review" (Alami et al., 2020).

The sentiment about larger patch sizes is generally negative. A paper that investigates the efficiency of a code review process finds that "patch size negatively affects all outcomes of code review that we consider as an indication of effectiveness" (dos Santos and Nunes, 2017). The existing research directs developers towards more minor code changes. Anecdotal assessment from the Chromium contributor's guide is that "[r]eview time often increases exponentially with patch size" (Chromium, 2023). A study about code review performance finds that "review effectiveness is higher for smaller code changes" (Baum et al., 2019). Another study about participation in Modern Code Review finds that patches with smaller sizes receive fewer comments than the larger patches, and larger patches go through more iterations (Thongtanunam et al., 2017; Baysal et al., 2015).



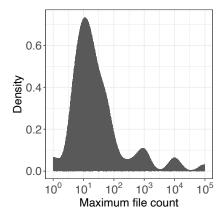


Fig. 1 Source lines of code.

Fig. 2 File count.

We asked study participants about the maximum acceptable number of SLOC and the number of files in the code review. Figure 1 and Figure 2 display the density plots ("smoothed histograms") of both variables. Before the analysis, we cleaned the data and removed entries that were "inconsistent with the remainder of the set of data" (Barnett and Lewis, 1984) and "surprising or discrepant to the investigator" (Beckman and Cook, 1983). We found only one entry for each metric that we considered an outlier. Shapiro-

Wilk tests (Shapiro and Wilk, 1965) confirmed that neither SLOC (W=0.33, p<.001) nor the file count (W=0.15, p<.001) were normally distributed. A Mann-Whitney U test (Mann and Whitney, 1947) indicated that the difference between medians for SLOC was not statistically significant $U(N_{\rm Industry}=33, N_{\rm OSS}=26)=979.5, z=-0.16, p=.87$. Similarly, we do not observe differences for the number of files $U(N_{\rm Industry}=34, N_{\rm OSS}=26)=1,088.5, z=0.78, p=.44$.

Observation 1

The median maximum acceptable number of SLOC for code review is 800. This finding is surprising given the constant theme of suggesting that developers should aim for more minor changes.

We discuss different code review periods and how there is no consensus on what to optimize in Section 2.1. Our goal is to understand what a heterogeneous collection of practitioners values the most when optimizing code velocity. We asked the participants to rank time-to-first-response, time-to-accept, and time-to-merge in the order of importance. A total of 65 participants ranked different code review periods. As a feedback, we also received 3 comments. We present the results in Table 1.

Table 1 Rankings of different code review periods in the order of importance to optimize for code velocity. Each column describes how many times a particular metric was ranked as a specific priority. The number of respondents and percentage of total responses per each entry is given. TTFR = time-to-first-response, TTA = time-to-accept, TTM = time-to-merge.

1 st priority		2 nd priority		$3^{\rm rd}$ priority	
1. TTM 2. TTFR 3. TTA 4. Other	32 (49%) 19 (29%) 12 (18%) 2 (3%)	1. TTA 2. TTFR 3. TTM 4. Other	33 (50%) 22 (34%) 10 (15%) 0 (0%)	1. TTFR 2. TTM 3. TTA 4. Other	24 (36%) 21 (32%) 20 (31%) 0 (0%)

In 49% of cases, the time-to-merge was ranked as the first priority metric to optimize. This result is like the findings from the Xen hypervisor project study (Izquierdo-Cortazar et al., 2017), anecdotal evidence from the industry (Tanna, 2021), and our personal experiences for more than two decades. One participant pointed out that "all of those metrics are totally irrelevant..." but did not clarify what else may be relevant. Two other comments suggested a different set of metrics: "response time on changes in review by both author and reviewers" (like *Time In Review* that Meta measures (Riggs, 2022)) and "[t]ime to re-review."

4.3.2 Perceived benefits of increased code velocity

We compare each Likert-style item between two groups separately using a Mann-Whitney U test. No statistically significant differences exist for any of the items between the industry and OSS groups. Figure 3 shows that for most categories, participants perceive increased code velocity as beneficial. Participants think that code velocity benefits aspects such as job satisfaction, reputation, or relationship with peers

The benefits of a code review process generally have been associated with career development and growth (Cunha et al., 2021a). However, in our study, the item with

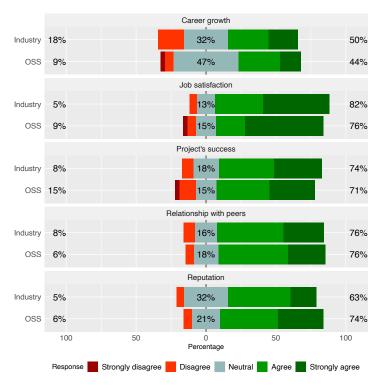


Fig. 3 Likert scales for the Q11 ("Velocity of your code improves your \dots ").

the lowest score is "Career growth," where only 50% of the industry and 44% of OSS group respondents rated it positively. Career growth in the corporate environment generally means increased professional scope, monetary rewards, and promotion speed. This finding is somewhat concerning. While intrinsic motivation is essential, it is hard to motivate engineers to conduct efficient code reviews if there is no substantial payoff.

For the OSS group, one possible explanation is that it is much more challenging to define career progression in the open-source software community than in a traditional corporate environment. However, given that only 50% of responses from the industry rated the "Career growth" category positively, we think this topic is worth exploring further.

Observation 2

On a median, 74% of rankings are positive regarding the belief that increased code velocity improves job satisfaction, project success, relationship with their peers, and reputation of engineers.

4.4 RQ2: compromises that are acceptable to increase code velocity

4.4.1 Commit-then-review model

The foundation of the Modern Code Review is the *review-then-commit* model. Some projects use the opposite of that approach. One major software project that started using the *commit-then-review* model was Apache (Rigby et al., 2008). While most of the projects have abandoned commit-then-review for review-then-commit, we wanted to study what developers think about the *resurrection of the commit-then-review model*. Several data points influence the decision to research this possibility. We discuss them below.

Industry experience The primary motivation to survey developers about commit-thenreview is our industry experience. We have witnessed several complaints and discussions about the "slowness" of the review-then-commit model. Developers are frustrated that even for trivial changes, such as one or two lines of code that fix formatting issues or compiler warnings, they must wait hours or days for someone to approve the code changes formally. Even organizations that use cutting-edge approaches to software development, such as Meta, state that "every diff must be reviewed, without exception" (Riggs, 2022). We frequently observe this frustration about the inflexibility of the code review process, primarily in teams of experienced and senior engineers.

Efficacy of code reviews Data from Microsoft shows that "[o]nly about 15% of comments provided by reviewers indicate a possible defect, much less a blocking defect" (Czerwonka et al., 2015). Given this finding, the trade-off between blocking the commits until the code review finishes to satisfy the process versus optimizing for code velocity and taking some risks needs investigation. Our observations indicate that senior engineers consider code reviews to be efficient only if the reviewer is as or more senior than the author. The quality of various linters and static analysis tools has improved over the years. We observe that tools automatically flag issues related to formatting, coding conventions, and fundamental coding issues without any human intervention.

Development process The establishment of CI and CD as de facto approaches to developing modern software shows that industry and open-source software value the speed of software delivery as the critical success metric.

Figure 4 displays how much participants are willing to compromise on various characteristics to increase the code velocity. We compared each Likert-style item separately using a Mann-Whitney U test. There are significant differences only between two items: "Code quality" and "Security." A Mann-Whitney U test for "Code quality" indicated that the difference in mean ranks is statistically significant $U(N_{\rm Industry}=39,N_{\rm OSS}=34)=1,664,z=2.65,p=.008$. Similarly, for "Security," a Mann-Whitney U test indicated that the difference in mean ranks is statistically significant $U(N_{\rm Industry}=39,N_{\rm OSS}=34)=1,621,z=2.33,p=.02$.

An encouraging finding in our survey is what developers think about compromises related to code quality and software security.

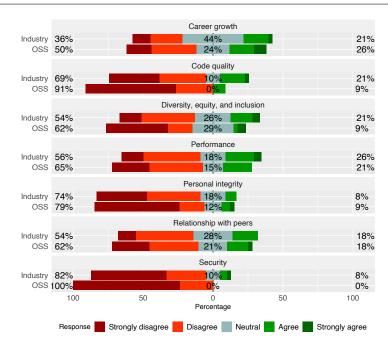


Fig. 4 Likert scales for the Q12 ("I am willing to compromise on ... if it improves code velocity").

Observation 3

For 100% of OSS developers, software security is something they are unwilling to compromise on to increase code velocity.

The 82% of industry developers who gave negative responses and 10% who gave neutral responses to this question share a similar sentiment. Code quality is something that 91% of OSS developers and 69% of industry participants are not willing to negotiate over. One potential explanation for the differences is that industry developers view code quality and security as one software characteristic regarding what they can make trade-offs. On the other hand, the OSS developers are "true believers" who are not willing to compromise to release less secure software.

As a next step, we asked participants what aspects of software a commit-then-review model can improve. We display the results in Figure 5. We compared each Likert-style item separately using a Mann-Whitney U test. There are significant differences only between two items: "Code velocity" and "Job satisfaction." A Mann-Whitney U test for "Code velocity" indicated that the difference in mean ranks is statistically significant $U(N_{\mathrm{Industry}} = 39, N_{\mathrm{OSS}} = 33) = 1,603.5, z = 2.12, p = .034$. The code velocity is also a category that most participants thought could be improved. Of industry respondents, 64% gave a positive response, with 48% of OSS respondents feeling similarly. Application of the commit-then-review model means that developers no longer have to wait for a code review to be complete. The potential improvements in code velocity are a logical result of this process change.

A Mann-Whitney U test for "Job satisfaction" indicated that the difference in mean ranks is statistically significant $U(N_{\mathrm{Industry}}=39,N_{\mathrm{OSS}}=33)=1,620,z=2.28,p=.023.$ Of industry respondents, 46% gave a positive response, with 24% of OSS respondents feeling similarly. Approximately half of the survey participants from the industry think that their job satisfaction could improve with the commit-thenreview model. The difference between the industry and OSS is as significant as two times. This finding makes sense because of how the industry evaluates the performance of software engineers. Based on our experience, the ability to complete the assigned tasks on time and reduce code review idle time is directly associated with engineers' anxiety levels and productivity.

For "Career growth," 51–52% of respondents in industry and OSS chose a neutral response. This finding indicates a need for more clarity regarding the relationship between code velocity and its direct impact on an individual's career. While it may be beneficial for an organization or a project to be released on a faster cadence, there is not necessarily a significant reward for individuals responsible for that cadence. This finding is like the data from Section 4.3.2, indicating that developers do not view an increase in code velocity as beneficial to their careers.

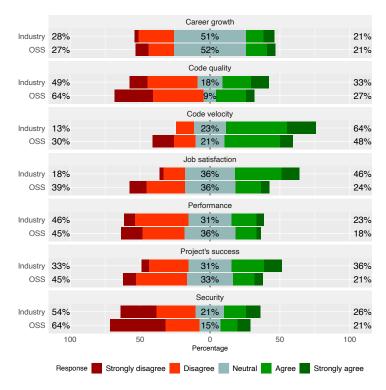


Fig. 5 Likert scales for the Q13 ("I think the post-commit review model (changes are first committed and then reviewed at some point later) can improve ...").

4.4.2 Abandonment of code reviews in favor of code velocity

Code reviews are optional in some contexts. For example, the FreeBSD project defines a committer as "an individual with write access to the FreeBSD source code repository" (FreeBSD Foundation, 2022). While committers are "required to have any nontrivial changes reviewed by at least one other person before committing them to the tree" (McKusick et al., 2015), the definition of nontrivial is open to interpretation. Therefore, experienced developers can commit code changes without the review at will. We have also witnessed multiple instances in the industry where a similar practice is employed. To increase the code velocity, senior developers with a significant contribution history to the project have ignored the official code review process or accepted each other's changes immediately to "satisfy the process."

Based on the observations from the industry and the committer model used by projects such as DragonFlyBSD, FreeBSD, NetBSD, and OpenBSD, we decided to survey what engineers think about the possibility of abandoning code reviews. We asked participants under what conditions engineers can commit code without someone else reviewing that code. We display the results in Figure 6 and discuss them below.

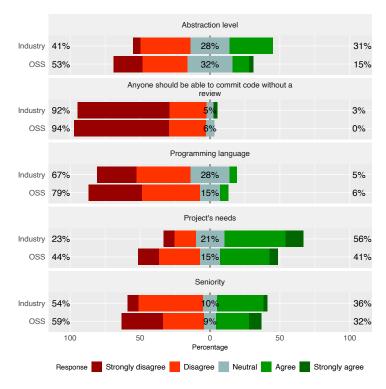


Fig. 6 Likert scales for the Q14 ("Should engineers be allowed to commit their code without the code review depending on \dots ").

We compared each Likert-style item separately using a Mann-Whitney U test. There were no statistically significant differences for any of the items between the "Industry" and "OSS" groups.

Observation 4

Developers are uniformly against abolishing the code review process. 92% of the industry and 94% of OSS respondents think the existence of the code review process is valuable.

The highest items with positive responses are "Project's needs" and "Seniority." For "Project's needs," 56% of the industry and 41% of OSS respondents thought it permissible to commit code without conducting a code review. Based on our industry experience, this sounds reasonable. Engineers must exercise their judgment in cases like build breaks or issues blocking an entire project and not blindly follow the process. For example, suppose an application is not building, and an engineer has a potential fix. In that case, it is reasonable to take a calculated risk to commit the changes immediately without waiting for hours for a code review.

The choice of "Seniority" is reasonable as well. Senior engineers typically have more tribal knowledge, related experience, and in-depth knowledge than junior engineers. Therefore, if anyone can occasionally "break the rules," it makes the most sense for them to do that. In our industry experience, code reviews can find apparent mistakes. However, finding problems in either complex algorithms or design nuances works best if a reviewer has a similar or higher level of knowledge. Code reviews where a junior engineer reviews the senior engineer's code are effective in detecting defects only in a subset of cases. Suppose the goal is to improve code velocity. In that case, we recommend that a project explicitly discuss the risk versus reward in a situation where senior engineers can exercise their judgment on when to require reviews for their changes.

4.5 RQ3: suggestions to increase code velocity

To solicit feedback from developers, we asked respondents, "In your opinion, how can code velocity be improved for your projects?" We display the word cloud that summarizes the comments from the survey participants in Figure 7. A word cloud is a widely used visualization technique to analyze and summarize qualitative data. The rendering of the most frequently used words (e.g., "commit," "time," "tooling," and "smaller") causes them to be displayed more prominently. While the word cloud is an initial indicator of the themes that emerge after analyzing the text corpus, a more detailed grouping of the results is necessary.

Two researchers manually analyzed and coded the 47 comments received from the survey participants. Each comment was assigned one or more labels depending on its content. After the coding, researchers discussed the results and disagreements (less than ten), normalized the labels used for coding, and summarized the findings. Our goal was to reach 7 ± 2 labels that adequately describe the nature of the suggestions (Miller, 1956). We display the labels and their distribution in Table 2.



Fig. 7 A word cloud of suggestions about how to improve code velocity.

Table 2 Different themes that result from coding the survey responses.

Name	Count
Code review size	5
Communication	6
Coordination and scheduling	13
Development process	16
Early fault detection	5
Infrastructure	15
Response time	12

4.5.1 Improved conventions and standards

We notice the desire for more formalized standards and the establishment of coding conventions. Going forward, we use the notation $\mathbf{R}i$ to indicate a respondent i. An R5 states that "[e]stablishing and following company-wide standards for coding style and code review" can be helpful. Similarly, R29 suggests that "[t]eam needs established coding conventions or tooling that enforces conventions to reduce debate." The standards help to set expectations and reduce the number of round-trips between an author and reviewer. According to R11, it will be helpful to "decrease the number of surprises - have the review criteria prepared and explicit beforehand as much as possible/sensible." An R56 points out that "[w]e need to improve our general code review guidelines - both

for reviewer and reviewee (like "reviewer is not tester!")." The sentiment is shared in R59 by asking for "stricter guidelines."

4.5.2 Prioritization of follow-up changes

Determining clearly what is critical and what is not is another suggested improvement. Feedback from R56 suggests that "code reviews shall be prioritised over new feature requests." An R30 suggests a potential two-phased approach to code reviews: "[b]e crisp on what is critical to fix and what can be done in a follow up" and "[s]top wasting time in LINT feedback and preferences about what code should look like and focus only on functionality and correctness." We have noticed similar behavior in the industry where reviewers try to separate the feedback into mandatory and optional. The optional items are improvements that can be made in an independent code review or later.

4.5.3 Infrastructure and tooling improvements

The critical requirement from infrastructure is fast capabilities for early fault detection. The general requirement is for "good code review tooling" for various tools "to perform more checks before actual "peer" review" (R37). Developers want "[a]utomated testing, tooling" (R59). The individual responses describe this need as "[f]aster automated signals with more actionable error" (R6), "[a]utomatic code linters, style cops, CI builds and CI test passes" (R9), and "... automated CI/CD environments for initial checks, builds, tests ..." (R28).

4.5.4 Response time

Compared to the status quo, the responses indicate the need for faster responses. The desire to respond quickly to code reviews is hardly a surprise. Various existing studies and code review guidelines specify that different periods of development processes should optimize (MacLeod et al., 2018b; Google, 2023b; Izquierdo-Cortazar et al., 2017). In addition to the increased anxiety caused by waiting for feedback, there are other drawbacks, such as the cost of a context switch and the potential for introducing new defects (Czerwonka et al., 2015). Respondents use phrases such as "more responsive engineers" (R63), "validation turn-around time" (R41), "[c]ommunicating with reviewers promptly" (R1), and "reducing the feedback loop" (R12) to describe the potential improvements. One of the respondents (R12) mentions that reducing time-to-first-response is crucial. While tooling is essential, one respondent (R4) points out that "the limiting factor is not really about tooling, but if reviewers are willing to spend the time it takes to review other people's changes."

4.5.5 Scheduling time for code reviews

Existing research shows that developers spend 6.4 hours per week on reviewing code (Bosu and Carver, 2013). That is almost 20% of the typical 40-hour work week. In our industry experience, the management often classifies the time spent performing code reviews as the "cost of doing the business." Consequently, nobody accounts for this work during the formal planning process (if any). The feedback from survey participants indicates that the time for conducting code reviews is an activity that planners need to include

in the schedule formally. Various responses demonstrate the need for better scheduling: "have dedicated time to spend on code review" (R67), "time allocated on their calendar to review the code" (R8), and "[m]ore time allocated for reviews" (R10). Planning for the code review time can (a) reduce the potential anxiety developers have, (b) expose the cost of the Modern Code Review process, and (c) help to increase code review quality because engineers can now take time to review code thoroughly. We are not aware of any organizations in the industry that use the practice of accounting for a specific amount of code review time. According to R37, "commitment from engineers and the organization" can help to improve the code velocity.

4.5.6 Too much focus on speed

An R33 provides an interesting observation: "[i]f anything development should be slowed down. Being in a (constant) hurry is a red flag." While we agree with this sentiment, we question if decreasing the code velocity is possible for commercial software development. We have rarely observed cases where industrial software projects attempt to decelerate the pace of code changes. The rare situations in which that happens falls into two categories. The first case is related to a stabilization period, such as weeks and months leading to shipping the product. A second case results from the fallout from significant events, such as discovering repeated zero-day vulnerabilities in the code.

4.5.7 Size of the code review

Other noteworthy themes include the size of the code review and communication. Requests such as "[s]maller sized patches" (R1), "smaller pieces to review" (R37), "[s]maller merge requests" (R47), "[h]ave less code to review" (R61), and "[b]reaking up the use cases into smaller chunks" (R36) indicate the desire for the upper bound of the code review size. While there is no definitive evidence to show that smaller sizes increase code velocity, the responses indicate that size is associated with overall job satisfaction. The smaller code review size is a strong preference amongst the engineers. Anecdotal guidance from FreeBSD suggests that "[t]he smaller your patch, the higher the probability that somebody will take a quick look at it" (The FreeBSD Documentation Project, 2022).

5 Discussion

We summarize the frequent themes that resulted from the analysis of the survey results. The concrete application of suggestions and potential solutions to problems depends on the context, such as corporate culture or the project stage.

Commonalities between different groups The beliefs and willingness to make trade-offs are very similar between the practitioners in the industry and the open-source software community. Therefore, whatever solutions will manifest will be helpful for both groups. The results from our survey indicate that for a majority of answers to Likert-style items, there are no statistically significant differences between industry and open-source software developers. The differences focus on career growth, job satisfaction, and what trade-offs engineers are willing to make to increase code velocity. The financial incentive structure is conceptually different between industry and largely unpaid open-source software development. We expected divergent views in these areas.

The need for speed Speed is the central theme across all the categories we cover in the survey. Engineers expect the code review process and the infrastructure to validate the code changes to be fast and responsive. Most importantly, engineers need the code reviewers to promptly pay attention to the new code changes, follow-up questions, and any other open issues. In our experience, some of these expectations and behavior are motivated by organizational culture. Companies evaluate engineers' performance using metrics such as the number of open pull requests and their age, features deployed to the production environment, and even the SLOC that an engineer has committed. These metrics can positively or negatively impact an engineer's career. Therefore, it is reasonable to assume that engineers will focus on improving these metrics.

Engineer's career and code velocity The impact of code velocity on an engineer's career is unidirectional. Actions such as missing deadlines, not completing the work items on time, or being unable to deploy the code in production to meet agreed-upon milestones negatively impact the engineer's career. The item "Career growth" ranks lowest in items that increase in code velocity impacts positively. This finding is concerning at multiple levels. Engineers can perform various actions to increase code velocity. For example, they can split their commits into isolated units, write concise commit messages, and establish an effective working relationship with their peers. All these tasks are non-trivial and take time. Based on the survey feedback, there is no objective payoff regarding career growth when engineers invest all that effort into increasing code velocity.

Splitting the code changes Previous research shows that patches introducing new features will receive slow feedback due to their size (Thongtanunam et al., 2017). There is no clear solution to mitigate this except splitting the code changes and sacrificing the cohesiveness of the code review. The current trend in the industry is to split more prominent features into smaller ones and incrementally enable a subset of functionality. However, implementation in small chunks is beneficial for only some features and products. For example, the commonly used software Microsoft Office has 12,000 feature flags (Schröder et al., 2022). Each feature flag can be enabled or disabled. It is not immediately evident that the introduction of 2^n of different configurations software can operate under is beneficial for its maintenance.

Potential for the return of the commit-then-review model We recommend that projects consider the application of the commit-then-review model under some conditions. In situations where most engineers have noticeable experience, in-depth knowledge about the product, and can make informed trade-offs, letting developers use their judgment to decide when to ask for code reviews seems a good use of time. A potential set of issues associated with this can be a decisions-process related to who is qualified enough to be permitted to act this way. For example, can only engineers at a certain level make changes without a code review? That in itself may be divisive among the engineers. Another issue could be engineers getting used to the "edit-compile-debug-commit-push" cycle and not requesting code reviews even for more extensive changes. Industry can use a process that open-source software uses to designate an individual as a committer (McKusick et al., 2015).

6 Threats to validity

Like any other study, the results we present in this paper are subject to specific categories of threats (Shull et al., 2008).

A primary threat to *internal validity* is that a survey, due to its nature, is a self-report instrument. We mitigate this threat by making the survey anonymous, indicating that all the questions are optional (except the consent), and avoiding any monetary incentives.

For conclusion validity, we rely purely on what our participants reported. Our primary data source is a collection of survey results. We rely on the correct self-identification of survey participants to draw a subset of our conclusions. To draw conclusions from our sample size and analyze the Likert-style items, we used non-parametric statistical methods recommended in the literature (Allen and Seaman, 2007; Boone, Jr. and Boone, 2012; Mann and Whitney, 1947; Shapiro and Wilk, 1965). Our survey could have reached a very homogeneous audience because we reached out to our contacts. As a result, the views expressed may be like the ones that the authors of this paper hold. We mitigated this concern by soliciting responses from Blender, Gerrit, FreeBSD, and NetBSD developer communities.

For external validity, the concern is if our findings are relevant in different contexts. Because we decided not to solicit participants based on the data mined from various source code repositories, such as GitHub or Gerrit, we could not target specific demographics precisely. We mitigated this by reaching out to several open-source software projects and our connections in the industry to solicit responses.

7 Conclusions and future work

This paper presents the qualitative results from a survey about code velocity and the beliefs and practices surrounding it. We analyzed the responses to 75 completed surveys. Ethical solicitation of survey participants was a painstaking process requiring exhaustive usage of various social media channels. Demographically, 39 participants were from the industry, and 36 respondents were from the open-source software community. Based on what we know, this is the first paper that studies the trade-offs engineers make to increase the code velocity and critical impediments that block engineers from increasing code velocity even more.

The software development processes in the industry and open-source community have conceptual differences. However, our survey suggests that most beliefs and trade-offs related to increasing code velocity in these ecosystems are similar. Engineers' critical concern is the payoff towards their career growth if code velocity improves. A controlled application of the commit-then-review model scored the highest as a potential means to increase code velocity. Reduced software security is something that 100% of open-source and 82% of industry developers will not compromise, even if it means increased code velocity.

In our future research, we plan to investigate the following topics: (a) the selective application of the commit-then-review model in the industry, (b) the benefit of reward-based incentives to motivate engineers to react faster to code reviews, and (c) the benefit of scheduling dedicated code review time to achieve a more precise planning outcome.

Data availability

The datasets generated and analyzed during the current study are available in the Zenodo repository.² The dataset includes the anonymized responses to the survey and the R script that analyzes the survey data.

Acknowledgements We thank all the survey participants for their insightful comments and suggestions. We are incredibly grateful to the Blender, Gerrit, FreeBSD, and NetBSD developer communities. Our contacts in these teams either circulated the survey internally or allowed us to use their forum platforms or mailing lists to solicit survey participants.

Conflict of interest

The authors declare that they have no conflict of interest.

References

Alami A, Cohn ML, Wąisowski A (2020) How Do FOSS Communities Decide to Accept Pull Requests? In: Proceedings of the Evaluation and Assessment in Software Engineering, Association for Computing Machinery, New York, NY, USA, EASE '20, pp 220–229, DOI 10.1145/3383219.3383242, URL https://doi.org/10.1145/3383219.3383242

Allen IE, Seaman CA (2007) Likert scales and data analyses. Quality Progress 40:64–65, URL http://rube.asq.org/quality-progress/2007/07/statistics/likert-scales-and-data-analyses.html

Armstrong K (2022) Category Direction—Code Review. URL https://about.gitlab.com/direction/create/code_review/

Bacchelli A, Bird C (2013) Expectations, Outcomes, and Challenges of Modern Code Review. In: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, ICSE '13, pp 712–721, DOI 10.1109/ICSE.2013.6606617, URL https://doi.org/10.1109/ICSE.2013.6606617

Bagert DJ (2002) Texas licensing of software engineers: All's quiet, for now. Commun ACM 45(11):92-94, DOI 10.1145/581571.581603, URL https://doi.org/10. 1145/581571.581603

Baltes S, Diehl S (2016) Worse Than Spam: Issues In Sampling Software Developers. In: Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Association for Computing Machinery, New York, NY, USA, ESEM '16, DOI 10.1145/2961111.2962628, URL https://doi.org/10.1145/2961111.2962628

Barnett M, Bird C, Brunet Ja, Lahiri SK (2015) Helping developers help themselves: Automatic decomposition of code review changesets. In: Proceedings of the 37th International Conference on Software Engineering - Volume 1, IEEE Press, Florence, Italy, ICSE '15, pp 134–144, DOI 10.1109/ICSE.2015.35, URL https://doi.org/10.1109/ICSE.2015.35

Barnett V, Lewis T (1984) Outliers in Statistical Data. Biometrical Journal 30(7):866–867, DOI 10.1002/bimj.4710300725, URL https://doi.org/10.1002/bimj.4710300725

 $^{^2}$ https://doi.org/10.5281/zenodo.7312098

- Baum T, Schneider K, Bacchelli A (2019) Associating working memory capacity and code change ordering with code review performance. Empirical Softw Engg 24(4):1762-1798, DOI 10.1007/s10664-018-9676-8, URL https://doi.org/10.1007/s10664-018-9676-8
- Baysal O, Kononenko O, Holmes R, Godfrey MW (2015) Investigating technical and non-technical factors influencing modern code review. Empirical Software Engineering 21(3):932–959, DOI 10.1007/s10664-015-9366-8, URL https://doi.org/10.1007/s10664-015-9366-8
- Beckman RJ, Cook RD (1983) Outlier.....s. Technometrics 25(2):119–149, DOI 10.1080/00401706.1983.10487840, URL http://www.tandfonline.com/doi/abs/10.1080/00401706.1983.10487840
- Bird C, Carnahan T, Greiler M (2015) Lessons Learned from Building and Deploying a Code Review Analytics Platform. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR), IEEE Computer Society, Los Alamitos, CA, USA, pp 191–201, DOI 10.1109/MSR.2015.25, URL https://doi.org/10.1109/MSR.2015.25
- Blender (2022) Code Review. URL https://wiki.blender.org/wiki/Tools/CodeReview Boone, Jr HN, Boone DA (2012) Analyzing Likert Data. Journal of Extension 50, URL https://archives.joe.org/joe/2012april/tt2.php
- Bosu A, Carver JC (2013) Impact of Peer Code Review on Peer Impression Formation: A Survey. In: 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, pp 133–142, DOI 10.1109/ESEM.2013.23, URL https://doi.org/10.1109/ESEM.2013.23
- Brown JD (2011) Likert items and scales of measurement? Shiken: JALT Testing & Evaluation SIG Newsletter 15(1):10–14, URL https://hosted.jalt.org/test/PDF/Brown34.pdf
- Carifio J, Perla RJ (2007) Ten Common Misunderstandings, Misconceptions, Persistent Myths and Urban Legends about Likert Scales and Likert Response Formats and their Antidotes. Journal of Social Sciences 3(3):106–116, URL https://thescipub.com/pdf/jssp.2007.106.116.pdf
- Chen L, Rigby PC, Nagappan N (2022) Understanding Why We Cannot Model How Long a Code Review Will Take: An Industrial Case Study. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2022, pp 1314–1319, DOI 10.1145/3540250.3558945, URL https://doi.org/10.1145/3540250.3558945
- Chen LT, Liu L (2020) Methods to Analyze Likert-Type Data in Educational Technology Research. Journal of Educational Technology Development and Exchange 13(2), DOI 10.18785/jetde.1302.04, URL https://doi.org/10.18785/jetde.1302.04
- Chouchen M, Ouni A, Olongo J, Mkaouer MW (2023) Learning to Predict Code Review Completion Time In Modern Code Review. Empirical Software Engineering 28(4):82, DOI 10.1007/s10664-023-10300-3, URL https://doi.org/10.1007/s10664-023-10300-3
- Chromium (2023) Contributing to Chromium. URL https://chromium.googlesource.com/chromium/src/+/HEAD/docs/contributing.md#Creating-a-change
- Clason D, Dormody T (1994) Analyzing Data Measured by Individual Likert-Type Items. Journal of Agricultural Education 35(4), DOI 10.5032/jae.1994.04031, URL https://doi.org/10.5032/jae.1994.04031

- Cunha AC, Conte T, Gadelha B (2021a) Code Review is just reviewing code? A qualitative study with practitioners in industry. In: Proceedings of the XXXV Brazilian Symposium on Software Engineering, Association for Computing Machinery, New York, NY, USA, SBES '21, pp 269–274, DOI 10.1145/3474624.3477063, URL https://doi.org/10.1145/3474624.3477063
- Cunha AC, Conte T, Gadelha B (2021b) What Really Matters in Code Review? A Study about Challenges and Opportunities Related to Code Review in Industry. In: XX Brazilian Symposium on Software Quality, Association for Computing Machinery, New York, NY, USA, SBQS '21, DOI 10.1145/3493244.3493255, URL https://doi.org/10.1145/3493244.3493255
- Czerwonka J, Greiler M, Tilford J (2015) Code Reviews Do Not Find Bugs. How the Current Code Review Best Practice Slows Us Down. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol 2, pp 27–28, DOI 10.1109/ICSE.2015.131, URL https://doi.org/10.1109/ICSE.2015.131
- Feitelson DG (2023) "We do not appreciate being experimented on": Developer and Researcher Views on the Ethics of Experiments on Open-Source Projects. Journal of Systems and Software 204:111774, DOI 10.1016/j.jss.2023.111774, URL https://doi.org/10.1016/j.jss.2023.111774
- Feitelson DG, Frachtenberg E, Beck KL (2013) Development and Deployment at Facebook. IEEE Internet Computing 17(4):8–17, DOI 10.1109/MIC.2013.25, URL https://doi.org/10.1109/MIC.2013.25
- Felderer M, Horta Travassos G (eds) (2020) Contemporary empirical methods in software engineering, 1st edn. Springer Nature, Cham, Switzerland
- Fowler M (2006) Continuous Integration. URL https://martinfowler.com/articles/continuousIntegration.html
- FreeBSD Foundation (2022) Obtaining Write Access to the FreeBSD Source Tree. URL https://wiki.freebsd.org/BecomingACommitter
- Frenkel S, Kang C (2021) An Ugly Truth: Inside Facebook's Battle for Domination. Harper, New York City, NY, USA
- $\label{eq:continuity} \begin{array}{llll} \mbox{GitHub} & (2021) & \mbox{Metrics} & \mbox{available} & \mbox{with} & \mbox{GitHub} & \mbox{Insights} \mbox{GitHub} \\ \mbox{Docs.} & \mbox{URL} & \mbox{https://docs.github.com/en/enterprise-server@2.21/insights/exploring-your-usage-of-github-enterprise/metrics-available-with-github-insights\#code-review-turnaround} \end{array}$
- Gold NE, Krinke J (2021) Ethics in the mining of software repositories. Empirical Software Engineering 27(1), DOI 10.1007/s10664-021-10057-7, URL https://doi.org/10.1007/s10664-021-10057-7
- Gonçalves PW, Fregnan E, Baum T, Schneider K, Bacchelli A (2020) Do explicit review strategies improve code review performance? In: Proceedings of the 17th International Conference on Mining Software Repositories, Association for Computing Machinery, New York, NY, USA, MSR '20, pp 606–610, DOI 10.1145/3379597.3387509, URL https://doi.org/10.1145/3379597.3387509
- Gonzalez-Barahona JM (2020) Mining Software Repositories While Respecting Privacy. URL https://2020.msrconf.org/details/msr-2020-Education/1/Mining-Software-Repositories-While-Respecting-Privacy
- Google (2023a) Google Engineering Practices Documentation. URL https://google.github.io/eng-practices/
- Google (2023b) Speed of Code Reviews. URL https://google.github.io/eng-practices/review/reviewer/speed.html

- Greiler M (2020) Code Reviews—From Bottleneck to Superpower with Michaela Greiler. URL https://learning.acm.org/techtalks/codereviews
- Groves RM (2006) Nonresponse rates and nonresponse bias in household surveys. Public Opinion Quarterly 70(5):646–675, DOI 10.1093/poq/nfl033, URL https://doi.org/10.1093/poq/nfl033
- Hong Y, Tantithamthavorn CK, Thongtanunam PP (2022) Where Should I Look at? Recommending Lines that Reviewers Should Pay Attention To. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 1034–1045, DOI 10.1109/SANER53432.2022.00121, URL https://doi.org/10.1109/SANER53432.2022.00121
- Izquierdo-Cortazar D, Sekitoleko N, Gonzalez-Barahona JM, Kurth L (2017) Using Metrics to Track Code Review Performance. In: Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering, Association for Computing Machinery, Karlskrona, Sweden, EASE'17, pp 214–223, DOI 10.1145/ 3084226.3084247, URL https://doi.org/10.1145/3084226.3084247
- Jiang Y, Adams B, German DM (2013) Will my patch make it? And how fast?: Case study on the Linux kernel. In: Proceedings of the 10th Working Conference on Mining Software Repositories, IEEE Press, MSR '13, pp 101–110, DOI 10.1109/MSR.2013. 6624016, URL https://doi.org/10.1109/MSR.2013.6624016
- Killalea T (2019) Velocity in software engineering. Commun ACM 62(9):44-47, DOI 10.1145/3345626, URL https://doi.org/10.1145/3345626
- Kim H, Kwon Y, Joh S, Kwon H, Ryou Y, Kim T (2022) Understanding automated code review process and developer experience in industry. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2022, pp 1398–1407, DOI 10.1145/3540250.3558950, URL https://doi.org/10.1145/3540250.3558950
- Kononenko O, Baysal O, Godfrey MW (2016) Code Review Quality: How Developers See It. In: Proceedings of the 38th International Conference on Software Engineering, Association for Computing Machinery, Austin, Texas, ICSE '16, pp 1028–1038, DOI 10.1145/2884781.2884840, URL https://doi.org/10.1145/2884781.2884840
- Kononenko O, Rose T, Baysal O, Godfrey MW, Theisen D, de Water B (2018) Studying Pull Request Merges: A Case Study of Shopify's Active Merchant. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, Association for Computing Machinery, New York, NY, USA, ICSE-SEIP '18, pp 124–133, DOI 10.1145/3183519.3183542, URL https://doi.org/10.1145/3183519.3183542
- Kushner D (2011) Facebook philosophy: move fast and break things. URL https://spectrum.ieee.org/facebook-philosophy-move-fast-and-break-things
- Liddell TM, Kruschke JK (2018) Analyzing ordinal data with metric models: What could possibly go wrong? Journal of Experimental Social Psychology 79:328–348, DOI 10.1016/j.jesp.2018.08.009, URL https://doi.org/10.1016/j.jesp.2018.08.009
- $\label{linux} Linux~(2023)~Everything~you~ever~wanted~to~know~about~Linux~-stable~releases.~URL~https://www.kernel.org/doc/html/v4.15/process/stable-kernel-rules.html$
- Llull R (1988) Blanquerna, 2nd edn. Dedalus European classics, Dedalus Hippocrene books, Sawtry, Cambs, United Kingdom
- LLVM Foundation (2023a) Contributing to LLVM—LLVM 12 documentation. URL https://llvm.org/docs/Contributing.html#format-patches

- LLVM Foundation (2023b) LLVM code-review policy and practices. URL https://llvm.org/docs/CodeReview.html
- MacLeod L, Greiler M, Storey MA, Bird C, Czerwonka J (2018a) Code reviewing in the trenches: Challenges and best practices. IEEE Software 35(4):34–42, DOI 10.1109/MS.2017.265100500
- MacLeod L, Greiler M, Storey MA, Bird C, Czerwonka J (2018b) Code Reviewing in the Trenches: Challenges and Best Practices. IEEE Software 35(4):34–42, DOI 10.1109/MS.2017.265100500, URL https://doi.org/10.1109/MS.2017.265100500
- Maddila C, Upadrasta SS, Bansal C, Nagappan N, Gousios G, Deursen Av (2022) Nudge: Accelerating Overdue Pull Requests Towards Completion. ACM Trans Softw Eng Methodol DOI 10.1145/3544791, URL https://doi.org/10.1145/3544791
- Mann HB, Whitney DR (1947) On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. The Annals of Mathematical Statistics 18(1):50–60, DOI 10.1214/aoms/1177730491, URL http://doi.org/10.1214/aoms/1177730491
- Martin RC (2002) Agile Software Development, Principles, Patterns, and Practices. Alan Apt series, Pearson, Upper Saddle River, NJ, USA
- McCarthy J (1995) Dynamics of software development. Microsoft Press, Redmond, Wash
- McIntosh S, Kamei Y, Adams B, Hassan AE (2015) An empirical study of the impact of modern code review practices on software quality. Empirical Software Engineering 21(5):2146–2189, DOI 10.1007/s10664-015-9381-9, URL https://doi.org/10.1007/s10664-015-9381-9
- McKusick MK, Neville-Neil GV, Watson RNM (2015) The Design and Implementation of the FreeBSD Operating System, 2nd edn. Addison Wesley, Upper Saddle River, NJ, USA
- McMartin A (2021) Introducing Developer Velocity Lab—A Research Initiative to Amplify Developer Work and Well-Being. URL https://techcommunity.microsoft.com/t5/azure-developer-community-blog/introducing-developer-velocity-lab-a-research-initiative-to/ba-p/2333140
- Microsoft Research (2019) 14th IEEE/ACM International Workshop on Automation of Software Test. URL https://www.microsoft.com/en-us/research/event/14th-ieee-acm-international-workshop-on-automation-of-software-test/
- Microsoft Research (2023) Developer Velocity Lab. URL https://www.microsoft.com/en-us/research/group/developer-velocity-lab/
- Miller GA (1956) The magical number seven, plus or minus two: Some limits on our capacity for processing information. Psychological Review 63(2):81–97, DOI 10.1037/h0043158, URL https://doi.org/10.1037/h0043158
- Mozilla (2023) Code reviews—Firefox Source Docs documentation. URL https://firefox-source-docs.mozilla.org/devtools/contributing/code-reviews.html
- Nazir S, Fatima N, Chuprat S (2020) Modern code review benefits-primary findings of a systematic literature review. In: Proceedings of the 3rd International Conference on Software Engineering and Information Management, Association for Computing Machinery, New York, NY, USA, ICSIM '20, pp 210–215, DOI 10.1145/3378936. 3378954, URL https://doi.org/10.1145/3378936.3378954
- Palantir (2018) Code review best practices. URL https://blog.palantir.com/code-review-best-practices-19e02780015f
- Phabricator (2021) Writing Reviewable Code. URL https://secure.phabricator.com/book/phabflavor/article/writing_reviewable_code/#many-small-commits

- PostgreSQL (2019) Submitting a Patch PostgreSQL wiki. URL https://wiki.postgresql.org/wiki/Submitting a Patch
- Raina S (2015) Establishing association. Indian Journal of Medical Research 141(1):127, DOI 10.4103/0971-5916.154519, URL http://doi.org/10.4103/0971-5916.154519
- Rigby PC, Bird C (2013) Convergent contemporary software peer review practices. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2013, pp 202–212, DOI 10.1145/2491411.2491444, URL https://doi.org/10.1145/2491411.2491444
- Rigby PC, German DM (2006) A preliminary examination of code review processes in open source projects. Tech. rep., Concordia University, URL https://users.encs.concordia.ca/~pcr/paper/Rigby2006TechReport.pdf
- Rigby PC, German DM, Storey MA (2008) Open Source Software Peer Review Practices: A Case Study of the Apache Server. In: Proceedings of the 30th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '08, pp 541–550, DOI 10.1145/1368088.1368162, URL https://doi.org/10.1145/1368088.1368162
- Riggs P (2022) Move faster, wait less: Improving code review time at Meta. URL https://engineering.fb.com/2022/11/16/culture/meta-code-review-time-improving/
- Sadowski C, Söderberg E, Church L, Sipko M, Bacchelli A (2018) Modern Code Review: A Case Study at Google. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, Association for Computing Machinery, Gothenburg, Sweden, ICSE-SEIP '18, pp 181–190, DOI 10.1145/3183519. 3183525, URL https://doi.org/10.1145/3183519.3183525
- dos Santos EW, Nunes I (2017) Investigating the Effectiveness of Peer Code Review in Distributed Software Development. In: Proceedings of the XXXI Brazilian Symposium on Software Engineering, Association for Computing Machinery, New York, NY, USA, SBES '17, pp 84–93, DOI 10.1145/3131151.3131161, URL https://doi.org/10.1145/3131151.3131161
- Schröder M, Kevic K, Gopstein D, Murphy B, Beckmann J (2022) Discovering Feature Flag Interdependencies in Microsoft Office. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2022, pp 1419–1429, DOI 10.1145/3540250.3558942, URL https://doi.org/10.1145/3540250.3558942
- Shan Q, Sukhdeo D, Huang Q, Rogers S, Chen L, Paradis E, Rigby PC, Nagappan N (2022) Using Nudges to Accelerate Code Reviews at Scale. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2022, pp 472–482, DOI 10.1145/3540250.3549104, URL https://doi.org/10.1145/3540250.3549104
- Shapiro SS, Wilk MB (1965) An analysis of variance test for normality (complete samples). Biometrika 52(3-4):591–611, DOI 10.1093/biomet/52.3-4.591, URL https://doi.org/10.1093/biomet/52.3-4.591
- Shull F, Singer J, Sjøberg DIK (2008) Guide to Advanced Empirical Software Engineering. Springer, London
- Smith MG, Witte M, Rocha S, Basner M (2019) Effectiveness of incentives and follow-up on increasing survey response rates and participation in field studies.

- BMC Medical Research Methodology 19(1), DOI 10.1186/s12874-019-0868-8, URL $\frac{\text{https:}}{\text{doi.org}} = \frac{10.1186}{\text{s}} = \frac{12874-019-0868-8}{\text{o}}$
- Söderberg E, Church L, Börstler J, Niehorster D, Rydenfält C (2022) Understanding the Experience of Code Review: Misalignments, Attention, and Units of Analysis. In: Proceedings of the International Conference on Evaluation and Assessment in Software Engineering 2022, Association for Computing Machinery, New York, NY, USA, EASE '22, pp 170–179, DOI 10.1145/3530019.3530037, URL https://doi.org/10.1145/3530019.3530037
- Storey MA, Houck B, Zimmermann T (2022) How developers and managers define and trade productivity for quality. In: Proceedings of the 15th International Conference on Cooperative and Human Aspects of Software Engineering, Association for Computing Machinery, New York, NY, USA, CHASE '22, pp 26–35, DOI 10.1145/3528579.3529177, URL https://doi.org/10.1145/3528579.3529177
- Synopsys, Inc (2020) The Heartbleed Bug. URL https://heartbleed.com/
- Tan X, Zhou M (2019) How to Communicate When Submitting Patches: An Empirical Study of the Linux Kernel. Proc ACM Hum-Comput Interact 3(CSCW), DOI 10. 1145/3359210, URL https://doi.org/10.1145/3359210
- Tanna J (2021) Improving Team Efficiency by Measuring and Improving Code Review Cycle Time. URL https://www.jvt.me/posts/2021/10/27/measure-code-review/
- The FreeBSD Documentation Project (2022) Committer's guide. URL https://docs.freebsd.org/en/articles/committers-guide/#pre-commit-review
- The Linux Foundation (2022) A beginner's guide to Linux kernel development. URL https://trainingportal.linuxfoundation.org/learn/course/a-beginners-guide-to-linux-kernel-development-lfd103/
- Thongtanunam P, Tantithamthavorn C, Kula RG, Yoshida N, Iida H, Matsumoto Ki (2015) Who should review my code? A file location-based code-reviewer recommendation approach for Modern Code Review. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp 141–150, DOI 10.1109/SANER.2015.7081824, URL https://doi.org/10.1109/SANER.2015.7081824
- Thongtanunam P, Mcintosh S, Hassan AE, Iida H (2017) Review participation in modern code review. Empirical Softw Engg 22(2):768–817, DOI 10.1007/s10664-016-9452-6, URL https://doi.org/10.1007/s10664-016-9452-6
- Tsay JT (2017) Software Developers Using Signals in Transparent Environments. PhD thesis, Carnegie Mellon University, DOI 10.1184/R1/6723026.v1, URL https://doi.org/10.1184/R1/6723026.v1
- Vanian J (2022) Internal Facebook memo warns company must be disciplined, prioritize ruthlessly. URL https://www.cnbc.com/2022/06/30/internal-facebook-memo-warns-company-must-be-disciplined-prioritize.html
- Weißgerber P, Neu D, Diehl S (2008) Small patches get in! In: Proceedings of the 2008 international working conference on Mining software repositories, Association for Computing Machinery, Leipzig, Germany, MSR '08, pp 67–76, DOI 10.1145/1370750.1370767, URL https://doi.org/10.1145/1370750.1370767
- Winters T, Manshreck T, Wright H (2020) Software Engineering at Google: Lessons Learned from Programming Over Time, 1st edn. O'Reilly, Beijing Boston Farnham Sebastopol Tokyo
- Wu Q, Lu K (2021) On the feasibility of stealthily introducing vulnerabilities in open-source software via hypocrite commits. URL https://github.com/QiushiWu/QiushiWu.github.io/blob/main/papers/OpenSourceInsecurity.pdf

Zanjani MB, Kagdi H, Bird C (2016) Automatically Recommending Peer Reviewers in Modern Code Review. IEEE Transactions on Software Engineering 42(6):530–543, DOI 10.1109/TSE.2015.2500238, URL https://doi.org/10.1109/TSE.2015.2500238

Zhu J, Zhou M, Mockus A (2016) Effectiveness of Code Contribution: From Patch-Based to Pull-Request-Based Tools. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, FSE 2016, pp 871–882, DOI 10.1145/2950290.2950364, URL https://doi.org/10.1145/2950290.2950364