Chapter 1

Introduction

There’s daylight again outside the window,
The day challenges me to a battle.
I feel, closing my eyes, —
The whole world declares war on me.

— Viktor Tsoi & Kino, “Song without words”

1.1 Foundations

1.1.1 Motivation

The image on the front page of this dissertation is a stopwatch that counts down the time until the deadline. A professional software engineer lives and breathes deadlines. Meeting or missing deadlines for critical projects is the difference between a software company continuing to exist or going bankrupt. A project that is repeatedly late is likely to get canceled. Engineers who do not meet their deadlines have a difficult time progressing in their careers. The United States Navy SEALs have a saying that “it pays to be a winner” [Couch, 2005]. In the software industry, it pays to be fast and on time.

One critical metric determining if the engineers have met the deadline is that the desired code changes have reached their destination. That destination can be a final version of a software product, a deployed mobile application, the main branch on GitHub, or a production environment in the corporate data center. After the code changes are merged with the existing code base, they are ready for deployment, performance measurements, and testing. Being late early in the development cycle causes a chain reaction that impacts each consecutive phase of product development. When a project is constantly behind schedule, the team morale decreases, stress increases, and the feeling of inevitable doom appears [Weinberg, 1998]. As a result, each software development team member
who values their career or has a certain amount of professional pride cares about meeting deadlines and completing tasks as fast as possible.

Several descriptions of code velocity exist in the industry. One widely used definition of code velocity is “the time between making a code change and shipping the change to customers” [Microsoft Research, 2019]. Making a code change can mean submitting the changes for a code review or committing them to a main development branch. The code changes being accessible to customers (i.e., having been “shipped”) is a moment when a consumer can purchase the software or use the version of the software that contains the deployed changes (e.g., a mobile application that is available in the Apple Store).

The software industry wants to increase the code velocity, i.e., enable users to consume the new or updated product faster. In the early 2000s, agile software development became mainstream and dominant [K. Beck et al., 2001]. Agile methodologies include Crystal, Extreme Programming (XP), Kanban, and Scrum. The Merriam-Webster dictionary defines agile as “able to move quickly and easily” [Merriam-Webster, Inc., 2023]. A critical goal in agile software development is delivering software changes to users as quickly as possible. Though various methodologies have different nuances, the end goal is the same—faster software development.

The speed at which code changes are merged is essential for several reasons. In the industry, the main incentives are monetary. It pays to be the first on the market and beat the competitors. The first-mover advantage is a powerful incentive [Lieberman & Montgomery, 1988]. In the software industry, the first-mover advantage means that a company that releases a specific category of a product first can occupy and maintain a significant portion of the market share. The caveat is that the product must meet the quality and usability criteria acceptable to the general public. Typical examples are Apple’s iPhone or Google’s search engine. While there were touchscreen devices before iPhone (e.g., IBM Simon) and search engines before Google (e.g., AltaVista), they were not ready for mass adoption.

Another reason relates to shortening the feedback loop and getting immediate opinions from the users. The feedback from the production environment enables companies, products, and engineers to pivot and make corrections in the project early. The ability to course-correct avoids situations where engineers learn that their architecture, design or strategy is incorrect only after years of working on a software project [Ji & Sedano, 2011].

One more unfortunate reason is related to being able to patch software fast. In the era where various cloud services and mobile devices are an integral part of
our lives, a zero-day vulnerability can potentially disrupt the lives of billions of people. An organization’s capacity to review, test, and deploy code changes is critical to its reputation.

From a purely people-centric point of view, increase in code velocity helps to recruit and retain engineers. A company that enables engineers to quantitatively observe the results of their efforts in days, rather than years, has a competitive hiring advantage.

### 1.1.2 Historical developments and status quo

Historically the accepted range for code velocity has significantly shortened during the last 2–3 decades. In the 1990s and early 2000s, the projected time to complete a new version of an operating system, such as Microsoft Windows, was 3–5 years [Custer, 1993; Lucovsky, 2000; Zachary, 1994]. For example, in the author’s experience, an engineer working on a major product at Microsoft would commit (at a time, Microsoft used the term “check in” [Perforce Software, 2022]) the code changes in 2003, and those changes would finally reach a customer sometime between 2006 and 2008. A decade or so later, the technologies such as Continuous Integration (CI) and Continuous Delivery/Continuous Deployment (CD) would fundamentally disrupt the existing development methodologies that were mainly based on a waterfall model. In 2013, a code change made by a Facebook (now known as Meta) engineer on Monday, would reach millions of users on Wednesday or Thursday [Feitelson, Frachtenberg, & Beck, 2013; Kushner, 2011]. Instead of years, millions of people would have the software they consume updated in the span of days.

The competitive nature of the software industry is changing even the belief system that error-free software is what matters the most. Meta’s development philosophy from the early years has been to “move fast and break things” [Feitelson et al., 2013; Frenkel & Kang, 2021; Kushner, 2011]. Even a decade later, Mark Zuckerberg’s message to employees and investors amid an economic downturn describes the critical internal investments that intend to “help engineers write better code faster” and “spend more time iterating and less time waiting” [Zuckerberg, 2023]. The “good enough software” era that Edward Yourdon wrote about in 1995 has finally arrived [Yourdon, 1995]. As horrifying as it sounds to the non-compromising purists, sometimes it is practical to accept a trade-off between time-to-market and fixing all the known defects in the software.

The effort to maintain an acceptable level of code velocity is akin to running a marathon. The main goal of engineering and research activities related to
code velocity is to avoid slowing down and, if possible, increase the pace. As Frederick P. Brooks Jr. famously wrote, “Question: How does a large software project get to be one year late? Answer: One day at a time!” [Brooks, 1995]. Immediate deployment is a theoretical lower bound for the newly developed code changes to reach the production environment. Unlike a mathematical challenge, such as providing a formal proof for a theorem, the “problem of code velocity” does not have a definitive solution. Everything from the development methodology changes to toolset improvements is an optimization target. For example, Meta’s internal analysis shows that engineers with faster build toolsets “produced meaningfully more code” [Zuckerberg, 2023].

1.1.3 Industry versus open-source

An argument can be made that code velocity is a problem specific only to commercial software. The Linux kernel development guide says that “[t]he goal is to get the code right and not rush it in” [The Linux Foundation, 2022]. Could it be? Can the engineers working on open-source software not worry about ensuring that their changes deploy on time, are merged daily, or are reviewed in less than a certain number of hours? That premise may have been correct in the early years of the open-source software movement. During the initial stages of the open-source revolution most contributors were volunteers, engineers were self-selecting to work on the projects, and there was a certain anti-establishment aura associated with the open-source movement [Raymond, 1999]. Modern open-source software development has changed. As of December 2022, most contributions to the Linux kernel come from commercial companies such as AMD, Google, Intel, and Oracle [Corbet, 2022; Linux, 2023; Marsden, 2022]. Even Microsoft, the former nemesis of the open-source software movement, now has its own Linux distribution [Microsoft Corporation, 2023]. It is reasonable to assume that the companies involved in Linux kernel development are interested in ensuring that the patches they commit will be merged into the Linux kernel mainline sooner than later.

1.1.4 Code reviews as an optimization target

There are many different activities and phases between making the code changes and those changes reaching the customer. This dissertation focuses on a subset of the pipeline through which the code changes are integrated to reach the production environment. We focus on the code review process and how to optimize it. Other
1.1. Foundations

phases of the software development process, such as the design, testing, and deployment, differ between individual companies, projects, and methodologies. The code review process is one of the few activities with considerable similarities across various software projects. The findings from analyzing code reviews apply to a heterogeneous landscape of software projects.

In more detail, the reasons to optimize code reviews are:

- **Standardization of the process and terminology.** Each software engineer working with other individuals will eventually be involved with the code review process. A specific workflow may have minor differences between various companies or projects, but in this era it is conceptually similar and follows the Modern Code Review process [Bacchelli & Bird, 2013; McIntosh, Kamei, Adams, & Hassan, 2015; Sadowski, Söderberg, Church, Sipko, & Bacchelli, 2018]. Terms and operations such as branch, code review, commit, merge, patch, and source lines of code are widely adopted across different companies, projects, programming languages, and types of software. The definitions behind these terms are standard across the industry. Unified terminology and metrics enable comparison between various projects, assist with developing universal solutions, and serve as lingua franca between different subcultures in the software industry.

- **Common ground and familiarity.** The steps that precede and follow the code review depend highly on variables such as a particular component, deployment schedule, specific problems, and the project. Code review is a part of the continuous integration and delivery pipeline that all engineers participate in, are familiar with, and can influence. That makes code reviews a natural optimization target.

When the code changes have reached the main development branch, most software engineers’ involvement in the deployment process ends. Committing the code changes means an engineer has de facto completed their assigned task. This practice makes the code review process a last chance for engineers to help to increase code velocity. The DevOps methodology has become increasingly popular, and more engineers are familiar with product deployment. However, Production Engineers (Meta) or Site Reliability Engineers (Google) are still responsible for most decisions and steps that follow the code review process.

- **Availability of the data.** Several confounding and lurking variables can influence other parts of the deployment process. The data about various
economic, organizational, or political variables that influence the decisions about product deployment are typically not recorded anywhere. If they are, then rarely is that information accessible to researchers outside the particular project or organization. As a result, only secondhand anecdotal evidence is available to researchers. Code collaboration tools such as Gerrit, GitHub, and Phabricator act as repositories of information for the code review process. Depending on the tool, the code review data is associated with an actor, timestamp, and other characteristics that describe the code changes. The availability of public data enables us to use various statistical techniques to draw inferences about the code review process.

Our main goal is to determine what parts of the code review process we can make more efficient, what variables controlling the code review process we can tune to increase code velocity, and how to prioritize the optimization process between different system layers.

1.2 No easy solutions

Let us first look at the apparent solution related to increasing code velocity—the authoritarian approach. Many of our behavioral patterns in society and the workplace are caused by social contracts and consequences that result from their enforcement. We do not commit violent crimes because we would most likely go to prison. We do not miss work without a reason because otherwise, we would get fired. If the code velocity is so vital to the industry, why not apply the “or else” method to reduce the time of code reviews? Rules such as “an engineer must respond to each code review in 8 hours” or “each change must be deployed to production in less than 72 hours” can be enforced. Meet them . . . or else.

There are multiple problems with this approach in the software industry. Give or take the occasional economic downturn, for the last couple of decades, software engineers have enjoyed “seller’s market.” If engineers are annoyed or unhappy with their work environment, they will get new jobs. Suppose they cannot do that for various reasons. In that case, all the metrics related to code velocity can be easily manipulated like any other metrics used to evaluate the engineer’s or project’s performance [Weinberg, 1998]. Targeted optimization of various software metrics that either the organization or market uses to evaluate engineers or a product is something that the author has frequently observed in the industry. As an anecdote, the author learned about this for the first time as an undergraduate computer science student. One of the author’s professors who
used to work in a research institute in the former USSR used to give an example of bureaucrats who decided that programmers will be compensated per line of code they produced. As a result, the programmers working on a particular project decided to type up the poems of the Russian poet Alexander Pushkin in the comments to ensure that they receive their annual bonuses.

Enforcing the punitive metrics will likely have a similar effect. Engineers will optimize acquiring the rewards by conducting cursory code reviews and deploying untested code into production. Eventually, the consequences of this behavior will cause more harm than benefits. The solutions we search for need to be something engineers are willing to use without an enforcement mechanism in place.

In the research on developer productivity, the need to improve code velocity is considered a priori knowledge [Greiler, 2020; Killalea, 2019]. George Orwell wrote in “1984” that “Oceania had always been at war with Eastasia” [Orwell, 1991]. Similarly, in the software industry, the belief is that code velocity has always been a problem, and the fight against “slowness” must never cease. The existing data, research, and investments across the industry (e.g., Microsoft has a research organization called Developer Velocity Lab) [McMartin, 2021; Microsoft Research, 2023] and open-source software indicate that slow code velocity is a critical and almost existential problem. Nothing in the observable data indicates the presence of black swan type of surprises in the area of code velocity [Hakan, 2021; Taleb, 2010].

The industry’s metrics to measure code review efficiency and code velocity are quantitative [L. Chen, Rigby, & Nagappan, 2022; Czerwonka, Greiler, Bird, Panjer, & Coatta, 2018; GitHub, 2021c; Riggs, 2022]. Quantifiable measurements, such as the code review duration, response times, completed reviews per developer, and review size, serve as indicators of the project’s success. Multiple commercial organizations track that data to evaluate projects’ performance [Czerwonka et al., 2018; Izquierdo-Cortazar, Sekitoleko, Gonzalez-Barahona, & Kurth, 2017]. Some organizations also use these metrics to evaluate the performance of engineers.

One focus area in the research related to code velocity is the ability to predict the duration of code reviews. 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” [L. Chen et al., 2022]. However, a study based on Gerrit code reviews finds that “ML models significantly outperform baseline approaches with a relative improvement ranging from 7% to 49%” [Chouchen, Ouni, Olongo, & Mkaouer, 2023].
Another part of the research focuses on various tools and techniques to speed up code reviews. These approaches include optimizing the code review strategy [Gonçalves, Fregnan, Baum, Schneider, & Bacchelli, 2020], investigating the effectiveness of bot usage to automate the code review process [H. Kim, Kwon, Joh, et al., 2022; H. Kim, Kwon, Kwon, 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, Tantithamthavorn, & Thongtanunam, 2022], targeting the optimal reviewers [Thongtanunam, McIntosh, Hassan, & Iida, 2015], and improving automation to suggest reviewers [Zanjani, Kagdi, & Bird, 2016]. The published results show that repeatedly notifying, i.e., nudging developers, reduced the number of stale code reviews and the time code reviews take.

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, Church, Börstler, Niehorster, & Rydenfält, 2022] and papers about what factors impact the code review decisions [Kononenko, Baysal, & Godfrey, 2016; Kononenko et al., 2018].

1.3 On the author’s industry experience

In multiple instances, this dissertation makes use of industry experience. The term industry experience can mean a variety of things. It may describe an individual who worked a quarter of a century in a small software house or switched jobs between Silicon Valley startups every few years. Context and the implications derived from it are everything. This section contains a high-level overview of the author’s experiences in software engineering so that a reader can better understand the author’s perspective. Appendix C.1.1 includes the author’s concise resume.

Between 1995 and 2000 author worked as a part-time programmer (the term software engineer would have been overkill to describe the author’s role) for two smaller software houses in Estonia. When not working, the author was either an undergraduate or graduate computer science student. Depending on the company, it employed 4–30 programmers during the various growth phases. In retrospect, the primary experience that the author acquired was about how not to conduct activities related to software engineering and how not to manage software projects or people.
1.4 Research questions

After repeatedly reading through an ancient edition of the cult classic software engineering textbook [Sommerville, 2015] and being encouraged by a friend (thank you, Targo), the author decided there must be a better way to produce software.

Between 2000 and 2015, the author worked in various software engineering roles at Microsoft Corporation (in Redmond, WA, USA, with a two-year tour in a remote development center in Zurich, Switzerland). The invaluable experience that the author gathered there was how to work on multi-year software projects that require coordination between thousands of people to complete. The author’s focus was related to systems programming, server applications, and operating systems development. This period is something that the author considers to be his formative years as a software engineer.

In 2015, the author followed some of the engineers he highly respected to Facebook, Inc. (now known as Meta, Inc.). At Facebook (in Seattle, WA, USA), between 2015 and 2020, the author learned that producing software that billions of people use daily is possible without sticking to a waterfall model and waiting years for the code changes to reach users. In many ways, the experience at Facebook was directly the opposite of how Microsoft worked.

The author is immensely grateful to have worked at Facebook and Microsoft because he had the opportunity to learn from some of the world’s best software engineers and be at the sharp end of his chosen profession.

1.4 Research questions

In RQ1 (“Is there publicly accessible high-granularity code review data?”) (Chapter 2), we investigate the availability of data that we can use to evaluate the various phases of the code review process. As a result, we publish a dataset of Phabricator code reviews and document the taxonomy of various smells related to mining code review data. That dataset serves as a foundation for our subsequent studies.

During the research process, the author realized that existing public research needs to answer the question about the trend of code velocity across multi-year development history for projects of non-trivial size. Because of confidentiality reasons, it is unlikely that any of the internal corporate data will ever be published and outside researchers are able to access it. To anonymously quote one of the empirical software engineering researchers in the industry, “[T]he good stuff is in the vault, and it will stay there.” Armed with Phabricator field data, we
investigate in RQ2 (“What is the trend of code velocity across various software projects?”) (Chapter 2) if the code velocity is increasing or decreasing during the lifetime of the several major accessible open-source software projects.

Based on our findings that indicate either a flat trend or minor improvements, we proceed with RQ3 (“Does the size of code changes correlate to the duration of various code review phases?”) (Chapter 3). The fact that code velocity does not improve over time indicates the need to investigate what factors a project can tune to reduce the code review time. The size of code reviews is one of the few quantifiable attributes that industry and open-source projects use to allegedly make the code review process more efficient. The results from RQ3 tell us that we cannot use only the size of code changes to optimize the code review process. While we eliminated one of the existing software engineering myths, this finding alone does not help increase the code velocity.

Next, we focus on the potential inefficiencies in the code review process to investigate other possible optimization means. In RQ4 (“What phases of the code review process are inefficient, and what can we improve?”) (Chapter 4), we look holistically at the entire code review process and the potential optimizations. We determine two potential optimization targets: (a) the wait time from the proposal of code changes until the first response and (b) the wait time between acceptance and merging. Given the cost of experimentation and any process changes, engineers need to prioritize what layers of the software system need the most improvements.

To determine where practitioners should focus their efforts, we look at the difference between various layers of a more extensive system—an operating system. An operating system is a prime example of a comprehensive complex software system with well-defined abstraction layers. One way to classify an operating system’s abstraction layers is to separate them into the kernel and non-kernel code. In RQ5 (“Does the code velocity differ between kernel and non-kernel code?”) (Chapter 5), we investigate the differences in the code velocity between kernel and non-kernel code. We use the code review data from the FreeBSD operating system. We find that code reviews for kernel code are slower than for non-kernel code. That finding confirms our empirical observations and provides input to practitioners about where to target their optimization efforts.

In the author’s industry experience and observations over two decades, dead code and compiler warnings are responsible for the increase in code review iterations and the time it takes for an engineer to submit their code changes for a review. As a result, we investigate two other ways to improve code velocity: removing dead code and reducing the amount of work that goes into fixing
1.5. Research methods

Various empirical methods are used to research software engineering [Shull, Singer, & Sjøberg, 2008]. Methods such as case studies, controlled experiments, and survey research are frequently applied to selected research problems. The choice of research methods is dictated by the questions that investigators ask. Most of the research methods used in this dissertation are quantitative. The only exception is the study described in Chapter 7. In Chapter 7, we use qualitative survey research to ask several questions about code velocity from our representative sample. Based on the answers from our respondents, we use different data analysis techniques to generalize the answers. We enumerate the mapping of research questions into chapters and research methods in Table 1.1.

In Chapter 2, we ask an existence question [Shull et al., 2008] about the presence of the code velocity trend in various open-source software projects. The existence question is an inquiry: "Does X exist?" In addition, Chapter 2 conducts an exploratory case study to search for datasets containing information related to code reviews with the required granularity. In Chapter 3, we ask a relationship question about the correlation between the size of code changes and various code review periods. The relationship question [Shull et al., 2008] answers an inquiry in the form of: "Are X and Y related?" In Chapter 4, we conduct an exploratory case study to investigate if some code review periods can be optimized. In Chapter 5, we ask descriptive-comparative questions about compiler warnings. In RQ6 (“Should we delete dead code and stop fixing compiler warnings?”) (Chapter 6), we investigate if suggestions based on anecdotal evidence have any merit. We recommend ignoring compiler warnings as another technique to increase code velocity and further research into the benefits of deleting dead code.

During our research, we observed a set of differing beliefs, priorities, and attitudes related to code velocity between industry and open-source engineers. To understand if our observations are generalizable, we conducted a survey. In RQ7 (“What are the beliefs, practices, and convictions related to code velocity?”) (Chapter 7), we investigate what industry and open-source software developers think about various issues associated with code velocity. The findings in Chapter 7 indicate that despite all the technological advancements, such as advanced code collaboration tools, faster hardware, and improved IDEs, code velocity issues are still an essential concern for engineers.
Chapter 1. Introduction

<table>
<thead>
<tr>
<th>Research question</th>
<th>Empirical method</th>
<th>Data type</th>
<th>Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RQ1</strong> (Is there publicly accessible high-granularity code review data?)</td>
<td>Exploratory case study</td>
<td>Quantitative</td>
<td>Chapter 2</td>
</tr>
<tr>
<td><strong>RQ2</strong> (What is the trend of code velocity across various software projects?)</td>
<td>Exploratory case study</td>
<td>Quantitative</td>
<td>Chapter 2</td>
</tr>
<tr>
<td><strong>RQ3</strong> (Does the size of code changes correlate to the duration of various code review phases?)</td>
<td>Exploratory case study</td>
<td>Quantitative</td>
<td>Chapter 3</td>
</tr>
<tr>
<td><strong>RQ4</strong> (What phases of the code review process are inefficient, and what can we improve?)</td>
<td>Exploratory case study</td>
<td>Quantitative</td>
<td>Chapter 4</td>
</tr>
<tr>
<td><strong>RQ5</strong> (Does the code velocity differ between kernel and non-kernel code?)</td>
<td>Exploratory case study</td>
<td>Quantitative</td>
<td>Chapter 5</td>
</tr>
<tr>
<td><strong>RQ6</strong> (Should we delete dead code and stop fixing compiler warnings?)</td>
<td>Ethnography, Literature review</td>
<td>Qualitative</td>
<td>Chapter 6</td>
</tr>
<tr>
<td><strong>RQ7</strong> (What are the beliefs, practices, and convictions related to code velocity?)</td>
<td>Survey research, Grounded theory</td>
<td>Qualitative</td>
<td>Chapter 7</td>
</tr>
</tbody>
</table>

Table 1.1: Empirical methods used to answer the research questions.

The differences in code churn and code velocity between kernel and non-kernel code. A descriptive-comparative question studies how context impacts cause-effect relationships [Shull et al., 2008]. In Chapter 6, we pose description and classification questions about quantifying the impact of removing dead code and the benefits of fixing compiler warnings. A classification question [Shull et al., 2008] is an inquiry in the form of: "How can we measure X?" All of our quantitative studies rely heavily on data mining and data science. Data mining is “the process of extracting useful information from large data sets through the use of any relevant data analysis techniques” [S. B. Kim & Sukchotrat, 2010]. Data science is a “concept to unify statistics, data analysis, machine learning and their related methods” to “understand and analyze actual phenomena” with data [Felderer & Horta Travassos, 2020; Hayashi, 1998].

1.6 Primary metrics

The metrics we use to describe the duration of code reviews are standard throughout this dissertation. We introduce the critical terminology in this section to help with comprehension of the Modern Code Review process. Various subsections (Section 3.3.3, Section 5.2.2, and Section 7.2.2) describe the evolution of terminology in the context of a particular chapter in more detail.
Inspired from the range of definitions available to describe code review time, we choose the following metrics to characterize code review process:

- **Time-to-first-response** [Bird, Carnahan, & Greiler, 2015; MacLeod, Greiler, Storey, Bird, & Czerwonka, 2017]: is defined as the time from the publication of a code change until the first acceptance, comment, inline comment (comment on specific code fragments), or rejection by a person (excludes bots) other than the author of the code.

- **Time-to-accept** [Bird et al., 2015]: refers to the time from when a code review is published for review until the acceptance of the code review by someone other than the author of the code (excludes bots).

- **Time-to-merge** [GitHub, 2021c; Kononenko et al., 2016]: is defined as “…the time since the proposal of a change (…) to the merging in the code base…” [Izquierdo-Cortazar et al., 2017].

Figure 1.1 represents the relationship among these metrics.

Figure 1.1: Code review life cycle (using CI) when the proposed changes are accepted for inclusion. A code review can be abandoned by an author or rejected by a reviewer at any point during its life cycle.

### 1.7 Dissertation overview

The main contributions of this thesis are contained in six chapters (Chapters 2–7). Most of the chapters’ contents are based on papers previously published at various peer-reviewed conferences. In chronological order, the papers that form
the contents of a subset of Chapter 2, a subset of Chapter 6, Chapter 3, Chapter 4, Chapter 5, and another subset of Chapter 6 were published first. The main reason for this order of investigations were the existing beliefs about the well-understood nature of problems related to code velocity in the software industry, such as code velocity being crucial to an organization’s survival and the constant need to increase it.

The authorship of the papers that form this dissertation and their relationship to the other authors all follow the same pattern. The Ph.D. candidate was the first author and was responsible for the main contributions in all the papers. Dr. Ayushi Rastogi acted as a daily supervisor and guided the author’s thinking process and understanding of the rules related to academic research. One of the non-negotiable conditions for this dissertation was the requirement that the dissertation is helpful for and validated by the practitioners. All of the papers have coauthors who are either current or former software engineers or researchers at Meta, Inc. (formerly known as Facebook, Inc.), Microsoft Corporation, and Snap, Inc. The Ph.D. candidate used the coauthors mainly as early intellectual adversaries and opponents to verify that the ideas, methodology, and conclusions make sense to the individuals facing daily problems that this dissertation studies. Because of the coauthors’ wide range of experiences, they also pointed out certain engineering-related biases that the author exhibited and took measures to mitigate them.

Each chapter intends to answer or eliminate one research question we deem essential. We summarize the contents of Chapters 2–7 below. Finally, in Chapter 8, we come to the dissertation’s end, summarize the main findings, and enumerate the opportunities for future work.

Chapter 2 is based on two papers published in the Proceedings of the 19th and 20th International Conference on Mining Software Repositories (MSR 2022 and MSR 2023) [Kudrjavets, Nagappan, & Rastogi, 2022b, 2023]. The first paper documents and contributes a new dataset of Phabricator code reviews to Open Science. In addition, we introduce a taxonomy of smells related to data mining the code reviews. Based on the available information, we are the first researchers to categorize various issues that must be accounted for when correctly interpreting the data from code reviews. The second paper investigates the multi-year code velocity trend across major open-source software projects.

Chapter 3 is based on a paper published in the Proceedings of the 19th International Conference on Mining Software Repositories (MSR 2022) [Kudrjavets, Nagappan, & Rastogi, 2022a]. That paper states that there is no relationship between the size of code changes and various code review periods such as time-to-first-response,
time-to-accept, and time-to-merge. Our findings are based on analyzing hundreds of thousands of code reviews from Gerrit, GitHub, and Phabricator. When the paper was published, it was antithetical to accepted beliefs in the industry and open-source communities. Our controversial findings were confirmed in 2022 by software engineering researchers from Meta, Inc. [L. Chen et al., 2022], who, based on industry data, found that the size of code changes is not a valid predictor for how long the code reviews take.

Chapter 4 is based on a paper published in the Proceedings of the 19th International Conference on Mining Software Repositories (MSR 2022) [Kudrjavets, Kumar, Nagappan, & Rastogi, 2022a]. We analyze various phases of the Modern Code Review process and look for optimization options that organizations and projects can use. In the current model, much time is wasted waiting for a human to perform a rudimentary action. Our study indicates that reducing the time between acceptance and merging can speed up Phabricator code reviews by 29–63%. To reduce the frustration in the code review process, we determined that small code changes by the authors with a large number of previously accepted code reviews have the highest probability of being accepted immediately without any further iterations.

Chapter 5 is based on the paper published at the 38th IEEE International Conference on Software Maintenance and Evolution (ICSME 2022) [Kudrjavets, Thomas, Nagappan, & Rastogi, 2022]. We aimed to investigate the code churn, evolution, and code velocity patterns in different layers of large software systems. One type of extensive software system is an operating system. We divide an operating system into the kernel and non-kernel code to match the existing division in operating systems research. We conduct a case study on four BSD family operating systems: DragonFlyBSD, FreeBSD, NetBSD, and OpenBSD. We analyze code velocity for a decade of code reviews in FreeBSD. One of our findings is that code reviews for kernel code take longer than for non-kernel code.

Chapter 6 succinctly discusses two topics we investigated as potential quick solutions to the code velocity problem: the removal of dead code and the unknown benefit of fixing compiler warnings. The short paper that discusses the lack of evidence for fixing compiler warnings was published at the 44th International Conference on Software Engineering (ICSE 2022) [Kudrjavets, Kumar, Nagappan, & Rastogi, 2022b]. The follow-up paper that suggests to use data mining to determine and rank compiler warnings by their usefulness will be published in 20th International Conference on Mining Software Repositories (MSR 2023) [Kudrjavets, Kumar, & Rastogi, 2023]. The abstract pointing out that there are no existing means to quantify the benefits of removing dead code was published in
the 38th IEEE International Conference on Software Maintenance and Evolution (ICSME 2022) [Kudrjavets, Rastogi, Thomas, & Nagappan, 2022].

Chapter 7 is based on paper that is presently undergoing the academic peer review process. In the paper, we discuss the survey results that seek information about the current beliefs, practices, and opinions about code velocity. We analyze and summarize the responses from the 75 industry and open-source engineers about the trade-offs related to code velocity and suggestions about how to improve it.