RESEARCH-ARTICLE

Leveraging Risk Models to Improve Productivity for Effective Code Un-Freeze at Scale

AUDRIS MOCKUS, Meta Platforms, Inc., Menlo Park, CA, United States

RUI ABREU, Meta Platforms, Inc., Menlo Park, CA, United States

PETER CHRISTOPHER RIGBY, Meta, Menlo Park, CA, United States

DAVID AMSALLEM, Meta Platforms, Inc., Menlo Park, CA, United States

PARVEEN BANSAL, Meta Platforms, Inc., Menlo Park, CA, United States

KAAVYA CHINNIAH, Meta Platforms, Inc., Menlo Park, CA, United States

View all

Published: 14 August 2025
Online AM: 11 March 2025
Accepted: 17 December 2024
Revised: 16 December 2024
Received: 23 April 2024

Leveraging Risk Models to Improve Productivity for Effective Code Un-Freeze at Scale

AUDRIS MOCKUS, Meta Platforms Inc Menlo Park, California, USA and University of Tennessee, Knoxville, Tennessee, USA
RUI ABREU, Meta Platforms Inc, Menlo Park, California, USA
PETER C. RIGBY , Concordia University, Montreal, Quebec, Canada and Concordia University, Montreal, Quebec, Canada and Meta Platforms Inc, Menlo Park, California, USA
DAVID AMSALLEM, PARVEEN BANSAL, KAAVYA CHINNIAH, BRIAN ELLIS, PENG FAN, JUN GE, BINGJIE HE, KELLY HIRANO, SAHIL KUMAR, AJAY LINGAPURAM, ANDREW LOE, MEGH MEHTA, VENUS MONTES, MAHER SABA, GURSHARAN SINGH, MATT STEINER, WEIYAN SUN, and SIRI UPPALAPATI, Meta Platforms Inc, Menlo Park, California, USA
NACHIAPPAN NAGAPPAN, Meta Platforms Inc, Palo Alto, California, USA

Changing software is essential to add needed functionality and to fix problems, but changes may introduce defects that lead to outages. This motivates one of the oldest software quality control techniques: a temporary prevention of non-critical changes to the codebase—code freeze. Despite its widespread use in practice, research literature is scant. Historically, code freezes were used as a way to improve software quality by preventing changes during periods before software releases, but code freezes significantly slow down development. To address this shortcoming, we develop and evaluate a family of code un-freeze (permitting changes) strategies tailored to different occasions and products at Meta. They are designed to un-freeze the maximum amount of code without compromising quality. The three primary dimensions to un-freeze involve (a) the exact timing of (and the reasoning behind it) the code freezes, (b) the parts of the organization or the codebase where the codebase freeze is applied to, and (c) the method of screening of the code diffs during the code freeze with the aim to allow low risk diffs and prevent only the most risky diffs.

Authors' Contact Information: Audris Mockus (corresponding author), Meta Platforms Inc Menlo Park, California, USA and University of Tennessee, Knoxville, Tennessee, USA; e-mail: audris@utk.edu; Rui Abreu, Meta Platforms Inc, Menlo Park, California, USA; e-mail: rui@computer.org; Peter C. Rigby, Concordia University, Montreal, Quebec, Canada, and Meta Platforms Inc, Menlo Park, California, USA; e-mail: rigbypc@gmail.com; David Amsallem, Meta Platforms Inc, Menlo Park, California, USA; e-mail: amsallem@meta.com; Parveen Bansal, Meta Platforms Inc, Menlo Park, California, USA; e-mail: pb001@meta.com; Kaavya Chinniah, Meta Platforms Inc, Menlo Park, California, USA; e-mail: kanmanic@meta.com; Brian Ellis, Meta Platforms Inc, Menlo Park, California, USA; e-mail: bpe@meta.com; Peng Fan, Meta Platforms Inc, Menlo Park, California, USA; e-mail: pefa1@meta.com; Jun Ge, Meta Platforms Inc, Menlo Park, California, USA; e-mail: jakege@meta.com; Bingjie He, Meta Platforms Inc, Menlo Park, California, USA; e-mail: bingjiehe@meta.com; Kelly Hirano, Meta Platforms Inc, Menlo Park, California, USA; e-mail: hirano@meta.com; Sahil Kumar, Meta Platforms Inc, Menlo Park, California, USA; e-mail: sahilkum@meta.com; Ajay Lingapuram, Meta Platforms Inc, Menlo Park, California, USA; e-mail: ajaylin@meta.com; Andrew Loe, Meta Platforms Inc, Menlo Park, California, USA; e-mail: loe@meta.com; Megh Mehta, Meta Platforms Inc, Menlo Park, California, USA; e-mail: meghmehta@meta.com; Venus Montes, Meta Platforms Inc, Menlo Park, California, USA; e-mail: vmd@meta.com; Maher Saba, Meta Platforms Inc, Menlo Park, California, USA; e-mail: mahersaba@meta.com; Gursharan Singh, Meta Platforms Inc, Menlo Park, California, USA; e-mail: gurshi@meta.com; Matt Steiner, Meta Platforms Inc, Menlo Park, California, USA; e-mail: mattsteiner@meta.com; Weiyan Sun, Meta Platforms Inc, Menlo Park, California, USA; e-mail: wysun@meta.com; Siri Uppalapati, Meta Platforms Inc, Menlo Park, California, USA; e-mail: siri@meta.com; Nachiappan Nagappan, Meta Platforms Inc, Menlo Park, California, USA; e-mail: nnachi@meta.com.

ACM 1557-7392/2025/8-ART211
https://doi.org/10.1145/3722216

To operationalize the drivers of outages, we consider the entire network of interdependencies among different parts of the source code, the engineers that modify the code, code complexity, and the coordination dependencies and authors' expertise. Since the code freeze is a balancing act between reducing outages and allowing software development to proceed unimpeded, the performance of the various approaches to code un-freeze is evaluated based on the fraction of flagged/gated changes to measure overhead and the fraction of all outage-causing changes contained within the set of flagged set of changes to measure the ability of the code un-freeze to delay (or prevent) outages. We found that taking into account the risk posed by modifying individual files and the properties of the change we could un-freeze 2 and 2.5 times more changes correspondingly.

The change level model is used by Meta in production. For example, during the winter 2023 code freeze, we see that only 16% of changes are gated. Although 42% more changes landed (were integrated into the codebase) compared to the prior year, there was a 52% decrease in outages. This reduction meant less impact on users and less strain on engineers during the holiday period. The risk model has been enormously effective at allowing low-risk changes to proceed while gating high-risk changes and reducing outages.

CCS Concepts: • **Software and its engineering → Software defect analysis**; **Software maintenance tools**; **Risk management**;

Additional Key Words and Phrases: System outages, code freeze, defect prediction

ACM Reference format:
Audris Mockus, Rui Abreu, Peter C. Rigby, David Amsallem, Parveen Bansal, Kaavya Chinniah, Brian Ellis, Peng Fan, Jun Ge, Bingjie He, Kelly Hirano, Sahil Kumar, Ajay Lingapuram, Andrew Loe, Megh Mehta, Venus Montes, Maher Saba, Gursharan Singh, Matt Steiner, Weiyan Sun, Siri Uppalapati, and Nachiappan Nagappan. 2025. Leveraging Risk Models to Improve Productivity for Effective Code Un-Freeze at Scale. *ACM Trans. Softw. Eng. Methodol.* 34, 7, Article 211 (August 2025), 24 pages.
https://doi.org/10.1145/3722216

1 Introduction

Of the three key objectives of the software development process (shorter lead time, lower effort, and better quality), the quality aspect is often considered to be the most important. Low quality may manifest itself in various ways. At Meta, software is primarily used to run various internal and external services. Cases when the service does not operate or operates incorrectly are considered to be service outages and are referred to as **site events (SEVs)**. Such quality problems may lead to user dissatisfaction and are a prime target of the quality improvement efforts at Meta as in many other companies.

The fundamental challenge in software development quality improvement efforts is the inverse relationship between the delivery of software changes (referred to as diffs at Meta) that are necessary for enhancements or fixes to existing problems and outages (SEVs) these changes introduce. More numerous, especially rapidly made, changes lead to more problems on one hand, but fixes and enhancements are essential to keep users satisfied. Thus, the need for any software business is to deliver cutting-edge enhancements without significant deterioration in software reliability. The simplest (and obvious) solution to avoid outages is to keep the software codebase unchanged. Unfortunately, this approach conflicts with the need for rapid delivery of new functionality and new products.

Software is highly non-homogeneous with respect to the chances that a modification will result in a problem (see, e.g., [28]: "1% of project files are involved in more than 60% of the customer reported defects"). This phenomenon can be exploited to limit the impact of code freezes on development not just by applying freezes for a short duration, but also to the most problematic parts of software

or to most problematic tasks. We refer to such selective application of code freeze as "code un-freeze" to signify that the approach allows a significant portion of changes to land even when the quality concerns are paramount. Many of the software defects are a result of unanticipated dependencies or interactions, e.g., [13]. The complicated web of explicit and implicit dependencies is a hallmark of any nontrivial software product and development team. We, therefore, consider ways to capture and quantify the structure of software and authoring dependency networks and use it to identify problematic areas and tasks and segregate from the rest of the software that could be safely un-frozen.

Much of software quality research is, unfortunately, not tightly integrated into the actual software development process [19, 45]. To address that and inspired by the **participatory action research (PAR)** approach [3, 47], we both investigate practices of code freezes at Meta while actively participating in their evolution by modifying approaches from existing software engineering literature to fit Meta's needs.

1.1 Research Questions (RQs)

Although code freeze (a temporary prevention of non-critical changes to the entire codebase or to the maintenance branch) is widely used in industry, the practice is not clearly defined in the research literature. We, therefore, start by investigating how the concept is defined and applied at Meta (see Section 5).

We then provide answers to the following RQs.

RQ 1. Current Practice: How Does Meta Currently Conduct Code Freezes?

We want to understand how code freeze was and is currently conducted at Meta. To answer this RQ, we worked with quality engineers and managers responsible for conducting freezes.

Result Summary. We found that code freeze definition and implementation was substantially different from those found in the research literature. First, Meta does trunk-only development so branch-specific freezes do not apply. Second, the timing of code freezes is not based on release maturity but on other factors, such as based on disruptions to developer teams, and reliability. Third, in Meta's code-freeze some of the diffs are allowed to land. Fourth, the permission to land during the code freeze is not based on the type of diff (enhancement vs. fix) but on diff risk and, even for risky diffs, on the diff gating process. Fifth, to determine the risk each manager from distinct areas of the codebase designates an area expert to specify *the code paths* that they deem to be the most important. Any diff to such files would be either blocked (the so-called "red-zone") or gated ("yellow zone"). If their diff is gated, and the engineer still wants to land the diff, they must provide a rationale and potentially obtain approval from the area expert.

In a historical analysis, when code freeze is applied by restricting diffs to the file paths selected by experts (`ExpertPaths` approach), we find that 27.3% of diffs would have been gated and 38% of SEVs would have been captured (here and below by "would capture" we mean "would have been predicted as risky").

RQ 2. `FileModel`*: How Well Does the File Risk Model Gate Diffs with SEVs?*

To improve upon the `ExpertPaths` approach, we first developed a file-level risk model to exploit the fact that only a small fraction of files are involved in SEVs. Statistical software risk models have been in use for 40 years, e.g., [8, 11, 16, 18, 29–31, 34], so we had a wide selection of options to choose from. The key aspects essential for practical deployment included the simplicity of the model and of the predictors to be able to develop, retrain, and deploy it rapidly in a very large software development organization, the ability to simply explain the resulting risk to software developers whose diffs are gated, and the ability to easily modify the gating level based on the type and objectives of the freeze. This led us to choose a logistic regression model with predictors that capture the complexity of software and work interdependencies, e.g., [6, 29], and properties of the

source code, e.g., [31]. The risk for all modified files is calculated and if risk for at least one file is above a threshold, then the diff is gated.

Result Summary. Had we used the `FileModel` to gate the same number of diffs as the `ExpertPaths` approach, the `FileModel` would capture 56% of SEVs or double the odds ratio of capturing the SEV-causing diff in comparison `ExpertPaths` approach. The most important features (by the term "feature" we indicate predictor used in the model) are: the number of prior SEVs, whether the code was part of the core, and the number of **lines of code (LOCs)** in the file.

RQ 3. `DiffModel`*: How Well Does the Diff Risk Model Gate Diffs with SEVs?*

We further tried to improve upon the `FileModel` by exploiting individual properties of each diff. The models of the software diff risk have been around for 20 years, e.g., [16, 29, 55]. Instead of calculating the risk for individual files, we calculate the risk for the entire code change, i.e., diff. We use the classic features, including prior SEVs and churn, as well as supply chain metrics. Not only can a diff level model better account for file-level risk, but it can also factor aspects related to the diff author and reviewers.

Result Summary. Had we used the `DiffModel` to gate the same number of diffs as the `ExpertPaths` approach, the `DiffModel` would capture 60.8% of SEVs (an odds ratio of 2.5 over `ExpertPaths` approach and 1.2 over `FileModel`). The most important features are the prior SEVs, number of reviewers, and number of files in the diff.

RQ4. Production: How Well Does the Risk Model Work in Practice?

The first major test of the risk model was in July which saw high traffic levels. The largest production usage of the risk model occurred over the holiday period of mid-November to the start of January 2023, where outages can significantly reduce user satisfaction.

Result Summary. Despite record traffic levels in July, the `FileModel` ensured that no major SEVs were introduced. In total, 26% of diffs were gated, 8.8% of all diffs (or 33% of gated diffs) were allowed to land after the developer provided landing rationale. The `DiffModel` was deployed for the winter holiday period and only 16% of diffs were gated, 5.7% were allowed to land after the developer provided landing rationale. When we compared to the prior year, we see a 42% increase in the number of landed diffs. Instead of seeing a corresponding increase in SEVs, we see a 52% decrease in the number of SEVs compared to last year. This reduction in SEVs also reduced the operating cost related to dealing with SEVs by 47% and meant fewer disruptions for engineers during the holiday period. The risk model has been enormously effective at allowing low-risk diffs to land while gating high-risk diffs and reducing SEVs.

This article is structured as follows. In Section 5, we review the industrial reports and literature on code freeze, and describe code un-freeze practices at Meta in Section 4.1. In Section 2.2, we present the history and state-of-the-art on risk modeling in software engineering. In Section 3.2, we introduce the features, risk models, and evaluation that we developed for use during code freezes at Meta. In Sections 4.3 and 4.4, we present the results for our file-based and diff-based risk models, respectively. In Section 4.5, we discuss how the models were used to do code un-freezes at scale in production. In Section 6, we discuss threats to validity. In the final two sections, we discuss our contributions in the context of the literature and present concluding remarks.

2 Background

To motivate our RQs we explain the essential aspects of software development at Meta, present general code freeze practices reported in the literature, and give a very short overview of the risk modeling in software engineering and how it can be exploited to identify low risk files or tasks that could remain unfrozen.

2.1 Background on Software Development at Meta

Meta develops software for both its servers and client devices, including specialized hardware devices. This approach facilitates swift software updates and provides meticulous control over versioning and configurations. At Meta, this deployment strategy has cultivated a routine of frequently "pushing" new code to production. Prior to any push, the code undergoes peer review, in-house user testing, and comprehensive automated and canary tests. Once deployed, engineers scrutinize logs to spot potential problems.

At Meta, it is customary for engineers to review each other's code. This process serves several functions. Firstly, it motivates the original coder to maintain high coding standards. Secondly, a reviewing engineer, with a fresh perspective, might detect flaws or propose better solutions. Thirdly, it promotes the dissemination of coding practices and specific code knowledge throughout the organization. Meta uses Phabricator[1] as the cornerstone of its CI system. This platform facilitates contemporary code reviews, wherein developers submit code changes, that are referred to as *diffs* at Meta, and provide feedback on their peers' changes before they are either integrated into the codebase or rejected.

Phabricator and version control systems are used as part of the development process. Phabricator tracks code diffs, author/reviewer actions, and the current state of all diffs. Developers submit their code for review, creating a *patch* representing the initial version of the code. Reviewers can suggest improvements, leading to additional revisions until the diff is either approved and incorporated into the code base (the diff "lands"), or until the diff is abandoned. A diff is a collection of patches representing the initial version of a bug fix or enhancement, along with any revisions made during the diff's lifecycle.

2.2 Background on Risk Modeling

Software risk modeling literature is based on the empirical fact that the chances of a defect are not homogeneous over time, code, and tasks. We start with the key aspects of software risk modeling and describe how it can be applied to the problem of code freezes at Meta.

2.2.1 Risk Prediction Literature. Statistical software risk models have been in use for 40 years, e.g., [8, 11, 16, 18, 29–31, 34]. Research that predicts the risk of individual changes are more recent, but still spans over 20 years, e.g., [16, 29, 55]. A proper review of this literature would require a book, and numerous survey papers exist, with some of the latest being [34, 55]. We focus only on aspects most relevant to our context. This involves the object for which the risk is predicted (source code file or change), the kind and operationalization of what is considered to be an adverse event, and the distribution of these adverse events.

The unit for which the risk is predicted could be a time period, a piece of code (such as a file or a module), or a change/task (i.e., diff). Since methods differ depending on the target unit, to discriminate between task and file risk prediction, the task risk prediction is often referred to as "just in time prediction" [16, 29]. The object of this work are models and methods to predict if a file (or a code change) will be involved in (cause/trigger) a software failure. Much of the literature is focused on "bug prediction." Unfortunately, the notion of a bug is typically defined implicitly (marked as bug in an issue tracking system) and such "bug" changes may constitute most of the development work (with over 75% of all changes in a project).

A subset of the literature considers only the "bug" changes that are actually affecting end users or customers of the software product. In this context, only a tiny fraction of changes are related to

[1] http://phabricator.org.

customer problems unlike for bug fixes found during regular course of software development. Rare events are much harder to predict and may require different kinds of models, e.g., [49].

Unfortunately, even these customer-impacting changes typically represent fixes to customer problems and not the causes/triggers of these problems. To identify the causes, researchers use heuristics that tag previous changes made on the same files, but the accuracy of such tagging may be extremely low [7, 32, 36, 40, 52] and that low accuracy has enormous impact downstream on just-in-time prediction [10]. We are aware of only a few (one) prior publications where changes were explicitly tagged by the development team as causing SEVs [29].

In our context the outcome measures involve not just the fraction of SEV-causing diffs (changes) identified, but also the fraction of diffs that were gated to account for the process overhead. This attempt to take into account the overhead of flagging something as risky is somewhat analogous to so-called effort-aware models, e.g., [15], where effort is spent on remediating units incorrectly flagged as risky. What, perhaps, is unique in our study is that the code-freeze process can be easily controlled by deciding what fraction of diffs should be gated. The role of the prediction models is then simply to maximize recall (fraction of SEV-causing diffs) given that specified level of gating. Conceptually it is important to distinguish failures (the manifestation of a bug/fault) from faults (bugs). The same fault may cause many different failures while a single failure may be the result of a confluence of factors, including multiple faults. Outages are typically a result of failures caused by defects in the software used to run the service. At Meta, SEV resolution process requires determining and recording the cause of the SEV. For some SEVs the causes are not rooted in software changes but represent environmental changes, for example, hardware or network problems, unexpected level of traffic, or other factors. The process to handle each SEV involves, among other things, identifying its trigger. While some of the triggers are not software related, in many cases triggers are identified as software diffs. Due to the many-to-many relationship between faults and failures, more than one diff may be identified as a trigger and the same diff may trigger multiple SEVs. These extremely valuable data linking SEVs to the diffs that caused them were then used to create models of file and diff risk. While such data may not exist in all companies, we hope that the value it could provide to balance quality and productivity, would encourage more companies to collect such data.

2.2.2 Precursors of Risk. While the risk prediction literature is focused on how to use properties of files or changes to predict the associated risk score, it is worth considering more generally why certain aspects of software or a change are affecting the chances of a failure. To characterize this complexity, it is worth observing that each software change is produced by a developer who does not have a perfect understanding of what behavior the change should implement in conjunction to their imperfect understanding of how the change might affect all different parts and aspects of a large software system. In other words, failures and SEVs are often caused by the inability to fully comprehend all possible ramifications the change brings not only to a particular piece of code but also to all parts of the code it may be related to. An elegant framework [13] illustrates such scenarios via the concept of constraint violation. Specifically, problems arise when the new constraints implemented by a change are not perfectly aligned with the constraints imposed by previous changes or when the systems environment violates the constraints of the existing implementation. The complexity of the software supply chain with deep and complex interdependencies makes it harder to know and satisfy all the explicit and implicit constraints, hence we postulate the first cause of software failures to be based on software interdependencies. Prior work defined three types of software supply chain [26, 27]: dependencies, code copying, and knowledge transfer. The dependencies are typically represented by the call graph where a function that is invoked is implemented in another package or a file. Co-changes (files changed in a single diff) are often used to represent logical dependencies. We do not use file copying as it is not

necessary at Meta due to the megarepo (a single repository) all Meta projects use and, therefore, can reuse each other's code without copying. The knowledge transfer is typically represented by the same person working on different parts of the code.

Furthermore, in a large system, many developers are working in parallel, so it is not unusual for them to have a slightly different understanding of the system. In case their mutual decisions violate certain constraints (as discussed above), the violation may manifest itself in unexpected program behavior or a fault leading to an SEV. In short, software development may be thought of as a complex network of software dependencies and knowledge transfer (the so-called software supply chains [26]), where the complexity of the network may lead to software problems. Thus, coordination problems that may be partially expressed via software authorship network (and its interactions with software dependency network) is our second postulated cause of failures.

Finally, since the earliest days of assessing software quality it was recognized that the complexity of the particular piece of software may affect the chances of failure. While numerous metrics were proposed to measure file and class complexity, the simple measures such as the lines of code in a file or a function or cyclomatic complexity, tend to suffice, e.g., [44].

The structure of the network introduces unevenness in terms of which parts of the code and which changes are responsible for most of the faults. For example, it was reported that 3% of the code is responsible for over 90% of customer-facing faults in switching software [28]. The state-of-the-art code inspection, code analysis, and testing practices are critical for [12], but they are not sufficient to achieve optimal results and require significant effort. This unevenness can, therefore, be exploited to focus quality improvement resources on the most problematic areas. In fact, many of the measures used to predict failures in the past are measuring the properties of the network, though not always explicitly. For example, the number of files modified by a diff, the number of past diffs to a file are, in fact, degree centrality of the co-change and authorship network correspondingly. Some work explicitly considers network measures explicitly, for example clustering coefficient used in [6], and social networks of authors [4, 43, 48, 51, 53, 56].

3 Methodology and Data

In this section, we discuss the method we used to understand the current state of code freezes at Meta and ways to adapt and deploy relevant research results into practice. We then describe statistical modeling methodology and the data we used to model risk, and the outcome measures to evaluate how effective the models are.

3.1 Discovering the Context

Not all software quality research stems from or is tightly integrated into the actual software development process. Inspired by the PAR approach [3, 47], we both investigate practices of code freezes at Meta while actively participating in their evolution by adapting approaches from existing software engineering literature to the specific organizational needs at Meta. To support overall organizational priorities at the time, the research team was engaged by the units tasked with planning code freezes. Initial meetings were devoted to better understand the history and current code-freeze practices as well as understand the perception of the remaining problems. Armed with this understanding the research teams suggested potential short- and longer-term approaches to address them that were then evaluated and prioritized by both teams and their upper management. Such a close alignment with all key stakeholders: software developers, management team, and quality engineers have been cited as critical to success in various contexts [33, 35, 46, 54]. The second important aspect for the success of this initiative was appropriate timing. The problem was considered a priority, the solutions implemented at the time were lacking (e.g., no diff- or file-specific risk, low recall), and the enhancements proposed by research team had undergone

an evaluation using historic data that showed significant promise [46]. A third factor that likely contributed to success was the significant effort spent by the research team to adjust and reinterpret existing research results for the specific needs of the organization [33] (e.g., the lack of maintenance branches, specifics of the measurement process). The fourth success factor was rapid prototyping of potential approaches, including evaluation of their performance, so that the management and development teams could rapidly move in the most promising direction [21]. For example, back-testing various strategies under realistic scenarios and providing direct comparisons between existing and proposed approaches. Fifth, the introduction of new techniques was designed to be evolutionary and iterative, based on the existing set of processes and tools with only necessary changes at each iteration. For example, we started with a simpler task of file-specific risk using preexisting metrics and gradually added more sophisticated metrics and moved to diff-level risk. Sixth, the success of the initial improvements generated the trust and motivation to rapidly deploy further improvements [46]. Seventh, the researchers involved in the effort have, over several years, built good understanding of organizations' general needs and the reputation for providing ideas and tools that lead to practical improvements in other contexts.

In short, the understanding of the code freeze needs was built by discussion with the management team, reliability engineers, and developers spearheading quality improvement efforts. Based on these interactions and retrospectives of previous code freeze efforts a detailed understanding of aims, results, and shortcomings of the existing code freezes was obtained. It was then used to create the three-dimensional code freeze framework and suggested ideas on how best to move forward. Since the effort was primarily driven by the engineering and quality teams it resulted in an extremely rapid transformation of research prototypes into the production tool chains.

3.2 Risk Modeling at Meta

As noted above, the key aspects essential for practical deployment included the simplicity of the model and of the predictors to be able to develop, retrain, and deploy it rapidly in a very large software development organization, the ability to simply explain the resulting risk to software developers whose diffs are gated, the ability to easily change the gating level based on the type and objectives of the freeze. This led us to choose a logistic regression model with predictors that capture the complexity of software and work interdependencies, e.g., [4, 6, 29], and properties of the source code, e.g., [31].

For an industry deployment, all measures need to be calculated in near real-time (as diff is landing) and, for at least some of them, easily understood by engineers as causes of risk. We, therefore, chose to rely on a very small group of measures that represent the complexity of code interdependences, coordination complexities, code complexity, and the expertise of authors involved in making these diffs. As discussed in Section 6, many alternative models had similar performance. Strikingly, all models required very few predictors to reach that performance.

3.3 Data Collection

While the precise features are described in the next section, here we explain data collection procedure Meta for SEV models. We collect data from code measurement systems, such as lines of code, cyclomatic complexity, line test coverage, and code ownership, including whether the file or the owner were specially designated (as described in the next section). A system to track *every* SEV was used company-wide to determine the root cause of every outage. Relevant for our purposes, it recorded a SEV trigger, which in case of software outages, was a diff or a set of diffs. Due to strong focus on customer experience, the accuracy of SEV data was considered a priority. We used Phabricator to obtain the remaining measures, such as diff creation and landing times, affected files, author, reviewers, and test plans. Hence, we linked SEV and diff data via diffs that

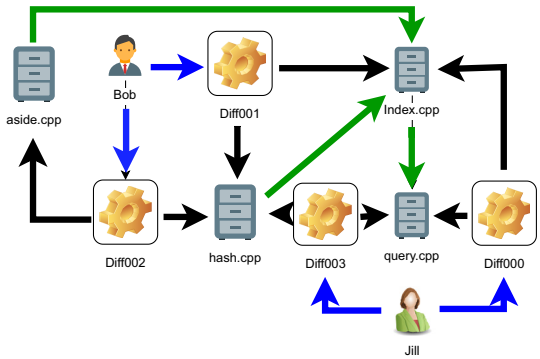

Fig. 1. Software supply network. Blue lines are author-to-diff, black diff-to-file, and green function call (file-to-file) links. File Aside.cpp is in the periphery with only a single author and co-changed with a single file, while index.cpp is more central with two authors, three other files either calling functions in it or being called from it, and co-changed with two files.

were triggers to SEVs and code metrics via file names modified by diffs and diff time (code metrics evolve over time as files are changed). From the supply chain network perspective, we obtained all the nodes represented by files, diffs, and authors from Phabricator. As shown in Figure 1 the links were co-changes (files modified by each diff), authorship (file-to-author via authored diffs), and call graph aggregated to file level. Since the call graph edges represent links from function invocation to function definition, there may be several functions defined in, for example, file query.cpp that are invoked from file index.cpp. By aggregation to file level, we mean that only one link is created from index.cpp to query.cpp if there is at least one call from the source code in file index.cpp to a function implemented in file query.cpp. The network was dynamic as each diff has time attributes.

Degree centrality is the simplest network measure, such as the number of reviewers or files for a diff. The main issue with the measure is that it does not take into account the importance of the nodes these edges lead to or from. We used Katz centrality [17] to take into account not just the number of other nodes (degree centrality), but also the importance of these nodes. We calculate centrality using the complete co-change, authorship, and call graph links to ensure the technical structure transfers to and is affected by the social structure. For example, many files may depend on a specific file F. Any developer who modifies F inherits some of that centrality by being linked to the central node.

3.4 Model Features

At a high level, we went through the following procedure for model selection (a similar procedure was used on a regular basis to update the production model as well). First, for each dimension we selected approximately five high-level measures previously described in research literature and then we investigated correlations among operationalizations of all resulting measures. Software engineering measures tend to be highly correlated, e.g., [22, 23], but models with highly correlated measures are harder to interpret, have unstable estimated coefficients, and poor predictive power, e.g., [9]. For our purpose, we evaluated models by fitting them on a year of historic data and evaluating based on their performance on (non-overlapping) 6 months of current data. To prevent hard-to-explain models, we selected at least one measure from each area (complexity, interdependencies, coordination, and experience) that was easiest to interpret (developers want to know why their diff is gated) and simple to calculate (the risk calculation had to be done in real time during diff landing). We also included all remaining measures that had a lower than 0.7 correlation with

Table 1. Features for `FileModel`

Measure	Description	Calculation
$had\_sev(f,t)$	Has the file been modified by an SEV-causing diff in the past?	$1 \iff \exists d : d_f \wedge d_{sev} \wedge d_t < t;\ \ else\ 0$
$past\_diffs(f,t)$	Number of times the file has been changed in the past	$\lvert\{d : d_f \wedge d_t < t\}\rvert$
$is\_core(f)$	True if the file belongs to an org that provides code with critical online services to most other orgs	
$test\_cvg(f,t)$	Percent of lines covered in unit tests for file f at time t	
$loc(f,t)$	Number of lines of code in the file f at time t	

For rapid deployment only the most basic easy-to-compute features were included. f: file, d_f: indicator that diff d modifies f, d_{sev}: indicator that diff d caused a sev, d_t: diff d landing time t. When diff d lands, maximum risk over the modified files is calculated: $r(d) = \max_{\{f:d_f\}} r(f, d_t)$.

this initial set. While initially we considered only approximately 20 candidate measures, eventually it grew to almost 100, though only 15 or so were used at a time for the best-performing production model. While all of the measures were related to gauging complexity of one sort or another, we also included a more direct measure of risk: whether or not any of the files modified by the diff have been previously modified by SEV-causing diffs. Finally, we fitted a logistic regression model on the training dataset. Measures that did not significantly contribute to explaining the variance (p-value $< .1$ or the difference in deviance under 2.6 for the likelihood ratio test if the addition increases likelihood significantly) were excluded from the model. Resulting models were then used to predict SEV-triggering diffs on the test datasets.

The measures used in `FileModel` and `DiffModel` are shown in Tables 1 and 2, respectively. The `FileModel` contains a limited set of features that could be pre-calculated and immediately used in production. The `DiffModel`, being the second iteration of the deployment, could contain more advanced features, such as the centrality of the author, that are more difficult to compute and are only gradually being rolled out in production.

More central file nodes would point to files that are often called or have methods called from many other files, have many authors, are frequently co-changed, and are connected to other files that are central. It thus captures almost all of the dimensions known to be associated with faults. We do not collapse the author-commit-file multi-graph into author-only or file-only networks as was commonly done previously [4, 6, 43, 48, 51, 53, 56].

The number of files in a diff (degree centrality) serves as a proxy of coordination requirements previously found to increase risk, e.g., [29]. The entirely new set of files modified by the diff makes it unlikely that the functionality implemented there would be immediately used (and hence cause a SEV). The number of reviewers may indicate that a diff requires expertise from several domains. It was previously found that author expertise as measured via past diffs to the same area of the codebase reduces risk, e.g., [29].

The size of the file is a proxy of complexity and larger files contain more code, so even if the SEV would be equally likely to affect any particular line in the codebase, large files would more likely contain it. The amount of code changed in a diff should increase risk. Cyclomatic complexity takes into account not just the size of the file but also the branching structure. Files implementing complex logic may be more risky.

The maximum of the test coverage represents the fraction of lines covered during test execution. Intuitively, more testing should capture faults before they become SEVs, the observational nature of this study suggests that, quite likely, files with (or likely to have) SEVs get more extensive test suites (and correspondingly larger test plans).

Table 2. Features for `DiffModel` Include Advanced Features at the Diff Level That Are Being Considered for Use in Production

Category	Measure	Description	Calculation
Defects	$had\_sev(d)$	True if any of the files in the diff have been modified by a SEV-causing diff in the past	$\max_{\{f:d_f\}} had\_sev(f, d_t)$
Interdependencies	$call\_centrality(f, t)$	The maximum Katz call graph centrality of the files in the diff	$\max_{\{f:d_f\}} cent(f, d_t)$
	num_of_files	The number of files modified by the diff	$\lvert\{f : d_f\}\rvert$
	$diff\_only\_creates\_files$	True if all of the files modified by the diff are created by the diff	$\lvert\{f : d_f\}\rvert$
Coordination	$num\_reviewers(d)$	The number of reviewers for the diff	$\lvert\{r : d_r\}\rvert$
	$author\_centrality(d)$	Katz centrality of the diff author based on co-change, authorship, and call graph links	$cent(a, d_t)$
Expertise	$sum\_author\_churn\_files(d)$	Author expertise measured by the number of diffs to the files modified by the diff in the past	$\lvert cup_{\{f:d_f\}}\{D : D_t < t \wedge D_f \wedge D_a\}\rvert$
File complexity	$churn(d)$	Sum of the lines of code in modified files	$\sum_{\{f:d_f\}} LOC(f, t)$
	$complexity$	Maximum cyclomatic complexity over modified files	$\max_{\{f:d_f\}} CC(f, t)$
Testing	$test\_plan\_length(d)$	Length of the test plan for diff d	
	$max\_test\_coverage(d)$	Maximum percent of lines covered in unit tests	$\max_{\{f:d_f\}} test\_cvg(f, d_t)$
diff duration	$total\_diff\_time(d)$	The elapsed time from diff d creation until landing d_t	
Codebase	$all\_recent\_files(d)$	Whether all modified files $\{f : d_f\}$ were created recently (within 3 months)	$\min_{\{f,D:d_f \wedge D_f\}} D_t \geq d_t$
	$is\_core(d)$	True if any of the modified files $\{f : d_f\}$ belong to an org that provides code with critical online services to most other orgs	

The measures are computed for each diff separately as they were at the time diff landed. The notation is the same as in Table 1. In addition, $cent(f, t)$: centrality of file f at time t, $CC(f, t)$, $LOC(f, t)$: cyclomatic complexity and lines of code of file f at time t, d and D, are diffs, a: author, r: reviewer, d_a, d_r: indicators that a authored and r reviewed d.

Diffs that take longer may indicate they are more difficult, hence more risky. We also add predictors for recently created files (assuming lower risk) and the type of functionality involved. Some of the core and business-critical services are deployed in many products: we assume that critical codebase could be identified by code ownership.

For files and diffs our response is a Boolean: whether or not a file will be modified by a SEV-causing diff in the former case and whether or not a diff will trigger a SEV in the second case. Logistic regression models are commonly used for such data. In addition to being simple to fit and extremely simple to predict and explain (via an explicit formula), logistic regression produces predicted probability of an SEV that could be easily have a threshold (see Section 3.5) to achieve various levels of gating without refitting the model. Furthermore, logistic regression is commonly used in risk prediction models. We represent the models using R syntax. All continuous variables are log transformed using the $log1p$ function because they have a skewed distribution. We add one to ensure that there are no zero values in the log transformation. We do not show $log1p$ to make the model easier to read. In practice, some of the data may be missing or incorrect (e.g., positive

quantities have a negative value). We correct such instances and impute missing values to avoid errors in the production environment.

The `FileModel` has the following form in R syntax:

$$\text{FileRisk} \sim \text{had\_sev} + \text{past\_diffs} + \text{is\_core} + \text{test\_coverage} + \text{loc}.$$

The `DiffModel` has the following form in R syntax:

$$\begin{aligned}\text{DiffRisk} \sim\ & \text{had\_sev} + \text{call\_centrality} + \text{num\_of\_files} + \text{diff\_only\_creates\_files} + \text{num\_reviewers} \\ & + \text{author\_centrality} + \text{sum\_author\_churn\_files} + \text{churn} + \text{complexity} \\ & + \text{test\_plan\_length} + \text{max\_test\_coverage} + \text{total\_diff\_time} + \text{all\_recent\_files} + \text{is\_core}.\end{aligned}$$

3.5 Thresholds and Outcome Measures

Unlike `PathModel`, `FileModel` allows for adjusting the number of `DiffsGated` for a particular type of the code freeze. The actual decisions on the exact timing and the threshold to be used in each code freeze concern specific business or engineering needs and are beyond the scope of this study. The documents planning the code freezes were worked out based on discussions among quality engineers, software engineers, and engineering management. The key part of the discussion was the quantitative evaluation of the tradeoffs between the fraction of `DiffsGated` and the percentage of SEVs that are captured by the model, i.e., `CapturedSEVs`. The thresholds varied because the objectives of each code freeze were different.

As noted above, the dual goals of the code freeze are to reduce the number of SEVs, i.e., severe faults and minimize the disruption to the development team. Once the threshold is selected, the model flags (gates) diffs exceeding this threshold risk. The fraction of all diffs that are gated and the fraction of `DiffsGated` that actually lead to SEVs, i.e., `CapturedSEVs` are, thus, the two primary outcomes used to compare models and to decide upon the threshold to be used in each code freeze. To calculate actual probabilities to be used as thresholds, a model was applied on 3–6 months of data producing the probability for each diff. The probability value p_c for which a desired (e.g., 20%) fraction of diffs in this set would have greater risk (e.g., $p_c = percentile_{20}p$) was then used to configure the gating tool.

When the model is released in production, engineers are allowed to select a rationale for why a diff that was marked as risky should still be landed. We report the number of diffs that have a rationale and are landed.

4 Results

4.1 RQ 1. Current Practice

Our first research question was to better understand existing software development practices related to the code freeze, or:

How does Meta Currently Conduct Code Freezes?

4.2 Types of Code Freezes

To understand how code freezes are conducted at Meta we follow methodology described in Section 3.1. We found that each code freeze strategy aimed to reach a meaningful balance between quality and productivity. The first approach restricts diffs during certain periods, such as before the release, when technical support is hard to come by, or when usage is high. The second approach restricts diffs to certain parts of the codebase. This is motivated by the fact that modifications to a very small fraction of the codebase are responsible for almost all SEVs. The last approach restricts landing of certain diffs that are deemed to be risky.

"Red Zone" Freezes. Since Meta conducts trunk-only development and there is only one "branch," so code freezes can only be short-lived. The initial approach avoided all diffs during certain high risk, high usage "red zone" periods.

Tooling changes were then introduced to automatically prevent landing of all diffs during "red zone" unless an exception was obtained. This undifferentiated approach was objected to by the engineers who could not land diffs that would not cause SEVs for end-users, e.g., diffs to internal tools.

Timing is arguably the most essential part of the code freeze as an infinite freeze would inhibit the ability to evolve the software permanently. Traditional code freezes affect only maintenance branches and, therefore, do not disrupt trunk development. At Meta, as in many other software companies that offer online services, releases are very frequent, and, because of that, branches for maintenance mode (where code freezes are typically applied), are not used. Instead, the code freeze is applied for two primary reasons. First, it is applied during special events when system SEVs would be especially detrimental to user experience and Meta's reputation. These events typically correspond with holidays or other special events of that kind. Second, freezes are applied when SEV would cause unwanted disruptions to the engineering team, such as during weekends or holidays. Weekend freezes also tend to cause much less disruption to the development process as diff landing activity is much lower. Generally, a much higher percentage of diffs are gated for the freezes of the first type than for the freezes of the second type. In addition to the two primary freeze types, other freezes are also conducted. For example, there may be specific occasions to support a one-off event. In these cases, only functionality that is relevant to that event is frozen, thus limiting the overall disruption caused by the freeze.

The next three approaches are discussed in more detail below and involve expert-based selection of risky files (`ExpertPaths`), model-based selection of risky files (`FileModel`), and, finally, model-based selection of risky diffs (`DiffModel`). The three approaches go beyond simply timing code freeze and into specific strategies to enforce the code freeze by where development occurs (location) and even by individual tasks (diffs).

Location-Based Code Freeze. The first, `ExpertPaths` approach, relies on experts being able to correctly identify a set of risky folders that, diffs modifying any file within these would tend to be more risky. Furthermore, not all files within a folder have the same risk. With a very large codebase it would be too cumbersome (and likely ineffective) for experts to assess the risk of each individual file. Finally, with `ExpertPaths` approach it is not possible to easily control what fraction of SEV-causing-diffs end up being flagged versus what fraction of all diffs are flagged. These shortcomings can be addressed by employing some of the well-studied defect prediction models described in Section 2.2. Historic data on past SEVs are particularly well-suited for risk models that, unlike experts, can be used to identify the most risky files, not just the most risky paths. Using a suitable risk model it is also possible to easily adjust risk thresholds used to gate the diffs. In summary, the resulting `FileModel` represents a more elaborate and flexible version of location-based code freeze.

Diff-Based Code Freeze. The time- and location-based code freeze can reduce code freeze inefficiencies substantially, but further improvements may be possible if we also take into account the properties of individual diffs. As described in Section 2.2, the so-called "just-in-time bug prediction" literature models risk of individual code diffs. The advantage of a diff-based model is that it can take into account the properties of individual diffs. For example, the risk of a diff that modifies only comments even in the most risky file is low, but location-based models would not be able to predict that. On the other hand, a diff modifying numerous, even if not risky, files, might pose higher risk.

As the approach was deployed, we first implemented a *file-risk* approach based on modeling the probability that a file will be involved in a diff that causes an SEV. If a diff modifies a file that exceeds threshold risk, the diff is gated. The gating percentage can be determined by observing the quantile of risks for past diffs and used as a threshold value for the code freeze.

Table 3. The `FileModel` Logistic Regression Predicting If a File Will be Part of an SEV-Causing Diff

Measure	Direction	Deviance	z value	Pr(>|z|)
(Intercept)	−	NA	−277	0.00
had_sev	+	60,000	238	0.00
past_diffs	+	51	8	0.00
is_core	+	15,000	111	0.00
test_coverage	−	650	−25	0.00
loc	+	3,200	57	0.00

The model explains 31% of the deviance. The "Deviance" shows likelihood ratio of including the predictor in the model.

In summary, the traditional code freeze prevents changes to the codebase for a short period of time before a release. At Meta we find that code freezes are done in several different ways.

> The objective of code freezes was to minimize disruption of development while ensuring the best user experience. Historically, they were implemented as restrictions on code integration based on the perceived risk and when the files were modified.

4.2.1 Code Freezes by `ExpertPaths`. One way to determine the risk of a diff is to ask a domain expert. This is, unfortunately, impractical due to the need to assess large numbers of diffs as they land. An alternative is to ask experts to assess the risk of changing each file. While the number of files is lower than the number of diffs, it is still formidable. As described in Section 4.2, a compromise approach was used at Meta, where the experts in each domain were asked to specify a range of paths (typically using regular expressions) where, according to their perception, the diffs are most likely to lead to a SEV, i.e., the location-based dimension of code freezes.

In practice each manager from distinct areas of the codebase designates a developer who is an area expert to specify *the code paths* that they deem to be the most important. Any diff to such files would be either blocked (the so-called "red-zone") or gated ("yellow zone"). If their diff is gated, and the engineer still wants to land the diff, they must provide a rationale and potentially obtain an approval from the area expert. To assess the effectiveness of this approach and to compare it to model-based approaches we conducted a historical analysis to understand how many diffs touch these file paths and how many of these gated diffs would have led to SEVs. We found the following:

> In a historical analysis, when code freeze is applied by restricting diffs to the file paths selected by experts (`ExpertPaths` approach), we find that 27.3% of diffs would have been gated and 38% of SEVs would have been captured.

4.3 RQ 2. File Risk Model

How Well Does the File Risk Model Gate Diffs with SEVs?

Instead of selecting files based on expert opinion, we use past diff history to identify if certain files are more likely to be modified by SEV-causing diffs. The features for the file risk model have been described in Sections 2.2.2 and 3.4. The results of a logistic regression predicting if a file will be modified by a SEV-causing diff are shown in Table 3.

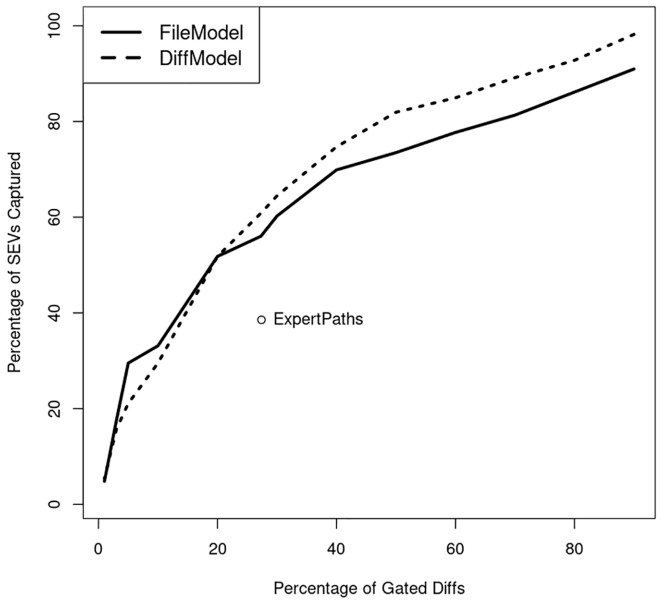

Fig. 2. Percentage of diffs gated vs. the number of SEVs that were captured by `FileModel` and `DiffModel`.

The correlations among predictors were especially high at the file level, hence we needed only a few predictors to get a good model with 31% of the deviance explained. Table 3 shows that the two most important predictors are whether the file was previously modified by a SEV-causing diff and if the file was a part of the core functionality. Cyclomatic complexity had 0.96 Spearman correlation with *loc*, hence it is not included in the model. The remaining predictors have an intuitive explanation: risk increases for larger files with more past diffs and decreases with test coverage. In fact, SEV-causing diffs modify relatively few files, similar to other industry reports [28].

> Had we used the `FileModel` to gate the same number of diffs as the `ExpertPaths` approach, the `FileModel` would capture 56% of SEVs or more than double the odds ratio of capturing SEVs than `ExpertPaths` approach. The most important features are the number of prior SEVs, whether the code was part of the core, and the number of LOCs in the file.

4.4 RQ 3. Diff Risk Model

How Well Does the Diff Risk Model Gate Differ with SEVs?

Using the `DiffModel` developed in Section 3.2, we conducted a historical analysis, given a threshold for the number of `DiffsGated`, how many SEVs would have been captured by the model, `CapturedSEVs`. Figure 2 plots the effectiveness of the model. The dot represents the path-expert rules which gates 27.3% of diffs and captures 38.5% of SEVs compared to 56.0% and 60.8% for `FileModel` and `DiffModel`, respectively. The `DiffModel` consistently outperforms the `FileModel` after 20% of diffs are gated. We see that once `DiffsDelayed` is above 20%, the `DiffModel` is always more effective than the `FileModel`. We also see that it far outperforms the `ExpertPaths`. Because most of the SEVs are associated with only a tiny fraction of files, `FileModel` is surprisingly good and performs comparably or better than the `DiffModel` for up to 20% of `DiffsGated`. Only once the gating level increases, `DiffModel` shows the advantage of taking properties of the code diff into account.

Table 4. The DiffModel Predicting the Chances That a Diff Will Cause an SEV

Category	Measure	Direction	Deviance	z value	Pr(>|z|)
	Intercept	−	NA	−19.01	0.00
Defects	had_sev	+	488	17.14	0.00
Interdependencies	call_centrality	−	4	−0.37	0.71
	num_of_files	+	24	5.17	0.00
	diff_only_creates_files	−	4	−1.80	0.07
Coordination	num_reviewers	+	18	2.87	0.00
	author_centrality	−	5	−1.78	0.08
Expertise	sum_author_churn_files	−	4	−0.99	0.32
File complexity	churn	+	80	8.52	0.00
	complexity	+	4	0.52	0.61
Testing	test_plan_length	−	3	−6.06	0.00
	max_test_coverage	+	29	4.80	0.00
diff duration	total_diff_time	+	10	3.06	0.00
Codebase	all_recent_files	−	17	−4.46	0.00
	is_core	+	91	9.39	0.00

The "Deviance" column shows likelihood ratio of including the predictor in the model.

Table 4 shows the most important features. We see that the two most important predictor (similar to the file-level model) are an indicator of the diff modifying at least one file previously involved in a SEV (has_prior_sev) and an indicator of whether the diff modifies core functionality (is_core) closely followed by the total number of lines changed in the diff, i.e., churn. All three increase the risk as expected. The secondary by importance group includes new code represented by entirely new or recent files (both decreasing risk as expected), the number of files modified by the diff, maximum file complexity, and total time needed to implement the diff (with all three increasing the risk). Author centrality and prior diffs on the modified files (both proxies of expertise) and the test plan length all decrease the risk as expected. The two apparent surprises include maximum test coverage and file call centrality that, unexpectedly, decrease the risk.

We cannot stress sufficiently that the risk models in software engineering are predominantly based on the observational data, since controlled experiments are rarely conducted in software quality domain where a serious SEV may be prohibitively expensive. The key problem with observational data is that of latent variables that cannot be directly observed but that both cause SEVs and affect other metrics, such as code coverage. Experienced engineers typically have an intuition about the risk of software components or diffs and try to reduce the risk via inspections, testing, and other quality improvement approaches. As a result, problematic areas of the code and diffs may have much higher test coverage and number of reviewers, but still be more likely to have an SEV despite that extra scrutiny: after all, neither reviews nor testing can catch all the problems. The model, without the measure of this developer intuition, simply associates more quality assurance with high SEV risk. Specialized causal analysis techniques may help identify the presence of such unobserved predictors, e.g., [2], but they are quite cumbersome to deploy in practice. In practice, it is extremely difficult to explain the implications of observational data, so we cannot deploy the model that appears to tell the developers to reduce reviews or testing to reduce the risk of diffs. While the full description of the approach we took to solve the actionability problem is beyond the scope of this article, the key point is that given high correlations among predictors we can pick different subsets of predictors without significantly affecting model prediction performance. The

model that implies meaningful quality improvement action and has good performance is eventually deployed in practice.

Discussing practical matters of using the model is important to note that displaying the risk is most useful in advance to diff landing so that the engineer can anticipate if they will be able to land it once the diff is complete. This puts additional constraints on the model. For example, the total_diff_time keeps changing, increasing the risk during the lifetime of a diff, hence it cannot be used in the model. Furthermore, as more reviewers join in, risk keeps (counter-intuitively) increasing. To address these problems, we can estimate the expected value for each predictor that changes during the diff lifetime and use it (instead of the actual value) to minimize distraction caused by constantly changing risk scores.

> Had we used the `DiffModel` to gate the same number of diffs as the `ExpertPaths` approach, the `DiffModel` would capture 60.8% of SEVs, or odds ratio greater than 2.5 than `ExpertPaths` approach and greater than 1.2 than the `FileModel`. The most important features are the prior SEVs, number of reviewers, and number of files in the diff.

4.5 RQ4. Production

How Well Does the Risk Model Work in Practice?

We present the results from our July deployment. While the risk model had been used to reduce the risk of diffs on the weekends, the July deployment window was the first major test of the model in production. At this time, the `FileModel` was being used to determine which diffs need to be gated. We report the results for the July 2023 window in this article. Over this 3-day period, 74% of diffs landed without any interruptions, 26% of risky diffs moved into the gated flow, 8.8% landed with pre-approved exception reasons, 1.0% were escalated for approval, and 0.28% had their escalation approved. After landing exceptions, this means that only 17% of diffs were delayed until after the July deployment window, i.e., a maximum "freeze" period of 3 days. Despite exceeding traffic records at Meta, we did not experience any SEVs.

The largest production use of the risk model was during the winter holiday period that begins in mid-November and ends in January with multiple intervals of code freeze. The `DiffModel` is now in use and only 16% of diffs were gated, 5.7% were allowed to land after the developer provided landing rationale. When we compare to the prior year's winter code freeze, we see a 42% increase in the number of landed diffs. Instead of seeing a corresponding increase in SEVs, we see a 52% decrease in the number of SEVs compared to last year. This reduction in SEVs also reduced the operating cost related to dealing with SEVs by 47%, and meant fewer disruptions for engineers during the holiday period. Risk modeling has been enormously successful and its use continues to expand at Meta.

> The first major test of the risk model was the July 2023 deployment window. Despite record traffic levels, the `FileModel` ensured that no major SEVs were introduced. In total, 26% of diffs were gated, and 8.8% were allowed to land after the developer provided landing rationale. The `DiffModel`'s first major production usage was during the winter code freeze. Compared to the prior year, despite 42% more diffs landed, there was a reduction in SEVs of 52%. The risk model has been enormously effective at allowing low risk diffs to land while gating high risk diffs and reducing SEVs.

5 Discussion

We start from a historic perspective and continue with implications for the current development process.

5.1 History of Code Freezes

Software engineering literature does not explicitly and precisely define code freeze. For example, "only obvious bug-fixes to existing functionality will be applied" [50], "during code-freeze the release engineer is vested with the authority to reject all changes other than bugfixes" [14]. We found a lone explicit definition of code freeze: "Feature freeze is a phase during release stabilization when enhancements are no longer added to the release and only critical defects can be fixed in a separate release branch, while new release development can continue in the trunk. 2) Code freeze is a phase during release stabilization when no diffs are done to the software and a release artifact is only tested" [20].

If we look at software development practice historically, the most common approach to focus software quality improvement was by designating periods of time during which extensive testing is conducted and no changes or only the critical fixes were allowed (even before [8] version control systems allowed separating codebase into development and maintenance branches). The reductions in or elimination of fixes and enhancements were used to stabilize the codebase before software releases were delivered to customers [38]. A common term used for such activities was "code freeze." In many domains, software releases used to be rare events with up to a multiyear gap between major releases. The released binaries, whether for shrink-wrap, server, or embedded software, were then delivered to the customers via installation media (tapes/CDs/USBs) or via physical devices running that embedded software. High quality was paramount as the updates were difficult to deliver, or extremely expensive (e.g., shipping a large-screen TV back to the vendor for repairs), or not possible. During code freezes, landing of all but the most important bug-fixing diffs would be prohibited resulting in a stable code base that would then be extensively tested, including via the long-running stability, availability, or stress tests.

The evolution of computer performance, version control systems, network deployment, alpha and beta testing, and other advances made it possible not only to maintain different versions of software (trunk, and maintenance branches), but also to release after each change (continuous integration) even for the largest products. Linux was one of the first projects to "release early and often" [39] providing developers with a 2-week window to request change to be merged and then a stabilization period where release candidates were created for testing by the community. The "code freeze" or stabilization period lasted on average 62 days [38]. Many projects use multiple stabilization branches with varying degrees of code freeze. For example, Chrome had development, beta, and stable branches. At 6-week intervals, the code on development would go to beta, beta would go to stable, and stable would be released [38]. This staggered branch strategy is very common across the industry [1].

Notably, all mentions of code freeze appear to be related to individual branches of the software, with maintenance branches getting only critical updates, and alpha/beta branches getting only bug fixes, and so on, with the development branches allowing all changes. Meta primarily does trunk-only development with developers integrating their own diffs into trunk. With continuous integration, diffs are immediately integrated into a master branch and tested, thus becoming available for delivery to the end user. In the case of Meta, the scale of diffs was so large, that instead of releasing each diff, the diffs were batched and gradually rolled out three times per day [41]. The move towards continuous integration and delivery has been described at Meta [41] and Ericsson [20] among others.

As Rossi [41] noted that as a project and company scale, it becomes impossible for a team of release engineers to determine which features are ready for production. The next evolution of Meta's practices pushed this responsibility back to the engineers to decide which features were

ready for production. An extensive infrastructure allowed for diffs that saw problems in production to be toggled off reducing the impact of bad code diffs [37].

5.2 Code-Freeze Considerations

As we investigated plans and retrospectives of past code freeze efforts at Meta and participated in creating and evaluating new approaches, several additional considerations that were not initially obvious stood out.

First, the primary aims of the code freeze varied substantially among organizations and among different occasions when code freezes were implemented. This points to the need to tailor the code freeze approach to a particular context. While, for example, high usage events require focus on preventing landing of any risky diffs, the freezes done during weekends are mostly to avoid disturbing engineers with emergency requests to handle the SEV. SEVs for internal tools primarily decrease the productivity of developers using these tools, but may not have a direct impact on customer satisfaction.

Furthermore, the main purpose of weekend freezes is to delay the SEV until weekdays, but the reliability-focused end-of-the-year freezes may be also used to make the diff less risky and engineers may need to be informed exactly why their diff was gated and what remedial actions are recommended. Besides these two extremes (long, usage-based and brief, engineer availability-based) other types of freezes may be employed in cases of introduction of major new functionality (for customer-facing products) or expected high development activity periods (for internal development tools).

While each type of code freeze has different objectives and requirements, generally speaking, brief code freezes are mostly focused on delaying SEVs and have lower levels of gating than the long-term periods of high system usage code freezes. The latter add additional requirements to explain to engineers the risk causes and potential remedial actions.

It is important to note that in all cases, the critical question is how to maximize the fraction of SEVs causing diffs for a given level of gating as only a tiny percentage of diffs cause SEVs.

The key lesson is that the code freeze approaches that have previously been used for maintenance branches, need significant modifications for trunk-only development. Trunk, unlike the maintenance branch, directly impacts most of software development. The first approach to avoid disruption was to use heavily constrained time periods where the diffs were prevented from landing. Even brief code freeze periods were causing significant overhead and motivated differentiated gating based on the location and task.

5.3 Implications for Developers

As our findings show, the return (after over 50 years) to trunk-only development brought the need to modernize code-freeze, which was relegated to help manage maintenance branches only. Furthermore, removal of release managers due to the sheer scale of changes that need to be incorporated almost in real time, required the development of other coping strategies to ensure uninterrupted operation.

We found that the traditional code freeze dramatically slows development during periods when it is applied. To prevent such slowdown code needs to be "un-frozen." As we describe above, generally three strategies are used at Meta to accomplish that. First, the timing of the freeze is based on the particular goals to be achieved, for example, to manage developer interruptions during weekends and to ensure uninterrupted operation during hi-usage periods. Outside these periods, code is unfrozen, removing impediments to developer work. The second approach is to identify parts of the code base that are more critical (i.e., used for most critical services). This unfreezes development of the remaining codebase. All changes modifying critical codebase are prevented, however. The third approach, instead of focusing on the codebase, considers the risk of individual changes, allowing remaining changes to proceed.

The effectiveness of the last two approaches highly depends on the ability to accurately pinpoint risky files or diffs, correspondingly. If, for example, the risk is spread uniformly over codebase and tasks, such an approach would not bring any value. Fortunately, risk of outages is extremely unevenly distributed over tasks and codebase and statistical models using historic data can easily "learn" these variations in risk and predict them for new changes done during the critical periods. Furthermore, data needed to construct such models is typically already available in most software companies. The data still needs to be cleaned, made available in near-real-time and the models periodically updated for maximum performance.

In short, trunk-only development with relatively modest constraints for the most risky changes is possible without extensive work-stoppages required by traditional code freeze. It is an open RQ whether (and in what cases) presently complex arrangements with concurrent releases that suffer from code synchronization problems could be solved using head-only development with more sophisticated code-freezes.

More generally, the bug prediction research, whether at the level of files or at the level of changes ("just -in-time" bug prediction), might find a new (and, arguably, more practical) outlet in developing increasingly sophisticated code freezes tailored to various scenarios where the tradeoff between quality and productivity may vary over parts of the codebase or over time.

One may ask why do trunk-only development in the first place? A look at the literature on synchronizing branches via back- and forward-porting and difficulties faced by release managers [42] suggests that trunk only development has major benefits of distributed decision making that does not require synchronization (and overheads associated with it) if the quality goals could be achieved.

5.4 Experiences in Deployment

To illustrate practical limitations, we intentionally present initial prototype models that could be rapidly evaluated and iteratively refined in production-based evolving needs and requirements. In fact, the process was established to evolve the model based on various stakeholder requirements and the improved ability to efficiently calculate more sophisticated features. The key part of the process was evaluating the performance of competing models on the validation dataset. Measures and models that did not improve the performance were put aside and models or measures that significantly improved model's performance were prioritized for implementation and deployment. Not only model performance was considered but also feedback from engineers. One of the first requests was to add an explanation for why the risk was high. Almost as high priority was a request not to gate diffs that do not modify code or configuration, i.e., diffs modifying only comments. Importantly, the added capabilities made possible (and lead to) entirely new functionality. For example, the ability to threshold `FileModel` resulted in the need to tailor `DiffsGated` for each freeze. Correlations among measures showed that some of the more difficult-to-compute measures were highly correlated to the simpler-to-compute measures that, then, could often be used to replace the more complex measure without decreasing the model's predictive power. A similar exercise was used to handle counter-intuitive coefficients resulting from the observational nature of the underlying data, such as the risk score increases with more test coverage. Alternative predictors were sought that convey actionable messages, yet keep predictive performance.

6 Threats to Validity

Generalizability. Drawing general conclusions from empirical studies in software engineering is difficult because any process depends on a potentially large number of relevant context variables. The analyses in the present article were performed at Meta, and it is possible that results might not hold true elsewhere. However, our study does cover a very wide swath of software engineering.

The software system covers millions of lines of code and 10s of thousands of developers who are both collocated and working at multiple locations across the world. We also cover a wide range of domains from user facing social network products and virtual and augmented reality projects to software engineering infrastructure, such as calendar, task, and release engineering tooling.

While the desire to prevent SEVs is common among companies, the exact ways in which it is done varies. Such variation may be partly due to unique circumstances surrounding the way software is developed and services are delivered in each company, but they may partly reflect different evolution of quality improvement efforts and would apply for other companies, even with different software development or service delivery models. It is this latter perspective based on experience of working in a number of substantially different large companies that bids us to share the experiences of quality improvement efforts more broadly.

Construct Validity. We have used a straightforward set of measures that have been widely used in the Mining Software Repositories bug prediction literature [2, 55], e.g., prior defects and churn. We also use more complex collaboration measures relying on software supply chains and centrality [4, 43, 48, 51, 53, 56].

As is commonly necessary for statistical models in software engineering [5, 22–25], we log-transformed variables with highly skewed distributions and inspected correlations among predictors in the logistic regression as well as inspecting residuals to check for nonlinear or non-monotone relationships and doing standard regression diagnostics. We also investigated interactions among predictors, but none satisfied our requirements (improve prediction, be easy to calculate, and be easily interpretable). We found most of predictors to be highly correlated, hence having only relatively few predictors provided best performance on the testing dataset. Furthermore, we found that we could pick and choose predictors from within the highly correlated clusters without negatively affecting model performance. For production, we thus focused on other aspects, like ease of calculation at the time of prediction and interpretability instead. We experimented with other techniques, including deep learning, to embed the diffs, but were not able to improve upon the simple regression models.

Internal Validity. The numerical results for RQs 1–3, rely on historical data, and simulated how the model would have performed had it been used by developers, i.e., how many `CapturedSEVs` given the number of `DiffsGated`. However, it is possible that the actual performance would have varied. To mitigate this threat, we also reported on how well the `FileModel` worked in production in Section 4.5. The real-world performance of the model suggests that it is very effective at capturing SEVs. The observational nature of the data is not an issue for the model's predictive performance, but it makes it challenging to explain counter-intuitive coefficients to engineers, e.g., why more reviewers and test coverage increase the risk? A standard statistical explanation of latent variable, i.e., an unobserved property of a diff that demands more reviewers and more testing also makes it more risky, is very difficult to communicate. We, therefore, focused on selecting or designing predictors that could be easily interpreted.

7 Concluding Remarks

Historically, code freeze has been used to stabilize a release branch allowing only critical fixes. At Meta, a single trunk-based monorepo is used. Code freeze in this context requires freezing different file paths or high-risk parts of the system during times of high-use or when engineering staff are less available, e.g., the weekend. We worked with engineers, engineering managers, and quality engineers to develop new approaches to code freeze that will improve engineering productivity via un-freeze: by only gating changes that are likely to lead to SEVs. Specifically, we found that when code freeze is applied by restricting diffs to the file paths selected by experts (`ExpertPaths`

approach) 27.3% of diffs would have been gated and 38% of SEVs would have been captured. Using the `FileModel` to gate the same number of diffs as the `ExpertPaths` approach, the `FileModel` would capture 56% of SEVs: an odds ratio >2. Had we used the `DiffModel` to gate the same number of diffs as the `ExpertPaths` approach, the `DiffModel` would capture 60.8% of SEVs, which has an odds ratio >2.5 over `ExpertPaths` approach and >1.2 over `FileModel`. The first major test of the risk model was the July 2023 deployment window. Despite record traffic levels, the `FileModel` ensured that no major SEVs were introduced. In total, 26% of diffs were gated, 8.8% were allowed to land after the developer provided landing rationale. The `DiffModel`'s first major production usage was during the winter code freeze. Compared to the prior year, despite 42% more diffs landed, there was a reduction in SEVs of 52%. The risk model has been enormously effective at allowing low risk diffs to land while gating high risk diffs and reducing SEVs.

First, we discovered and evolved a previously unreported family of code freeze strategies for trunk-only development used in a very large software organization. Second, we demonstrate and evaluate how traditional defect prediction and just-in-time defect prediction models can be used to reduce the overhead of code freezes thus extending the use-case scenarios for such techniques. Third, we proposed and evaluated how several software supply chain properties can be used to improve code freeze effectiveness by un-freezing all but the most risky changes. Fourth, we discussed aims and rationale for previously not reported code un-freeze strategies. Fifth, we described a very large-scale deployment code un-freeze strategies supported by risk models in Meta.

We expect our work to result in further research and broader practical application of innovative code un-freeze strategies and to provide a more practical rationale and more meaningful evaluation criteria for research on software risk.

References

[1] Bram Adams and Shane McIntosh. 2016. Modern release engineering in a nutshell – Why researchers should care. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 5, 78–90. DOI: https://doi.org/10.1109/SANER.2016.108

[2] Peter C. Rigby, Andrey Krutauz, Tapajit Dey, and Audris Mockus. 2020. Do code review measures explain the incidence of post-release defects? *Empirical Software Engineering* 25, 5 (2020), 3323–3356. Retrieved from https://www.mockute.com/papers/reviews_replication_emse.pdf

[3] Fran Baum, Colin MacDougall, and Danielle Smith. 2006. Participatory action research. *Journal of Epidemiology and Community Health* 60, 10 (2006), 854–857.

[4] Christian Bird, Nachiappan Nagappan, Harald Gall, Brendan Murphy, and Premkumar Devanbu. 2009. Putting it all together: Using socio-technical networks to predict failures. In *2009 20th International Symposium on Software Reliability Engineering. IEEE*, 109–119.

[5] B. W. Boehm. 1981. *Software Engineering Economics*. Prentice-Hall.

[6] Marcelo Cataldo, Audris Mockus, Jeffrey A. Roberts, and James D. Herbsleb. 2009. Software dependencies, the structure of work dependencies and their impact on failures. *IEEE Transactions on Software Engineering* 35 (2009), 864–878. Retrieved from https://www.mockute.com/papers/multicompany.pdf

[7] Daniel Alencar Da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E. Hassan. 2016. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering* 43, 7 (2016), 641–657.

[8] Siddharta R. Dalal and Collin L. Mallows. 1988. When should one stop testing software? *Journal of the American Statistical Association* 83, 403 (1988), 872–879.

[9] Carsten F. Dormann, Jane Elith, Sven Bacher, Carsten Buchmann, Gudrun Carl, Gabriel Carré, Jaime R. García Marquéz, Bernd Gruber, Bruno Lafourcade, Pedro J. Leitão, et al. 2013. Collinearity: A review of methods to deal with it and a simulation study evaluating their performance. *Ecography* 36, 1 (2013), 27–46.

[10] Yuanrui Fan, Xin Xia, Daniel Alencar Da Costa, David Lo, Ahmed E. Hassan, and Shanping Li. 2019. The impact of mislabeled changes by szz on just-in-time defect prediction. *IEEE Transactions on Software Engineering* 47, 8 (2019), 1559–1586.

[11] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. 2000. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering* 26, 7 (2000), 653–661.

[12] R. Hackbarth, A. Mockus, J. Palframan, and R. Sethi. 2016. Customer quality improvement of software systems. *IEEE Software* 33, 4 (2016), 40–45. Retrieved from https://mockus.org/papers/cqm2.pdf

[13] James Herbsleb and Audris Mockus. 2003. Formulation and preliminary test of an empirical theory of coordination in software engineering. In *2003 International Conference on Foundations of Software Engineering*. ACM Press, Helsinki, Finland, 138–137. Retrieved from http://dl.acm.org/authorize?787510

[14] Niels Jørgensen. 2001. Putting it all in the trunk: Incremental software development in the FreeBSD open source project. *Information Systems Journal* 11, 4 (2001), 321–336.

[15] Yasutaka Kamei, Shinsuke Matsumoto, Akito Monden, Ken-ichi Matsumoto, Bram Adams, and Ahmed E. Hassan. 2010. Revisiting common bug prediction findings using effort-aware models. In *2010 IEEE International Conference on Software Maintenance. IEEE*, 1–10.

[16] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2013), 757–773. DOI: http://doi.ieeecomputersociety.org/10.1109/TSE.2012.70

[17] Leo Katz. 1953. A new status index derived from sociometric analysis. *Psychometrika* 18, 1 (1953), 39–43.

[18] Hossein Keshavarz and Meiyappan Nagappan. 2022. Apachejit: A large dataset for just-in-time defect prediction. In *19th International Conference on Mining Software Repositories*, 191–195.

[19] Barbara A. Kitchenham, Tore Dyba, and Magne Jorgensen. 2004. Evidence-based software engineering. In *26th International Conference on Software Engineering. IEEE*, 273–281.

[20] Eero Laukkanen, Maria Paasivaara, Juha Itkonen, Casper Lassenius, and Teemu Arvonen. 2017. Towards continuous delivery by reducing the feature freeze period: A case study. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP). IEEE*, 23–32.

[21] S. Linkman and H. Dieter Rombach. 1997. Experimentation as a vehicle for software technology transfer-a family of software reading techniques. *Information and Software Technology* 39, 11 (1997), 777–780.

[22] Audris Mockus. 2007. Software support tools and experimental work. In *Empirical Software Engineering Issues: Critical Assessments and Future Directions*. In V. R. Basili, D. Rombach, K. Schneider, B. Kitchenham, D. Pfahl, and R. W. Selby (Eds.). LNCS, Vol. 4336. Springer, 91–99.

[23] Audris Mockus. 2008. Missing data in software engineering. In *Guide to Advanced Empirical Software Engineering*. J. Singer (Ed.), Springer-Verlag, 185–200.

[24] Audris Mockus. 2010. Organizational volatility and its effects on software defects. In *18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 117–126. Retrieved from http://dl.acm.org/authorize?309271

[25] Audris Mockus. 2014. Engineering big data solutions. In *Future of Software Engineering Proceedings (FOSE)*, 85–99. Retrieved from https://dl.acm.org/authorize?N14216

[26] Audris Mockus. 2019. Insights from open source supply chains. In *2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE '19)*. Retrieved from https://dl.acm.org/doi/10.1145/3338906.3342813?cid=81100250207

[27] Audris Mockus. 2023. Securing large language model software supply chains. In *International Conference on Automated Software Engineering (ASE '23)*.

[28] Audris Mockus, Randy Hackbarth, and John Palframan. 2013. Risky files: An approach to focus quality improvement effort. In *9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 691–694. Retrieved from http://dl.acm.org/authorize?6845890

[29] Audris Mockus and David M. Weiss. 2000. Predicting risk of software changes. *Bell Labs Technical Journal* 5, 2 (April–June 2000), 169–180. Retrieved from https://mockus.org/papers/bltj13.pdf

[30] John D. Musa, Anthony Iannino, and Kazuhira Okumoto. 1987. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill Book Company.

[31] Nachiappan Nagappan and Thomas Ball. 2005. Use of relative code churn measures to predict system defect density. In *27th International Conference on Software Engineering*, 284–292.

[32] Edmilson Campos Neto, Uirá Kulesza, Daniel Alencar Da Costa. 2018. The impact of refactoring changes on the SZZ algorithm: An empirical study. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 380–390.

[33] Tetsuto Nishiyama, Kunihiko Ikeda, and Toru Niwa. 2000. Technology transfer macro-process: A practical guide for the effective introduction of technology. In *22nd International Conference on Software Engineering*, 577–586.

[34] Jalaj Pachouly, Swati Ahirrao, Ketan Kotecha, Ganeshsree Selvachandran, and Ajith Abraham. 2022. A systematic literature review on software defect prediction using artificial intelligence: Datasets, data validation methods, approaches, and tools. *Engineering Applications of Artificial Intelligence* 111 (2022), 104773.

[35] Shari Lawrence Pfleeger. 1999. Understanding and improving technology transfer in software engineering. *Journal of Systems and Software* 47, 2–3 (1999), 111–124.

[36] Sophia Quach, Maxime Lamothe, Yasutaka Kamei, and Weiyi Shang. 2021. An empirical study on the use of SZZ for identifying inducing changes of non-functional bugs. *Empirical Software Engineering* 26, 4 (2021), 71.
[37] Md Tajmilur Rahman, Louis-Philippe Querel, Peter C. Rigby, and Bram Adams. 2016. Feature toggles: Practitioner practices and a case study. In *13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, 201–211. DOI: https://doi.org/10.1145/2901739.2901745
[38] Md Tajmilur Rahman and Peter C. Rigby. 2015. Release stabilization on Linux and Chrome. *IEEE Software* 32, 2 (2015), 81–88. DOI: https://doi.org/10.1109/MS.2015.31
[39] Eric Raymond. 1999. The cathedral and the bazaar. *Knowledge, Technology & Policy* 12, 3 (1999), 23–49.
[40] Gema Rodríguez-Pérez, Andy Zaidman, Alexander Serebrenik, Gregorio Robles, and Jesús M. González-Barahona. 2018. What if a bug has a different origin? Making sense of bugs without an explicit bug introducing change. In *12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 1–4.
[41] Chuck Rossi. 2017. Rapid release at massive scale. Retrieved from https://engineering.fb.com/web/rapid-release-at-massive-scale
[42] Emad Shihab, Christian Bird, and Thomas Zimmermann. 2012. The effect of branching strategies on software quality. In *ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 301–310.
[43] Param Vir Singh. 2010. The small-world effect: The influence of macro-level properties of developer collaboration networks on open-source project success. *ACM Transactions on Software Engineering and Methodology* 20, 2 (2010), 1–27.
[44] Dag I. K. Sjøberg, Bente Anda, and Audris Mockus. 2012. Questioning software maintenance metrics: A comparative case study. In *ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '12)*. ACM, New York, NY, 107–110. DOI: https://doi.org/10.1145/2372251.2372269
[45] Dag I. K. Sjøberg, Jo Erskine Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanovic, N.-K. Liborg, and Anette C. Rekdal. 2005. A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering* 31, 9 (2005), 733–753.
[46] Gregory N. Stock and Mohan V. Tatikonda. 2000. A typology of project-level technology transfer processes. *Journal of Operations Management* 18, 6 (2000), 719–737.
[47] Ernest T. Stringer and Alfredo Ortiz Aragón. 2020. *Action Research*. Sage Publications.
[48] Ashish Sureka, Atul Goyal, and Ayushi Rastogi. 2011. Using social network analysis for mining collaboration data in a defect tracking system for risk and vulnerability analysis. In *4th India Software Engineering Conference*, 195–204.
[49] Chakkrit Tantithamthavorn, Ahmed E. Hassan, and Kenichi Matsumoto. 2018. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering* 46, 11 (2018), 1200–1219.
[50] Linus Torvalds. 1994. Linux code freeze. *Linux Journal*, 1es (1994), 4–es.
[51] Song Wang and Nachiappan Nagappan. 2021. Characterizing and understanding software developer networks in security development. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 534–545.
[52] Chadd Williams and Jaime Spacco. 2008. Szz revisited: Verifying when changes induce fixes. In *2008 Workshop on Defects in Large Software Systems*, 32–36.
[53] Marcelo Serrano Zanetti, Ingo Scholtes, Claudio Juan Tessone, and Frank Schweitzer. 2013. Categorizing bugs with social networks: A case study on four open source software communities. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 1032–1041.
[54] Marvin V. Zelkowitz, Dolores R. Wallace, and D. Binkley. 1998. Culture conflicts in software engineering technology transfer. In *NASA Goddard Software Engineering Workshop*. Citeseer, 52.
[55] Yunhua Zhao, Kostadin Damevski, and Hui Chen. 2023. A systematic survey of just-in-time software defect prediction. *Computing Surveys* 55, 10 (2023), 1–35.
[56] Thomas Zimmermann and Nachiappan Nagappan. 2008. Predicting defects using network analysis on dependency graphs. In *30th International Conference on Software Engineering*, 531–540.

Received 23 April 2024; revised 16 December 2024; accepted 17 December 2024