Does Code Review Speed Matter for Practitioners?

Kudrjavets, Gunnar; Rastogi, Ayushi

Published in:
Empirical software engineering

DOI:
10.1007/s10664-023-10401-z

Publication date:
2023

Empirical Software Engineering (2024) 29:7
https://doi.org/10.1007/s10664-023-10401-z

Does code review speed matter for practitioners?

Gunnar Kudrjavets[1] · **Ayushi Rastogi**[1]

Accepted: 2 October 2023

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 qualitative survey to study the code velocity-related beliefs and practices in place. We analyzed 75 completed surveys from SurIndustryDevs 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 are divided about the benefits of increased code velocity for their career growth, (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 is a time-consuming part of evaluating the quality

Communicated by: Tayana Conte

✉ Gunnar Kudrjavets
g.kudrjavets@rug.nl

Ayushi Rastogi
a.rastogi@rug.nl

[1] Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence, University of Groningen, 9712 CP Groningen, Netherlands

Published online: 22 November 2023

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).

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. An accepted definition of *code velocity* is "the time between making a code change and shipping the change to customers" (Microsoft Research 2019). This paper focuses on code review s and the total duration of code review completion and merge time. The "customers" are other engineers, and "shipping" means a code review was accepted and committed. We include a detailed description of related terminology in Section 2.3.

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. We describe the work related to increasing code velocity in Section 2.2. 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?

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 review s 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 (c) respondents mainly focused on the development process, infrastructure and tooling support, response time, and the need to schedule pre-allocated time for code review s 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 review ers, 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 8500 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 review s 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 under-

stand 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 Related Work

The industry mainly drives the research related to increasing the code velocity. The default assumptions in the industry are that increased code velocity is desirable, and the code review s can never be fast enough (Riggs 2022; Killalea 2019; Greiler 2020). The a priori assumption is that the code review speed *does matter* for practitioners.

One focus area in the research related to code velocity is the ability to predict the duration of code review s. Existing research has resulted in contradicting findings. An industrial case study from Meta finds that core code review characteristics "did not provide substantial predictive power" (Chen et al. 2022). However, a study based on Gerrit code review s finds that "ML models significantly outperform baseline approaches with a relative improvement ranging from 7% to 49%" (Chouchen et al. 2023).

Another part of the research focuses on various tools and techniques to speed up code review s. These approaches include optimizing the code review strategy (Gonçalves et al. 2020), investigating the effectiveness of bot usage to automate the code review process (Kim et al. 2022), periodically reminding engineers to make progress with their code reviews (Maddila et al. 2022; Shan et al. 2022), prioritizing the subsets of code review that need attention (Hong et al. 2022), targeting the optimal reviewers (Thongtanunam et al. 2015), and improving automation to suggest reviewers (Zanjani et al. 2016).

We are unaware of any research about various compromises that engineers are willing to make to improve code velocity or explicit ways to improve code velocity. The closest to our research is a paper that investigates the "misalignments in the code review tooling and process" (Söderberg et al. 2022) and papers about what factors impact the code review decisions (Kononenko et al. 2016, 2018).

2.3 Terminology and Metrics

2.3.1 Overview of Terminology

Most commercial organizations share a similar goal: *reduce the duration of code review s 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".

2.3.2 Code Review Metrics

In this paper, we choose the following metrics to characterize code review process:

- **Time-to-first-response** (Bird et al. 2015; MacLeod et al. 2018a): the time from publishing the code review until the first acceptance, comment, inline comment (comment on specific code fragments), rejection, or any other activity by a person other than the author of the code. Survey participants at Microsoft indicated that "[r]eceiving feedback in a timely manner" is the highest-ranked challenge that engineers face during the code review process (MacLeod et al. 2018a). We exclude any code review-related activity by bots.
- **Time-to-accept** (Bird et al. 2015): the time from when an engineer publishes code changes for review until someone other than the author accepts the code review. This term is more precise than *time to completion* (Bird et al. 2015). Similarly, we exclude any code review-related activity by bots.
- **Time-to-merge** (GitHub 2021; Kononenko et al. 2016): encompasses the entire duration of the code review process. We define the time-to-merge as "…the time since the proposal of a change (…) to the merging in the code base …" (Izquierdo-Cortazar et al. 2017).

2.4 Expectations Related to Code Velocity

We find that *expected code review response times between industry and various open-source 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 review s 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 review s 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 essential questions. The survey starts with a question that asks participants for consent. The survey finishes with a question allowing participants to share their email addresses if researchers can contact them to share the findings from the survey. All the questions except the one that determined the participants' consent were *optional*. We display the shortened version of the survey questions in Table 1. The entire survey contents are part of Appendix A.

The first eight questions related to participants' demographics, such as experience with software development, code review s, 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.

Table 1 List of survey questions

Number	Question text
Q1	Ask for participant's consent (mandatory).
Q2	How many years of experience do you have with collaborative software development?
Q3	How many years have you been reviewing other people's code?
Q4	What is your role in the code review (e.g., diff, patch, pull request) process? Select all that apply.
Q5	How many times in a month do you submit code changes for review?
Q6	How many times in a month do you review other people's code changes?
Q7	What code reviewing environments do you use?
Q8	What type of software developer are you?
Q9	Choose an application domain you are most experienced with for the remaining questions?
Q10	Rank the following metrics in the order of importance to optimize code velocity.
Q11	Velocity of your code improves your …
Q12	I am willing to compromise on …if it improves code velocity.
Q13	I think the post-commit review model (changes are first committed and then reviewed at some point later) can improve …
Q14	Should engineers be allowed to commit their code without the code review depending on …
Q15	In your opinion, what is the maximum acceptable size of the code review?
Q16	What is a desired time range for someone to either accept your code review or give you detailed feedback?
Q17	In your opinion, how can code velocity be improved for your projects?
Q18	Ask for an email address.

All the questions except Q1 are optional

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]

[1] https://doi.org/10.5281/zenodo.8242289

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 review s 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 goal was to reach a heterogenous collection of practitioners. We used our industry connections to target professional engineers and hobbyists, industry and open-source software developers, industry veterans, and junior software developers. We used our academic connections to share the survey with software engineering researchers and computer science students who participated in the code review process. Geographically, most of our contacts reside in Europe and the United States.

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. We then used the breadth-first approach to share the survey invitation on social media. Our target platforms were Facebook, LinkedIn, Medium, Reddit, and Twitter (X).

Our approach did not use the snowball (chain-referral) sampling strategy to recruit additional survey participants. Snowball sampling is an approach where already recruited study participants recruit new people to participate in the survey. We intended to avoid the community bias and "the lack of definite knowledge as to whether or not the sample is an accurate reading of the target population" (Raina 2015).

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, Free BSD, and Net BSD. 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 Sections 4.3 and 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. Consequently, this experience translates to the time spent reviewing other people's code. We present the participants' development and code-reviewing experience in Table 2.

When it comes to the number of code review s that participants perform:

- 36% of respondents submit more than 10 code review s monthly, and 43% submit 3–10 code review s.
- 54% of respondents conduct more than 10 code review s monthly, and 38% conduct 3–10 code review s.

Table 2 Survey participant demographics

Development experience	Count	Reviewing experience	Count
< 3 years	3 (4%)	< 1 year	4 (5%)
3–10 years	25 (33%)	1–5 years	20 (27%)
> 10 years	45(60%)	5–10 years	11 (15%)
Unknown	2 (3%)	> 10 years	35 (46%)
		Does not review code	2 (3%)
		Unknown	3 (4%)

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 review s 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 review s 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 review s. 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 1000 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).

We asked study participants about the maximum acceptable number of SLOC and the number of files in the code review. Figures 1 and 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 $U(N_{\text{Industry}} = 33, N_{\text{OSS}} = 26) = 979.5$, $z = -0.16$, $p = .87$. Similarly, we do not observe differences for the number of files $U(N_{\text{Industry}} = 34, N_{\text{OSS}} = 26) = 1088.5$, $z = 0.78$, $p = .44$.

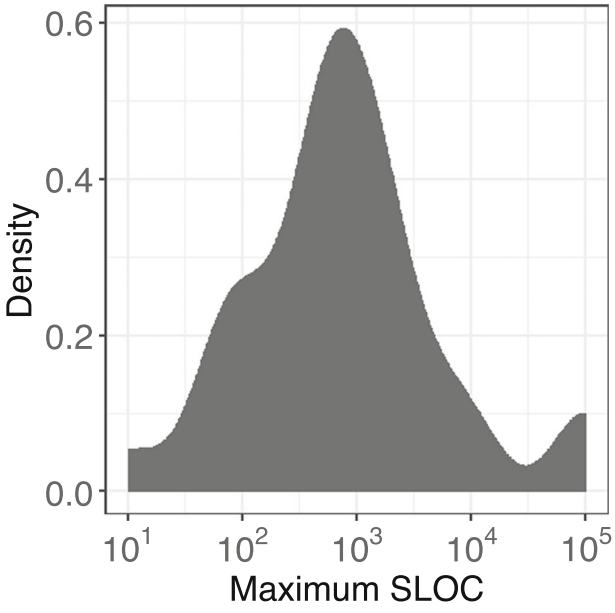

Fig. 1 Source lines of code

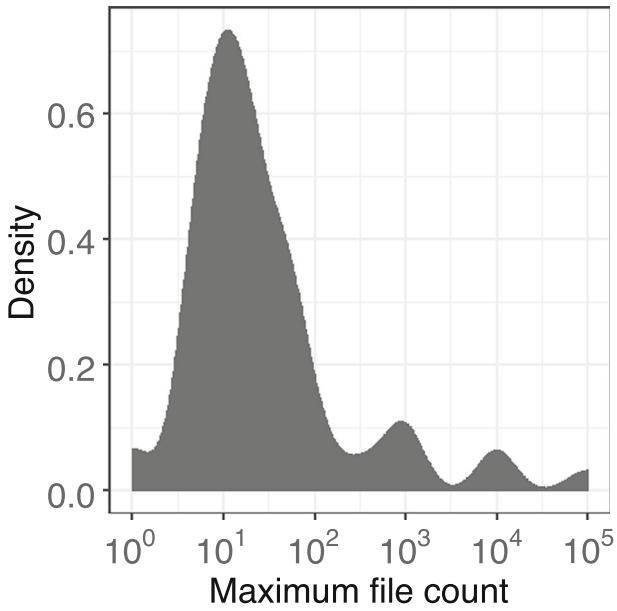

Fig. 2 File count

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.

Section 4.3.1 discussed the ambiguity of expectations related to an acceptable code review size. One of the problems that we have witnessed in practice is that engineers who switch from one project or team to another will have to adjust to a new definition of *small*. Having to invest time into manually splitting their changes differently to appease reviewers can decrease developer productivity. The existing research proposes solutions to automatically decompose the code review s into smaller changesets (Barnett et al. 2015). However, we are not aware of any project that actively uses this approach or a code collaboration environment that effectively supports splitting code review s. As an early warning mechanism, we propose that various tools used during the code review process warn engineers when they exceed the limit for a particular project. Early notification will prevent the unnecessary iteration between a submitter and a reviewer, where the reviewer will request that the submitter split the changes into smaller chunks.

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 3.

Table 3 Rankings of different code review periods in the order of importance to optimize for code velocity

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

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

Our ranking method that results in the data in Table 3 is subjective. We will also use the Borda count method to objectively evaluate how participants ranked different code review periods (Llull 1988). The Borda count method is a ranked election system. In the case of N candidates, the vote for the first place is worth N points, the vote for the second place is worth $N - 1$ points, the vote for the third place is worth $N - 2$ points, and the pattern continues similarly. The vote for the last place is worth 1 point. According to the Borda count, time-to-merge receives 202 points, time-to-first-response receives 190 points, time-to-accept receives 187 points, and other metrics receive 71 points.

In 49% of cases, the time-to-merge was ranked as the first priority metric to optimize. Similarly, time-to-merge wins the Borda count. 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 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 review s 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.

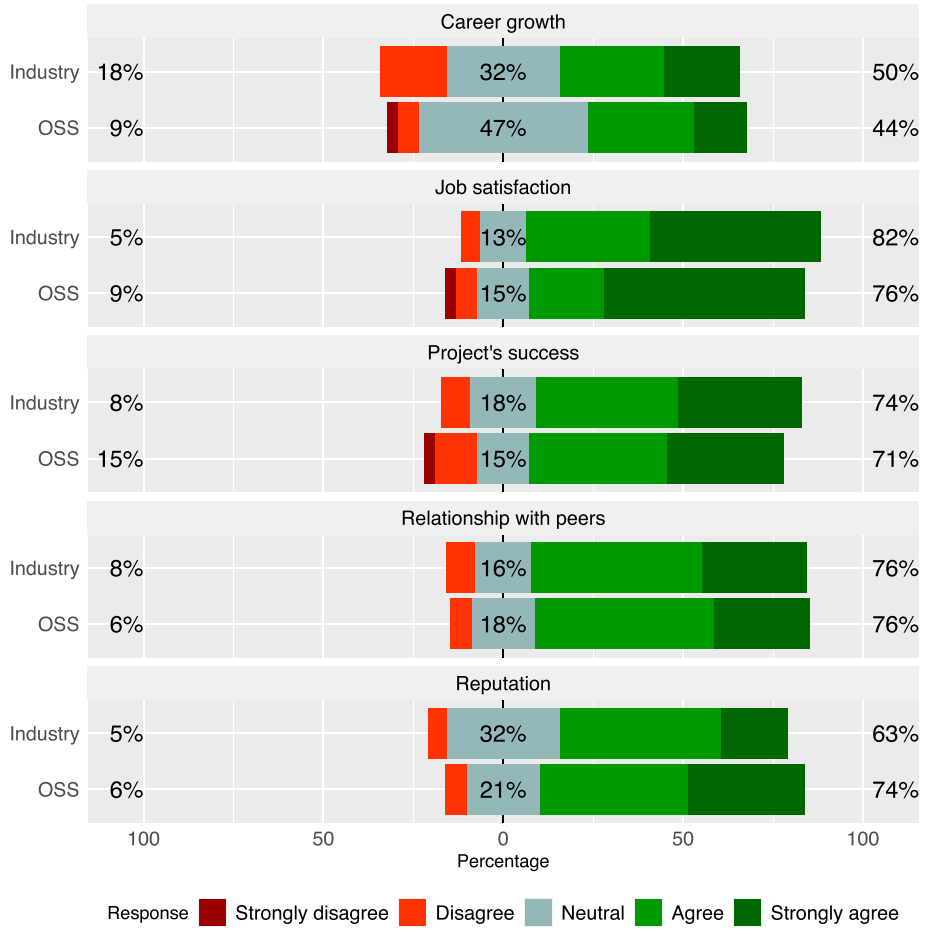

Fig. 3 Likert scales for the Q11 ("Velocity of your code improves your …")

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.

Observation 2 serves as a motivating factor behind investments in the developer infrastructure to increase the code velocity. This finding confirms our anecdotal observations from industry that "engineers are happier when they can commit their code changes fast". Similarly, the data from Meta shows that increase in code review time causes engineers to be less satisfied with the code review process (Riggs 2022). We recommend that projects include the metrics related to code velocity as one of the indicators of organizational health. Historically, Microsoft used the success of the daily build to indicate the project's health and overall state (McCarthy 1995). With the prevalence of CI and CD, metrics such as median daily time-to-merge can be a suitable replacement.

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-then-review 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 review s 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 $U\,(N_{\text{Industry}} = 39, N_{\text{OSS}} = 34) = 1664, z = 2.65, p = .008$. Similarly, for "Security", a Mann-Whitney U test indicated that the difference in mean ranks is statistically significant $U\,(N_{\text{Industry}} = 39, N_{\text{OSS}} = 34) = 1621, z = 2.33, p = .02$.

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

Observation 3

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

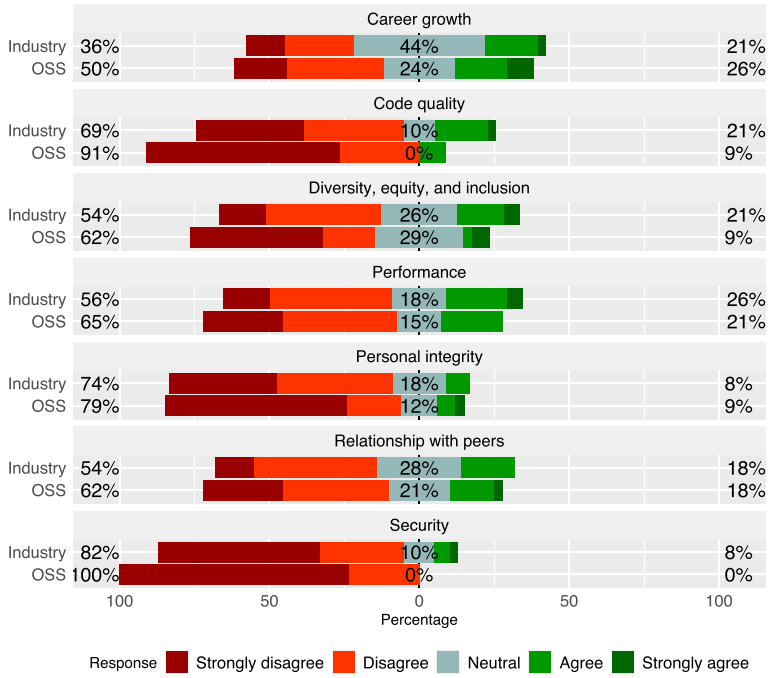

Fig. 4 Likert scales for the Q12 ("I am willing to compromise on … if it improves code velocity")

Critical security vulnerabilities such as the OpenSSL Heartbleed bug (Synopsys, Inc. 2020) could have contributed to the increased awareness that a mistake in a single line of code can cause catastrophic damage. This finding is very positive, given the impact of zero-day vulnerabilities and society's increased reliance on software. We interpret it to mean that even given external factors such as deadlines, pressure from the author of the code, or project needs, the engineers are not willing to compromise on security. There is no equivalent of the Hippocratic Oath for software engineers, and licensing software engineers is controversial (Bagert 2002). However, this finding indicates the presence of an internal code of conduct of "do no harm" that engineers strive to follow.

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 Fig. 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 $U(N_{\text{Industry}} = 39, N_{\text{OSS}} = 33) = 1603.5, z = 2.12, p = .034$. The code velocity is also a category that most participants thought could be improved. Of industry respondents, 64% gave a positive response, with

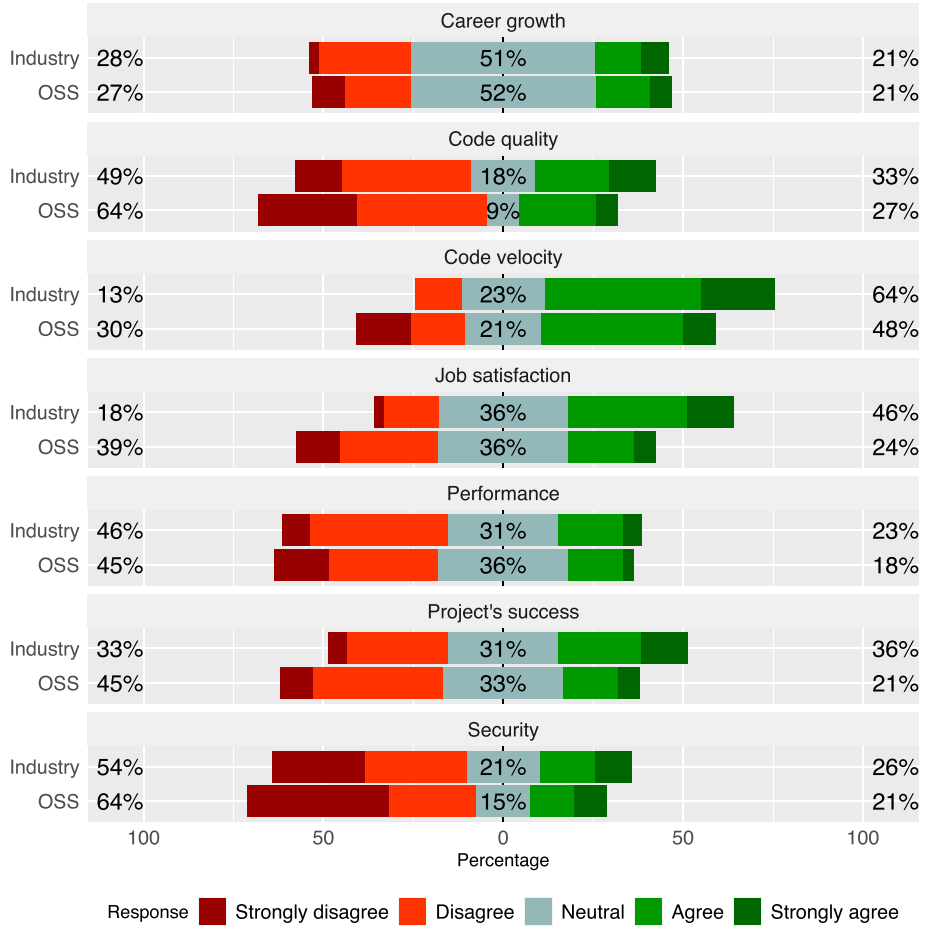

Fig. 5 Likert scales for the Q13 ("I think the post-commit review model (changes are first committed and then reviewed at some point later) can improve …")

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 $U\,(N_{\text{Industry}} = 39, N_{\text{OSS}} = 33) = 1620, z = 2.28, p = .023$. 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-then-review 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.

4.4.2 Abandonment of Code Review s in Favor of Code Velocity

Code reviews are optional in some contexts. For example, the Free BSD project defines a committer as "an individual with *write access* to the Free BSD 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 Dragon Fly BSD, Free BSD, Net BSD, and Open BSD, we decided to survey what engineers think about the possibility of abandoning code review s. We asked participants under what conditions engineers can commit code without someone else reviewing that code. We display the results in Fig. 6 and discuss them below.

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.

Observation 4 implies that even given the industry's relentless drive toward increasing the code velocity, engineers do not perceive the model where *anyone* can commit code freely as something genuinely beneficial. At the same time, the committer model that Dragon Fly BSD, Free BSD, Net BSD, and Open BSD use shows that there is a point where an engineer can "graduate" to a level where they are no longer required to have their code reviewed. A worthwhile research avenue is to investigate and estimate the cost of becoming a committer in one of the open-source software projects. For example, the Free BSD guidelines (FreeBSD Foundation 2022) say that "[t]he process to gain FreeBSD source access is a long one and "[i]t can take months of contributions and interaction with the project". Is the cost-benefit still there for the engineers who become committers and maintain the committer status?

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.

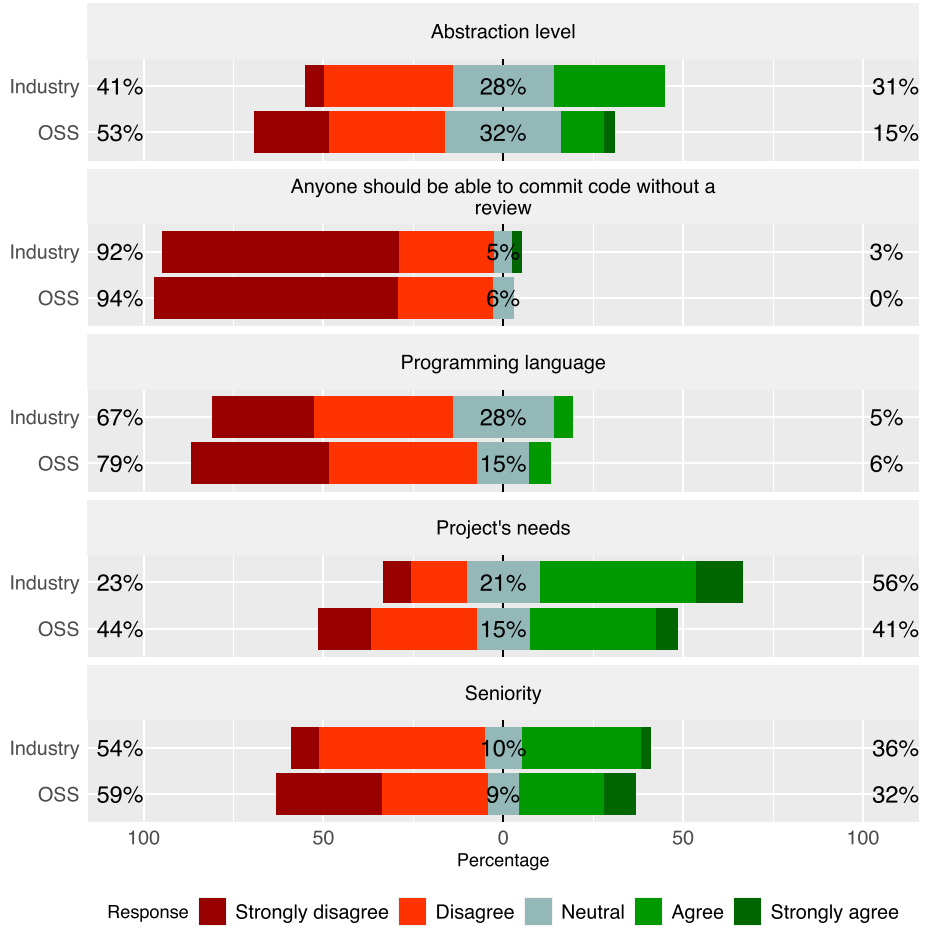

Fig. 6 Likert scales for the Q14 ("Should engineers be allowed to commit their code without the code review depending on …")

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 review s 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 Fig. 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 4.

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 Ri 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".

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

Table 4 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

4.5.2 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 review s: "[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 review s 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 review s 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 review s 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 (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 review ers 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 12000 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 review s 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 review

s 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, Free BSD, and Net BSD developer communities.

Another threat in this category relates to *readability*. While the intent of the questions may be apparent to the authors, they may be ambiguous to participants. We tried to mitigate this threat by asking a native speaker to review the questions and adjust them based on the feedback from a pilot group of participants. Similarly, there are no assurances that all the survey participants are native English speakers. Therefore, they can misinterpret some of the questions or terminology. For example, the terms like "Career growth" and "Job satisfaction" can have different meanings depending on the context and individual. We tried to mitigate this threat by composing pointed questions and highlighting the key terms. We did not provide formal definitions for each term to limit the survey's verbosity and attempt to increase the completion rate.

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[2] 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.

Appendix A: Survey Questionnaire

A.1 Q1

Dear participant,

This survey solicits your beliefs and experiences about code velocity (the speed with which code changes are reviewed and merged) as a software engineer. The results from this survey will help to improve developer productivity and the daily lives of large numbers of software engineers. The survey should take approximately 5–7 minutes to complete.

For questions, contact Gunnar Kudrjavets (g.kudrjavets@rug.nl) and Dr. Ayushi Rastogi (a.rastogi@rug.nl) at the University of Groningen.

You may refuse to take part in the research or exit the survey at any time without penalty. You may skip any question you do not wish to answer for any reason. Please refer to the link below for more details about the participation.

Consent form for code velocity survey.pdf.

Choosing "Agree" indicates that

- You have read the above information
- You voluntarily agree to participate
- You are 18 years of age or older

◯ Agree ◯ Disagree

A.2 Q2

How many **years of experience** do you have with **collaborative software development**?

◯ < 3 year
◯ 3–10 years
◯ > 10 years

A.3 Q3

How many **years** have you been **reviewing other people's code**?

[2] https://doi.org/10.5281/zenodo.8242289

◯ < 1 year
◯ 1 to 5 years
◯ 5 to 10 years
◯ > 10 years
I do not review code (e.g., only contribute code, moderate code review discussions)

A.4 Q4

What is **your role** in the code review (e.g., diff, patch, pull request) process? Select all that apply.

☐ Author (e.g., contribute to a software project, fix a bug, implement a feature)
☐ Moderator (e.g., resolve conflicts between participants, make final decisions)
☐ Reviewer (e.g., a maintainer for a subsystem, owner of a feature)
☐ Other, please specify _____

A.5 Q5

How many **times in a month** do you submit **code changes for review**?

◯ 1–2 times
◯3–6 times
◯6–10 times
◯10+ times
None, my role is different (e.g., I only review code). Please specify: _____

A.6 Q6

How many **times in a month** do you **review other people's code changes**?

◯1–2 times
◯3–6 times
◯6–10 times
◯10+ times
None, my role is different (e.g., I only submit patches. Please specify: _____

A.7 Q7

What **code reviewing environments** do you **use**? Select all that apply.

☐ Code collaboration tools (e.g., Gerrit, GitHub, Phabricator)
☐ Tools internal to a company (e.g., Google's Critique, Meta's Phabricator, Microsoft's CodeFlow)
☐ Mailing lists to review diffs/patches
☐ Pair programming
☐ Other, please specify _____

A.8 Q8

What **type of software developer** are you? Select all that apply.

☐ I work on closed-source and get paid
☐ I work on open-source and get paid
☐ I work on open-source and **do not get** paid
☐ Other, please specify _____

A.9 Q9

Choose an **application domain** you are **most experienced** with for the **remaining questions**?

◯ Application software (e.g., mobile and desktop applications)
◯ Systems software (e.g., drivers, kernel development)
◯ Real-time or critical software (e.g., aeronautics, embedded systems)
◯ Other, please specify _____

A.10 Q10

Rank the following **metrics** in the order of importance to **optimize code velocity**. Drag the items ↑ or ↓ with 1 being the most important metric.

1. **Time-to-first-response** (first indication when someone reacted to your code review or provided any actionable feedback)
2. **Time-to-accept** (someone formally accepted your code changes and the code changes can now be officially committed and merged)
3. **Time-to-merge** (the accepted code changes have finally ended up in the destination branch)
4. Other, please specify _____

A.11 Q11

Velocity of your code **improves** your . . .

Table 5 Survey question Q11

	Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree	I don't know
Project's success	◯	◯	◯	◯	◯	◯
Career growth	◯	◯	◯	◯	◯	◯
Relationship with peers	◯	◯	◯	◯	◯	◯
Job satisfaction	◯	◯	◯	◯	◯	◯
Reputation	◯	◯	◯	◯	◯	◯

A.12 Q12

I am willing to **compromise** on …if it **improves** code **velocity**.

Table 6 Survey question Q12

	Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree	I don't know
Performance	○	○	○	○	○	○
Code quality	○	○	○	○	○	○
Security	○	○	○	○	○	○
Relationship with peers	○	○	○	○	○	○
Career growth	○	○	○	○	○	○
Personal integrity	○	○	○	○	○	○
Diversity, equity, and inclusion	○	○	○	○	○	○

A.13 Q13

I think the **post-commit review** model (changes are first committed and then reviewed at some point later) can **improve** …

Table 7 Survey question Q13

	Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree	I don't know
Code velocity	○	○	○	○	○	○
Code quality	○	○	○	○	○	○
Performance	○	○	○	○	○	○
Security	○	○	○	○	○	○
Career growth	○	○	○	○	○	○
Project's success	○	○	○	○	○	○
Job satisfaction	○	○	○	○	○	○

A.14 Q14

Should **engineers** be allowed to commit their code **without the code review** depending on …

Table 8 Survey question Q14

	Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree	I don't know
Seniority	○	○	○	○	○	○
Project's needs	○	○	○	○	○	○
Abstraction level	○	○	○	○	○	○
Programming language	○	○	○	○	○	○

Table 8 continued

	Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree	I don't know
Anyone should be able to commit code without a review	◯	◯	◯	◯	◯	◯

A.15 Q15

In your opinion, what is the **maximum acceptable size** of the code review?

1. Source lines of code: _____
2. Number of files: _____

A.16 Q16

What is a **desired time range** for someone to either **accept** your code review or give you **detailed feedback**?

◯ Same day
◯ $\leq$ 24 hours
◯ 1–2 days
◯ A week
◯ Other, please specify _____

A.17 Q17

In your opinion, **how can code velocity be improved** for your projects?

For example, is the tooling (branch management, code review infrastructure) lacking, do other engineers need to be more responsive, are stricter guidelines for the code review process needed, etc.

_____
_____
_____

A.18 Q18

If you want to share more of your opinions about the the role of code velocity in software development or code review process then contact the researcher directly.

If you want to be notified about the findings from the survey then feel free to enter your email address below. We won't spam you ;-)

Email address: _____

Acknowledgements We thank all the survey participants for their insightful comments and suggestions. We are incredibly grateful to the Blender, Gerrit, Free BSD, and Net BSD 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.

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

Declarations

Funding and conflicts of interests The authors declare that they have no conflict of interest. The authors did not receive support from any organization for the submitted work.

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 EASE '20. Association for Computing Machinery, New York, pp 220–229. https://doi.org/10.1145/3383219.3383242

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

Armstrong K (2022) Category direction–code review 4. 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 ICSE '13. IEEE Press, pp 712–721. 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. 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 ESEM '16. Association for Computing Machinery, New York. 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 ICSE '15. IEEE Press, Florence, vol 1, pp 134–144. https://doi.org/10.1109/ICSE.2015.35

Barnett V, Lewis T (1984) Outliers in statistical data. Biom J 30(7):866–867. https://doi.org/10.1002/bimj.4710300725

Baum T, Schneider K, Bacchelli A (2019) Associating working memory capacity and code change ordering with code review performance. Empir Softw Eng 24(4):1762–1798. 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. Empir Softw Eng 21(3):932–959. https://doi.org/10.1007/s10664-015-9366-8

Beckman RJ, Cook RD (1983) Outlier.....s. Technometrics 25(2):119–149. 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, pp 191–201. https://doi.org/10.1109/MSR.2015.25

Blender (2022) Code review. https://wiki.blender.org/wiki/Tools/CodeReview

Boone Jr HN, Boone DA (2012) Analyzing likert data. J Ext 50. 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. 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. 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. J Soc Sci 3(3):106–116. 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 ESEC/FSE 2022. Association for Computing Machinery, New York, pp 1314–1319. https://doi.org/10.1145/3540250.3558945
Chen LT, Liu L (2020) Methods to analyze likert-type data in educational technology research. J Educ Tech Dev Exch 13(2). https://doi.org/10.18785/jetde.1302.04
Chouchen M, Ouni A, Olongo J, Mkaouer MW (2023) Learning to predicts code review completion time in modern code review. Empir Softw Eng 28(4):82. https://doi.org/10.1007/s10664-023-10300-3
Chromium (2023) Contributing to chromium. 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. J Agric Educ 35(4). 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 SBES '21. Association for Computing Machinery, New York, pp 269–274. 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 SBQS '21. Association for Computing Machinery, New York. 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. 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. J Syst Softw 204:111774. https://doi.org/10.1016/j.jss.2023.111774
Feitelson DG, Frachtenberg E, Beck KL (2013) Development and deployment at facebook. IEEE Internet Comput 17(4):8–17. 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
Fowler M (2006) Continuous integration. https://martinfowler.com/articles/continuousIntegration.html
FreeBSD Foundation (2022) Obtaining write access to the freeBSD source tree. https://wiki.freebsd.org/BecomingACommitter
Frenkel S, Kang C (2021) An ugly truth: inside Facebook's battle for domination. Harper, New York
GitHub (2021) Metrics available with GitHub insights—GitHub docs. https://docs.github.com/en/enterprise-server@2.21/insights/exploring-your-usage-of-github-enterprise/metrics-available-with-github-insights#code-review-turnaround
Gold NE, Krinke J (2021) Ethics in the mining of software repositories. Empir Softw Eng 27(1). 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 MSR '20. Association for Computing Machinery, New York, pp 606–610. https://doi.org/10.1145/3379597.3387509
Gonzalez-Barahona JM (2020) Mining software repositories while respecting privacy. https://2020.msrconf.org/details/msr-2020-Education/1/Mining-Software-Repositories-While-Respecting-Privacy
Google (2023a) Google engineering practices documentation. https://google.github.io/eng-practices/
Google (2023b) Speed of code reviews. https://google.github.io/eng-practices/review/reviewer/speed.html
Greiler M (2020) Code reviews—from bottleneck to superpower with Michaela Greiler. https://learning.acm.org/techtalks/codereviews
Groves RM (2006) Nonresponse rates and nonresponse bias in household surveys. Public Opin Q 70(5):646–675. 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. 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 EASE'17. Association for Computing Machinery, Karlskrona, pp 214–223. 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 MSR '13. IEEE Press, pp 101–110. https://doi.org/10.1109/MSR.2013.6624016
Killalea T (2019) Velocity in software engineering. Commun ACM 62(9):44–47. 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 ESEC/FSE 2022. Association for Computing Machinery, New York, pp 1398–1407. 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 ICSE '16. Association for Computing Machinery, Austin, pp 1028–1038. 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 ICSE-SEIP '18. Association for Computing Machinery, New York, pp 124–133. https://doi.org/10.1145/3183519.3183542
Kushner D (2011) Facebook philosophy: move fast and break things. 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? J Exp Soc Psychol 79:328–348. https://doi.org/10.1016/j.jesp.2018.08.009
Linux (2023) Everything you ever wanted to know about Linux -stable releases. https://www.kernel.org/doc/html/v4.15/process/stable-kernel-rules.html
Llull R (1988) Blanquerna, 2nd edn. Dedalus Hippocrene books, Sawtry, Cambs, United Kingdom, Dedalus European classics
LLVM Foundation (2023a) Contributing to LLVM—LLVM 12 documentation. https://llvm.org/docs/Contributing.html#format-patches
LLVM Foundation (2023b) LLVM code-review policy and practices. https://llvm.org/docs/CodeReview.html
MacLeod L, Greiler M, Storey MA, Bird C, Czerwonka J (2018) Code reviewing in the trenches: challenges and best practices. IEEE Softw 35(4):34–42. https://doi.org/10.1109/MS.2017.265100500
MacLeod L, Greiler M, Storey MA, Bird C, Czerwonka J (2018) Code reviewing in the trenches: challenges and best practices. IEEE Softw 35(4):34–42. https://doi.org/10.1109/MS.2017.265100500
Maddila C, Upadrasta SS, Bansal C, Nagappan N, Gousios G, Av Deursen (2022) Nudge: accelerating overdue pull requests towards completion. ACM Trans Softw Eng Methodol. 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. Ann Math Stat 18(1):50–60. https://doi.org/10.1214/aoms/1177730491
Martin RC (2002) Agile software development, principles, patterns, and practices. Alan Apt Series, Pearson
McCarthy J (1995) Dynamics of software development. Microsoft Press, Redmond
McIntosh S, Kamei Y, Adams B, Hassan AE (2015) An empirical study of the impact of modern code review practices on software quality. Empir Softw Eng 21(5):2146–2189. 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
McMartin A (2021) Introducing developer velocity lab—a research initiative to amplify developer work and well-being. 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. https://www.microsoft.com/en-us/research/event/14th-ieee-acm-international-workshop-on-automation-of-software-test/
Microsoft Research (2023) Developer velocity lab. 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. Psychol Rev 63(2):81–97. https://doi.org/10.1037/h0043158
Mozilla (2023) Code reviews—Firefox source docs documentation. 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 ICSIM '20. Association for Computing Machinery, New York, pp 210–215. https://doi.org/10.1145/3378936.3378954

Palantir (2018) Code review best practices. https://blog.palantir.com/code-review-best-practices-19e02780015f

Phabricator (2021) Writing reviewable code. https://secure.phabricator.com/book/phabflavor/article/writing_reviewable_code/#many-small-commits

PostgreSQL (2019) Submitting a patch - PostgreSQL Wiki. https://wiki.postgresql.org/wiki/Submitting_a_Patch

Raina S (2015) Establishing association. Indian J Med Res 141(1):127. https://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 ESEC/FSE 2013. Association for Computing Machinery, New York, pp 202–212. 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, 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 ICSE '08. Association for Computing Machinery, New York, pp 541–550. https://doi.org/10.1145/1368088.1368162

Riggs P (2022) Move faster, wait less: improving code review time at Meta. 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 ICSE-SEIP '18. Association for Computing Machinery, Gothenburg, pp 181–190. 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 SBES '17. Association for Computing Machinery, New York, pp 84–93. 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 ESEC/FSE 2022. Association for Computing Machinery, New York, pp 1419–1429. 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 ESEC/FSE 2022. Association for Computing Machinery, New York, pp 472–482. 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. 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 Med Res Methodol 19(1). https://doi.org/10.1186/s12874-019-0868-8

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 EASE '22. Association for Computing Machinery, New York, pp 170–179. 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 CHASE '22. Association for Computing Machinery, New York, pp 26–35. https://doi.org/10.1145/3528579.3529177

Synopsys Inc (2020) The heartbleed bug. 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). https://doi.org/10.1145/3359210

Tanna J (2021) Improving team efficiency by measuring and improving code review cycle time. https://www.jvt.me/posts/2021/10/27/measure-code-review/

The FreeBSD Documentation Project (2022) Committer's guide. https://docs.freebsd.org/en/articles/committers-guide/#pre-commit-review

The Linux Foundation (2022) A beginner's guide to Linux Kernel development. 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. 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 Eng 22(2):768–817. https://doi.org/10.1007/s10664-016-9452-6

Tsay JT (2017) Software developers using signals in transparent environments. PhD Thesis, Carnegie Mellon University. https://doi.org/10.1184/R1/6723026.v1

Vanian J (2022) Internal Facebook memo warns company must be disciplined, prioritize ruthlessly. 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 MSR '08. Association for Computing Machinery, Leipzig, pp 67–76. 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. 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 Trans Softw Eng 42(6):530–543. 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 FSE 2016. Association for Computing Machinery, New York, pp 871–882. https://doi.org/10.1145/2950290.2950364