Chapter 8
Conclusions and Future Work

What do you accomplish in life from the sidelines?
Absolutely nothing.
— JP Dinnell, Former U.S. Navy SEAL

8.1 Answers to research questions

This section summarizes the answers to the research questions formulated in Chapter 1.

RQ1: Is there publicly accessible high-granularity code review data? (covered in Chapter 2). We find that out of the popular open-source code collaboration tools, Phabricator is the only one that exposes the data we require for our research. Gerrit does not distinguish between time-to-accept and time-to-merge. GitHub uses the formal code review process in a limited capacity. So far, we have used that Phabricator dataset to investigate (a) the trend of code velocity for major open-source software projects (Chapter 2), (b) the relationship of code changes to acceptance time (Chapter 3), (c) the presence of non-productive time during the code review process (Chapter 4), and (d) the differences in code velocity between kernel and non-kernel code (Chapter 5).

RQ2: What is the trend of code velocity across various software projects? (covered in Chapter 2). We investigate the trend for code velocity in four major open-source software projects: Blender, FreeBSD, LLVM, and Mozilla. Our analysis is based on 283,235 code reviews that span, on average, seven years of development activity per project. We find that code velocity does not decrease over time. While the size of the code base in these projects increases on a median between 3–17% annually, the code velocity either stays the same or slightly improves.
This fact is both a surprising and a positive finding. We observe that most data points for larger projects (FreeBSD, LLVM, and Mozilla) for the 30-day rolling median code velocity stay in a fixed range of hours. We note this phenomenon for future research in Section 8.3.2.

**RQ3: Does the size of code changes correlate to the duration of various code review phases? (covered in Chapter 3).** This study presents whether the pull request size and composition can be meaningfully changed to increase code velocity. We selected 100 popular, actively developed projects on GitHub to study the relations of pull request size and composition to code velocity measured as time-to-merge. Our analysis shows no relationship between pull request size and composition to the time-to-merge, regardless of how we partition the data: day of the week the pull request was created, affiliation to industry, and programming language. We found no patterns even though pull requests affiliated with the industry are larger and take less time than non-industry equivalents. Our results remain the same on two other platforms: Gerrit and Phabricator, which also offered another indicator of code velocity—time-to-accept.

**RQ4: What phases of the code review process are inefficient, and what can we improve? (covered in Chapter 4).** This study quantifies non-productive time in Gerrit and Phabricator code reviews and investigates what happens to the code reviews after they are accepted. Our study shows no activity between acceptance and merging in more than half of the cases. Our exploration into the cause of delay offers actionable insights into potentially increasing code velocity. We estimate that in the case of Phabricator projects, the code velocity can increase by 29–63%.

**RQ5: Does the code velocity differ between kernel and non-kernel code? (covered in Chapter 5).** We conduct a large-scale study on four BSD family operating systems: DragonFlyBSD, FreeBSD, NetBSD, and OpenBSD. Based on the literature review, we are the first to explore the differences between commit sizes, commit taxonomy (neutral, additive, subtractive), and code velocity in the kernel and non-kernel code in the context of operating systems development.

Our key finding is that researchers and practitioners should view a larger software system as a collection of subsystems and sub-components, not just one entity. Our analysis shows that when making code changes, (a) developers modify either kernel or non-kernel code, but rarely code belonging to both categories, (b) the
median size of commits to kernel code is larger than non-kernel commits, (c) both kernel and the non-kernel code bases have a similar annual growth rate, and (d) in FreeBSD, the code reviews for kernel code take longer than code reviews for non-kernel code.

RQ6: Should we delete dead code and stop fixing compiler warnings? (covered in Chapter 7). This chapter 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. Demographically, 39 participants were from the industry, and 36 respondents were from the open-source software community. Based on what we know, this is the first paper that studies the trade-offs engineers make to increase the code velocity and critical impediments that block engineers from increasing code velocity even more.

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

8.2 Contributions

This section succinctly summarizes what this dissertation adds beyond state-of-the-art.

- In Chapter 2, we introduce a dataset of Phabricator code reviews for the researchers to use. The toolset for mining the Phabricator data has been available [Cotet, 2019, 2021]. To the best of our knowledge, this is the first time a significant number of code reviews have been exposed in an easily accessible format as a MySQL database. In addition, Chapter 2 introduces a taxonomy of code review mining smells that is a novel contribution to mining software repositories. In the same chapter, we also establish that based on the available data, there are no noticeable improvements in the overall code velocity for Blender, FreeBSD, LLVM, and Mozilla.
• In Chapter 3, we dispel a longstanding misconception that smaller code changes merge faster. The traditional application of this myth has manifested itself as guidance given to software developers to produce smaller changes because they will be reviewed and subsequently committed faster. While the smaller code changes may be beneficial for other reasons, we show no relationship between the size of changes and time-to-merge. The researchers from Meta [L. Chen et al., 2022] confirm our findings in the industrial context. Meta’s “model shows that review times are consistent regardless of factors such as diff size” and factors such as diff size “are not strong enough indicators of review time.”

• In Chapter 4, we study the temporal characteristics of half a million code reviews. Reducing the time between acceptance and merging can speed up Phabricator code reviews by 29–63%. A simple change in process to switch from manual merge to automatically merging accepted code changes can increase code velocity. We are the first researchers to quantify the time wasted in code reviews at this scale. We also find that the small code changes and changes made by authors with a large number of previously accepted code reviews have a higher chance of being immediately accepted. While there is no direct relationship to increased code velocity here, the reduced number of code review iterations anecdotally implies less time wasted on the same code review and potentially higher job satisfaction for the engineers involved in the process.

• In Chapter 5, we compare the code churn and code velocity between kernel and non-kernel code. We are unaware of any other research in developer productivity and software evolution that investigates these issues in the context of an operating system. While the critical finding (code velocity for kernel code is slower than code velocity in the non-kernel code) is evident to an operating systems engineer, it is the first attempt to quantify the code velocity between the different abstraction layers of a larger software system.

• In Chapter 6, we point out that engineers have spent energy and time fixing compiler warnings for decades, but the return on investment for that work is unknown. Compiler research mainly focuses on technical aspects such as correctness, code generation optimization, or fault detection. We are the first public researchers to question the idea behind fixing the compiler warnings. While corporate research at Google [Winters et al., 2020] may have determined that fixing the compiler warnings has no predictive value
for maintenance or quality, these findings have not been made public. Similarly, we point out that the inability to quantify the lack of benefits from removing dead code is a topic that is worth more research.

- In Chapter 7, we studied code velocity involving industry and open-source software developers. Some of the findings about the time-to-merge being an essential code review metric to improve and quick reaction to code reviews being of utmost importance to engineers are confirmatory. We discovered that engineers have concerns about the benefits of increased code velocity for their career growth, and (somewhat surprisingly for the author) the industry and open-source community hold similar beliefs related to code velocity.

## 8.3 Future work

This section categorizes the future work from the papers related to code velocity that form this dissertation.

### 8.3.1 Dataset and data mining

The current Phabricator dataset opens avenues for new research opportunities. For example, (a) utilizing the formal association between code reviews and bugs (tracked by Mozilla project [Mozilla Foundation, 2021]), (b) evolution of differential revisions by analyzing their subsequent versions, and (c) investigating events that occur during the code review life cycle that other code collaboration tools do not track.

In this version of the dataset, we did not include all the details about each version of the differential revision, such as statistics about code changes per file and file names. Our intention here is to avoid duplication of data stored in a source control system. The current dataset can be augmented for deeper insights if such fine-grained data appears relevant.

We plan to publish a checklist of potential issues that need mitigation when mining code reviews. Researchers can use that list to determine if these smells threaten the validity of their research project.
8.3.2 Code velocity trends

- **Confirmatory replication.** We intend to replicate our findings for other code collaboration tools like Gerrit and GitHub. While they do not provide the same granularity level as Phabricator, the findings will help invalidate or strengthen our claims.

- **Fixed range.** Scatterplots in Figure 2.1 indicate that time-to-merge for most code reviews is between $10^{1.25}$ and $10^{2.5}$ hours, regardless of the project and stage. This observation about the code review lifetime falling into a specific range is worth additional research. Are there independent quantifiable variables that can influence that range? Are enough samples falling outside that range, such as too fast or slow code reviews, an early indication of problems associated with a project?

- **Differences in code churn and velocity differ at a more granular level.** The kernel code can be subdivided even into more specific areas. For example, file systems, memory management, or networking. Though the level of technical complexity is similar (albeit in a different domain), do we see noticeable differences there as well?

- **The differences in code churn between kernel and non-kernel code over a certain period?** One of the beliefs we have encountered is that kernel code changes at a slower rate than non-kernel code. We can quantify the change as code churn during a period, e.g., how does the total amount of code churn per month relate to the size of the overall code base?

8.3.3 Optimizing code reviews

1. **Reasons why additional code changes are requested after the initial acceptance.** Are the instances when the first acceptance is considered incorrect caused by the initial reviewer missing defects they noticed later or by another reviewer discovering additional set of problems with the code? Is there a relationship between factors such as the size of changes, previous contribution history of the author, and the experience of the reviewer? Given that invalidating the initial acceptance is a failure in the code review process it is essential to understand the causes of these cases and explore the potential for process improvements.

2. **Factors influencing the time-to-first-response.** Because time-to-first-response is responsible for a significant portion of overall time-to-merge, shortening
this time is the next obvious step to increase the code velocity. Several variables can impact how fast a qualified reviewer reacts to a code review. These variables can include everything from the author’s identity, availability of reviewers, complexity, size of changes, and interpersonal relationship between the reviewer and author. The goal is to understand how a code change author can solicit a faster response.

3. *Implications of code review policy on defect density.* We do not have data points on different code review policies (acceptance being gated by CI validation results, number of required reviewers per each change, committing changes automatically on accept versus manually by the author). One future research direction can be to investigate if policy differences have a measurable impact on code quality or defect density. For example, do projects requiring two reviewers (instead of one) or mandating that a member of the core committers group must review all code changes have a lower defect density? Is Linus’s law stating that “given enough eyeballs, all bugs are shallow” supported by factual evidence [Raymond, 1999]? 

4. Our finding that changing size or composition does not influence code velocity prompts new questions, such as the impact of personality traits and interpersonal relationships on time-to-merge. Besides pull request size and composition, engineers have limited control over their behavior, communication, and mannerisms while interacting with other team members. A study has shown that an engineer’s reputation in the general community and the project is a good predictor for pull request acceptance [Baysal et al., 2013]. Intuitively, it makes sense that engineers with good communication skills, well-versed in conflict resolution, and empathy towards other participants will receive more cooperation.

8.3.4 Code churn and code velocity between different subsystems

As part of our future research, we intend to focus on the following topics:

- Do developers gravitate towards kernel as they gain more experience with operating systems development? New contributors joining an open-source software project typically start contributing by making more straightforward changes. Generally, the opportunities associated with the least risk are in user mode, e.g., making changes in a command-line tool. As engineers gain
more confidence and experience, do they change their focus to areas where the stakes are higher than in user mode, e.g., device drivers?

- **Do developers mainly contribute to their abstraction layer of choice?** It is common in BSD and Linux development to have maintainers for each area. One of the interesting questions is related to the distribution between “specialists” (engineers who contribute only to a few narrow areas) and “generalists” (engineers who make changes in various components). Based on the Linux kernel research, most engineers (62%) who contribute to the kernel have a narrow specialist profile [Avelino, Passos, Hora, & Valente, 2017]. We do not know if this holds in the context of an entire operating system.

### 8.3.5 Organizational and interpersonal stimuli

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.

### 8.4 Discussion

#### 8.4.1 Reflection on the contributions

One of the main realizations that the author had is that it is significantly easier to conduct meaningful research in software engineering with access to industrial data. Conducting proper research that the author can stand by was a more involved challenge than the author predicted at the beginning of this journey. Commercial organizations can spend millions of dollars and the efforts of hundreds of engineers to improve daily engineering workflows. Most open-source software projects operate on either selected donations or volunteer efforts. As a result, the differences between open-source and state-of-the-art industry code collaboration tools, CI systems, and other parts of the infrastructure used for daily development work are vast.

One of the research avenues that the author abandoned early on was to use experimental research methods such as basic A/B testing to investigate potential ways to increase code velocity. In our subjective estimates, an experiment that would have taken less than a day to set up in the industry would have taken
months or even years of embedding into a specific community, literal lobbying, and convincing to apply to larger open-source software projects. While all the current findings are publicly available, the author still wonders how much difference they would make in an environment such as Microsoft Windows, where thousands of engineers [Lucovsky, 2000; Maraia, 2005] work daily to ensure that their code changes reach from one branch to another promptly.

On a positive note, the next time someone asks the author to split his patch of 38 lines of pure C code into three separate patches because they will reach the target branch faster, there is enough evidence to question that request purely on the merit of increasing the code velocity.

While the clinical evidence about the benefit of fixing compiler warnings is missing, the author will continue his quest toward warnings-free code bases. If anything, the past countless sleepless nights and weekends consumed by chasing elusive integer conversion issues, memory corruptions, and mismatches in calling conventions have installed a strong (and possibly unjustified) belief in the author’s mind that “it pays to be warnings-free.”

### 8.4.2 Limitations of the contributions

A critical limitation of this work is that we draw our conclusions from the available data versus the desired data. One of the most popular code collaboration tools, GitHub, introduced the formal code review process relatively recently in 2016 [GitHub, 2016]. Based on the sampling we conducted in Chapter 2, Chapter 3, and Chapter 4, there need to be more projects that use the formal code review process consistently. As a result, our ability to draw conclusions based on GitHub data is limited. A significant limitation related to Gerrit data is that in the context of Gerrit, there is no conceptual difference between time-to-accept and time-to-merge. That constraint prevents us from making certain inferences for Gerrit.

In Chapter 5, where we discuss the differences in code churn and code velocity in the operating system context, our ideal research target would have been different. We would have investigated various versions of Apple macOS and its derivatives, such as iOS and iPadOS, or different releases of Microsoft Windows. These commercial operating systems have 2–3 decades worth of active development history involving thousands of engineers’ contributions. However, internal data from these companies is not accessible to outside researchers, and there are no guarantees that even the internal findings can be made available to the research community.
Various Linux distributions or kernels in isolation have been a popular research target [Erdamar, 2021; Israeli & Feitelson, 2010; Jiang et al., 2013; Tan & Zhou, 2019]. Determining various formal state changes in the Linux patch review process and even “retrieving and cleaning the data is a non-trivial job” [Xu & Zhou, 2018]. Similarly, in Chapter 5, we mention that various distributions of the BSD family of operating systems do not follow the consistent code review process or even store the data about patch reviews. That limits our ability to make claims based on the data we can mine from the available sources.

Our work on compiler warnings discovers that we need empirical evidence about how fixing the compiler warnings can benefit software developers [Kudrjavets, Kumar, et al., 2022b]. For our research, this finding prunes one of the possible branches in the space of the possible solutions. However, we have yet to propose any practical solutions to this problem. Likewise, we need a usable framework to quantify the benefits of deleting dead code [Kudrjavets, Rastogi, et al., 2022]. We leave this problem unsolved for either future papers or researchers to tackle.

### 8.4.3 Forecast of the future

We predict that the issues related to code velocity will be a challenge in the foreseeable future for all organizations and projects that operate on any deadlines. The advances in the hardware, such as faster CPUs and SDDs, increased opportunities for parallelization of validation steps in CI systems, and caching improvements in modern build systems will provide temporary incremental improvements. However, the fundamental requirements of timely human involvement and engagement in the code review process will persist.

Technologies such as ChatGPT can potentially disrupt the current code review process if researchers apply large language models to provide feedback to authors of code changes. The recent advances in automated code reviews [Chatley & Jones, 2018; H. Kim, Kwon, Kwon, et al., 2022; Zhou et al., 2023] and using bots [Wessel et al., 2020] are encouraging. Based on the author’s experience and anecdotal evidence from the industry, the current state of bot application still needs noticeable improvements to be helpful for practitioners.

Like any other metric, we do not expect that the desire to increase code velocity will stop even after the potential noticeable improvements. In 1954 Roger Bannister officially became the first human being to run a sub-four-minute mile in 3:59.4. The current world record by Hicham El Guerrouj is 3:43.13 as of 1999. Nevertheless, world-class athletes are pushing the boundaries of human
endurance and speed each day to reduce that time by a second or two. In the author’s experience, the traditional Olympic motto of “Citius, Altius, Fortius” is highly applicable to various fields of software engineering.

The author anecdotally observes that even in 2023 his former colleagues are still battling the ageless issues of software performance engineering, such as reducing network or storage I/O, increasing the speed of inter-process communication, improving various synchronization primitives, and reducing memory usage. All of it to be able to either deploy, validate or consume code changes faster.

The safety-critical software category is one niche category that the author hopes will stay untouched by the drive to increase code velocity. The malfunction in safety-critical systems can potentially cause death to people and cause severe damage to equipment and the environment. Typical examples include systems responsible for aviation, medical equipment, and nuclear engineering. We can argue that engineers and scientists should work towards decreasing the code velocity and take all the time necessary to ensure the correctness of the software. Fortunately, various industry standards exist in those fields to avoid life-threatening defects.

Given the nature of the Sisyphean task to increase the code velocity, one wonders if adding a mandatory course of stoicism to the modern computer science curriculum will be beneficial to prepare future generations of engineers and scientists for the unpleasantness of reality.