Debunking the Myths of Code Velocity: The Elusive Search for Speed

Gunnar Kudrjavets*
Amazon Web Services
410 Terry Ave N
Seattle, WA 98109, USA
gunnarku@amazon.com

ABSTRACT

We demonstrate that no empirical evidence supports long-held beliefs related to code velocity. The best-known pragmatic way to increase code velocity is to periodically nudge engineers to unblock the code review process. The industry needs to revisit the Modern Code Review process in light of the availability of various tools and techniques that analyze and prevalidate the proposed code changes. The code review bots are coming.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*code inspection, code review*; D.2.8 [**Software Engineering**]: Metrics—*time-to-first-response, time-to-accept, time-to-merge.*

Keywords

Developer productivity, code velocity, code review.

1. CONTEXT

The first part of this short column summarizes the author's doctoral dissertation, published in October 2023 [12]. The second part of this opinion piece intends to trigger a discussion about the future of code reviews in an era when code velocity matters more than ever, and various bots have become active participants in the software development process.

2. INTRODUCTION

One possible definition of *code velocity* is "the time between making a code change and shipping the change to customers" [15]. The cadence and process of "shipping" (releasing software to customers) are highly variable and depend on different factors. Some of these factors are the type of industry, the project, and the software's closed-source versus open-source nature.

This paper focuses on a common activity during the development cycle regardless of how the software is shipped—the code review process and its duration. We use the following three terms to describe the various stages of a code review: time-to-first-response, time-to-accept, and time-to-merge.

- *Time-to-first-response* [4, 13] is the period between when the engineer publishes the code changes for review until the first code review-related activity (e.g., acceptance, comment, rejection) from another human who is not the author of the changes.

- *Time-to-accept* [4] is the period between when the engineer publishes the code changes for review until the necessary quorum of human reviewers agrees that the changes are ready to be merged into their destination branch.

- *Time-to-merge* [10]: "... the time since the proposal of a change (...) to the merging in the code base ..." [9].

Our goal is to investigate what factors influence those periods, how we can shorten them, and what types of beliefs about code velocity are prevalent in the larger developer community.

3. BACKGROUND AND MOTIVATION

Empirical software engineering researchers have extensively studied the Modern Code Review process [1, 16, 3, 2]. Academic researchers are free to investigate different aspects of the code review process. However, for a practitioner, only a few aspects practically matter.

Based on the author's empirical experiences and observations, a practitioner in the software industry is primarily interested in the following data points:

1. "Will someone react to my code review, and will there be a meaningful engagement?" That is the case mainly for open-source software developers. Nobody is obligated to pay attention to the proposed code changes in the open-source software ecosystem. In the industry, someone is responsible for reviewing proposed code changes, so eventual engagement is assumed.

2. "When will a first response to my code review manifest?" A delayed response is rarely beneficial to the code author, reviewers, or the project. Earlier feedback enables engineers to iterate faster and work on code changes until they meet a particular project's requirements.

3. "When will my code review be accepted?" Acceptance is a significant step towards propagating the code changes to their destination. Once the changes are accepted, then in most cases, the next steps include resolving the merge conflicts and working through whatever issues the Continuous Integration and Continuous Delivery/Deployment pipelines will surface.

4. "When will my code review be merged into a destination branch?" The moment the code changes are committed to a destination branch is when they finally "become real" for other engineers and the project.

*The dissertation's contents, which serve as the basis of this article, and the article itself were written before Gunnar Kudrjavets became an employee of Amazon.com, Inc. All opinions and statements communicated in this paper are the author's own.

In the industry, code velocity is critical because the success of a company and individual engineers depends on their ability to deliver results within a given timeframe. If engineers do not complete the code reviews promptly, they cannot deploy the code changes on time, and therefore, the fixes for defects or features will not reach customers on time.

The primary motivation behind current research was the author's daily feeling of cognitive dissonance between the informal guidance to engineers, the existing beliefs related to code velocity, industry practices, and the observed reality. The following widely held beliefs motivated different research papers that served as the basis of the author's dissertation:

1. "As the size of the projects increases, the code reviews become slower." We show in the answer to RQ_2 that the opposite is true.
2. "If an engineer publishes small code changes for review then they will be accepted and merged faster." We show in the answer to RQ_3 that there is no evidence to support this claim.
3. "The code reviews for kernel code take less time because most authors and contributors are very senior, so there are less iterations during code reviews." We show in the answer to RQ_5 that at least based on publicly accessible data from FreeBSD code reviews the opposite it true.
4. "It is worth fixing all the compiler warnings even if it reduces code velocity because that increase software quality?" We show in the answer to RQ_6 that somewhat surprisingly there is no scientific evidence showing the meaningful relationship between post-release defects and the compiler warnings.

4. RESEARCH QUESTIONS

Author's dissertation poses the following research questions:

1. RQ_1: "Is there publicly accessible high-granularity code review data?"
2. RQ_2: "What is the trend of code velocity across various software projects?"
3. RQ_3: "Does the size of code changes correlate to the duration of various code review phases?"
4. RQ_4: "What phases of the code review process are inefficient, and what can we improve?"
5. RQ_5: "Does the code velocity differ between kernel and non-kernel code?"
6. RQ_6: "Should we delete dead code and stop fixing compiler warnings?"
7. RQ_7: "What are the beliefs, practices, and convictions related to code velocity?"

5. METHOD

To answer these questions we use mainly quantitative methods. The empirical method for RQ_1–RQ_5 is exploratory case study. For RQ_6 we use a combination of ethnography and literature review. For RQ_7 we employ survey research and content analysis as a main vehicle behind the qualitative research.

6. FINDINGS AND RESULTS

RQ_1. A variety of open-source code collaboration tools exist. The most popular ones are Gerrit, GitHub, and Phabricator. We find that GitHub uses the formal code review process for only a subset of projects and the usage is inconsistent. One of the drawbacks of Gerrit is that Gerrit does not distinguish between time-to-accept and time-to-merge. Phabricator is the only tool that exposes high-granularity data about various events that occur during the code review. We use the existing data mining tools to publish a set of Phabricator code reviews as a queryable relational database.

RQ_2. Based on our dataset of 283,235 code reviews we investigate the trend for code velocity in Blender, FreeBSD, LLVM, and Mozilla. On average, our dataset covers seven years of development history. Our critical finding is that *code velocity does not decrease over time*. The code velocity either *stays the same* or *slightly increases*. This fact is both surprising and positive because the size of the code bases for these projects increases on a median between 3–17% annually.

RQ_3. We investigate if there is a correlation between the pull request size and time-to-merge. Our data source is 100 GitHub projects that are under active development. We find no relationship between the pull request size and time-to-merge. The results are the same for the code reviews conducted using Gerrit and Phabricator. For Gerrit and Phabricator we show that there is no correlation between code review size and time-to-accept.

RQ_4. We investigate the presence of non-productive time in Gerrit and Phabricator code reviews. We quantify what happens with the code reviews when they are accepted and ready to be merged. Our findings show that in more than half of the cases there is no activity between time-to-accept and time-to-merge. Enabling automatic merging of changes for Phabricator projects has a potential to increase code velocity by 29–63%.

RQ_5. We study four BSD family operating systems: DragonFlyBSD, FreeBSD, NetBSD, and OpenBSD. Our intent is to research the differences in the kernel and non-kernel code for commit sizes, commit taxonomy, and code velocity. We find that engineers change either kernel or non-kernel code, but rarely both categories at the same time. We discover that in FreeBSD the code velocity for kernel code is slower than for non-kernel code.

RQ_6. We find surprisingly that there is no existing evidence that shows either the correlation or causal relationship between fixing compiler warnings and reduction in post-release defect density. In addition, we discover that industry lacks metrics to quantify the benefits of deleting dead code.

RQ_7. We surveyed the wider developer community about the beliefs and practices related to code velocity. Our analysis is based on 75 completed surveys with 39 responses from the industry and 36 responses from the open-source software community. We find that the majority of beliefs and trade-offs in both of these ecosystems are similar. For the engineers in the industry, it is obvious that decreasing code velocity is not good for anyone's career progression. However, it is not clear that investment into increasing code velocity results in career growth. Out of potential solutions to increase code velocity, a selective application of the commit-then-review model scored the highest. On a positive note we find that 100% of open-source and 82% of industry developers are unwilling to compromise on software security to increase code velocity.

7. DISCUSSION

7.1 A limited number of contrarian views

The popularization of code reviews in the industry started with Fagan's seminal paper on code inspections [8]. Even more than half a century later, very few researchers have questioned the effectiveness of the code review process. While the Modern Code Review process does not require engineers to conduct formal review meetings and assign official roles to participants in the review process, little else has changed.

One of the rare papers that questions the widely accepted beliefs about the benefits and established code review process comes from Microsoft [7]. The paper states that "code reviews often do not find functionality issues that should block a code submission" and "that effective code reviews should be performed by people with specific set of skills." These claims match the developer folklore and author's experience with industry and open-source projects.

This critique from industry illustrates another problem with empirical software engineering research—the different values and viewpoints of academics and practitioners. The view from the trenches (of reviewing code) of a practicing software engineer is often very different than that of a researcher [13]. The map is not the territory [11]. Fred Brooks's quote feels appropriate here: "Thinkers are rare; doers are rarer; and thinker-doers are the rarest" [5]. We need more of the *doers* writing papers than just writing code to help advance the field of developer productivity.

7.2 Advancements in pre-validation toolchains

The software industry has made massive leaps in pre-release defect detection. Compilers with improved error detection (e.g., the `-fanalyzer` switch in GCC), linters, static analyzers, style validators, and various sanitizers such as ASan, TSan, and UBSan are all available to engineers to validate their changes *before* even submitting any lines of code for a review. In the industry, engineers can typically run numerous pre-validation test cases before the code reviews are published. As a result, automation now discovers many defects that previously required a human to detect. For example, various tools can effectively validate that a function releases memory allocated when it exits because of an obscure error condition and does not leak memory. The value proposition that a human code reviewer can provide in 2024 is very different from what it was even a decade ago.

7.3 Trade-offs between speed and quality

Another issue the industry should revisit is the dictum that someone must review all the code changes. Most software engineers do not write code that supports interplanetary travel or manages the daily operations of a nuclear reactor. Even in non-critical software, there are different layers. The requirement to have mandatory code reviews sounds reasonable in some areas where the consequences of defects are severe—such as cryptography libraries, file systems, or operating system kernels. Everything else? It depends.

For example, most open-source projects in the operating systems space have separate *committer* and *contributor* roles. A committer has historically proven to have good judgment and thus can be trusted to make code changes with less oversight. The FreeBSD project requires that "[a]ll non-trivial changes should be reviewed before they are committed to the repository" [19]. A critic can immediately reason that one engineer's trivial change is another engineer's feature. A counter-argument is that if an engineer is qualified to implement a high-performance thread pool that runs in kernel mode, they should be trusted to make judgment calls on who will review their code and when.

8. THREATS TO VALIDITY

The main threat to the validity of the author's research is that all the findings base themselves on open-source software. While open-source software powers everything from cell phones to clusters of supercomputers, a large amount of code produced worldwide is closed-source. Research in empirical software engineering utilizes code review activity of operating systems such as FreeBSD and various Linux distributions because the relevant data is freely available. However, we rarely see papers about the code review process for Apple's macOS or Microsoft's Windows or the evolution of code velocity in other commercial products.

The industry limits access to code bases for obvious reasons (competitive advantage, patents, potential reputational damage, trade secrets). Even if the researchers who work for a particular organization discover results useful for the wider research and software engineering community, it is uncertain if those results can ever be published.

Another obvious threat to validity is the author's industrial background and the inherent biases associated with it. A "proper academic" may look at the challenges related to code velocity from a very different angle than a practicing software engineer whose professional career depends on it.

9. WHAT DOES THE FUTURE LOOK LIKE?

The Roman Stoic philosopher Lucius Annaeus Seneca has highly influenced the author's take on the future—the future is *uncertain* [17]. However, to spark some discussion, the author will entertain some thoughts on how the code review process can change in years to come.

In 2021, when the author started working on his dissertation, ChatGPT was was yet to be launched. In 2024, in addition to ChatGPT, a plethora of similar chatbots such as Copilot, Gemini, and Grok have become a part of a toolset to enhance everyone's productivity. The term "Generative AI" has become a part of everyday vocabulary. We even have "Devin, the first AI software engineer" [21]. Surely, Devin will not want to wait for days for someone (a human or a bot) to review their one character change to convert a function call from `strncpy()` to `strlcpy()`? Devin would like their colleague *Devin′* (or an instance of it) to review and either approve or reject the changes in seconds.

While seasoned AI researchers have differing views on whether chatbots are just a reincarnated version of ELIZA that runs on a cluster of GPUs or advancements in AGI, it is hard to argue that their presence has not impacted the fabric of our everyday lives.

The author hopes that shortly (in a year or two), we see the application of chatbots and LLMs in a way that significantly reduces the cognitive load on software engineers. Most of the decisions in software engineering will always be trade-offs between the risk and various characteristics of a particular project. However, in most software projects, there is no reason that trivial code changes such as fixing compilation errors or updating comments will have to wait for days or weeks for another human to approve.

An ability to have highly customizable AI-powered agents or bots that roam in the code base and either suggest or review the code changes will become a matter of survival in the industry. The "found by a bot, fixed by a bot, reviewed by a bot, committed by a bot, and deployed by a bot" sequences will become a new normal. Likely, the software industry will go through the phase of an equivalent of "AI doomers" predicting the imminent appearance of

Skynet if we let bots attempt to fix the build errors while engineers are soundly asleep.

Some of these concerns are definitely valid. We do not want an instance of a rogue CodeReviewBot that overrides its governor module and spends most of its time watching soap operas such as "The Rise and Fall of Sanctuary Moon" [20]. On the other hand, if this results in faster code reviews, the trade-off may be acceptable.

10. CALL TO ACTION

During the last 10–15 years, the majority of commercial software companies have switched from delivering shrink-wrapped software every N years to Continuous Integration and Continuous Delivery/Deployment. The advances in hardware have given us faster build times, parallelized test execution, and the ability to integrate several pre-validation toolchains into a code review process.

However, *the human element is still a bottleneck in the code review process*. While the other parts of the software development lifecycle have improved from taking years to days, there have been few improvements to the code review process. The state-of-the-art method to increase code velocity is nudging, i.e., annoying bots reminding engineers to fulfill their reviewer duties on time [14, 18]. The predictive modeling in the industry is ineffective in identifying what factors contribute to the duration of code reviews [6].

Though the peer review process in software engineering has been in practice for half a century, it remains one of the few areas where the software industry remains "stuck in the 70s." The Modern Code Review process needs more significant changes to justify being called *modern*.

11. ACKNOWLEDGMENTS

The author is grateful to Dr. Ayushi Rastogi from the University of Groningen for supervising a nontraditional Ph.D. student and supporting the author's often contrarian opinions on research. The author's former colleagues from Microsoft and Meta (Aditya Kumar, Jeff Thomas, and Dr. Nachiappan Nagappan) were instrumental in providing criticism related to the author's research.

12. REFERENCES

[1] A. Bacchelli and C. Bird. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 712–721. IEEE Press, May 2013.

[2] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey. Investigating Technical and Non-Technical Factors Influencing Modern Code Review. *Empirical Software Engineering*, 21(3):932–959, Mar. 2015.

[3] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens. Modern Code Reviews in Open-Source Projects: Which Problems Do They Fix? In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 202–211, New York, NY, USA, May 2014. Association for Computing Machinery.

[4] C. Bird, T. Carnahan, and M. Greiler. Lessons Learned from Building and Deploying a Code Review Analytics Platform. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR)*, pages 191–201, Los Alamitos, CA, USA, May 2015. IEEE Computer Society.

[5] F. P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison Wesley, Boston, MA, USA, 2nd edition, Aug. 1995.

[6] L. Chen, P. C. Rigby, and N. Nagappan. 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, pages 1314–1319, New York, NY, USA, Nov. 2022. Association for Computing Machinery.

[7] J. Czerwonka, M. Greiler, and J. Tilford. 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*, volume 2, pages 27–28, May 2015.

[8] M. E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15(3):182–211, 1976.

[9] D. Izquierdo-Cortazar, N. Sekitoleko, J. M. Gonzalez-Barahona, and L. Kurth. Using Metrics to Track Code Review Performance. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, EASE '17, pages 214–223, Karlskrona, Sweden, June 2017. Association for Computing Machinery.

[10] O. Kononenko, O. Baysal, and M. W. Godfrey. Code Review Quality: How Developers See It. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 1028–1038, Austin, TX, USA, May 2016. Association for Computing Machinery.

[11] A. Korzybski. *Science and Sanity: An Introduction to Non-Aristotelian Systems and General Semantics*. Institute of General Semantics, 4th edition, Jan. 1995.

[12] G. Kudrjavets. *The Need for Speed: Increasing the Code Review Velocity*. PhD thesis, University of Groningen Press, Groningen, Netherlands, 2023.

[13] L. MacLeod, M. Greiler, M.-A. Storey, C. Bird, and J. Czerwonka. Code Reviewing in the Trenches: Challenges and Best Practices. *IEEE Software*, 35(4):34–42, June 2017.

[14] C. Maddila, S. S. Upadrasta, C. Bansal, N. Nagappan, G. Gousios, and A. v. Deursen. Nudge: Accelerating Overdue Pull Requests Towards Completion. *ACM Trans. Softw. Eng. Methodol*, June 2022.

[15] Microsoft Research. 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/, May 2019.

[16] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli. 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, pages 181–190, Gothenburg, Sweden, May 2018. Association for Computing Machinery.

[17] L. A. Seneca. *On the Shortness of Life*. Great Ideas. Penguin Books, London, United Kingdom, Sept. 2005.

[18] Q. Shan, D. Sukhdeo, Q. Huang, S. Rogers, L. Chen, E. Paradis, P. C. Rigby, and N. Nagappan. 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, pages 472–482, New York, NY, USA, 2022. Association for Computing Machinery.

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

[20] M. Wells. *All Systems Red*. Tor Books, New York, NY, USA, May 2017.

[21] S. Wu. Introducing Devin, the first AI software engineer. https://www.cognition-labs.com/introducing-devin.