# An Industrial Study on the Risk of Software Changes

Emad Shihab and Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
Queen's University, Canada
{emads, ahmed}@cs.queensu.ca

Bram Adams
Lab on Maintenance, Construction and Intelligence of Software (MCIS)
École Polytechnique de Montréal, Canada
bram.adams@polymtl.ca

Zhen Ming Jiang
Research In Motion
Waterloo, ON, Canada

## ABSTRACT

Modelling and understanding bugs has been the focus of much of the Software Engineering research today. However, organizations are interested in more than just bugs. In particular, they are more concerned about managing risk, i.e., the likelihood that a code or design change will cause a negative impact on their products and processes, regardless of whether or not it introduces a bug. In this paper, we conduct a year-long study involving more than 450 developers of a large enterprise, spanning more than 60 teams, to better understand risky changes, i.e., *changes for which developers believe that additional attention is needed in the form of careful code or design reviewing and/or more testing*. Our findings show that different developers and different teams have their own criteria for determining risky changes. Using factors extracted from the changes and the history of the files modified by the changes, we are able to accurately identify risky changes with a recall of more than 67%, and a precision improvement of 87% (using developer specific models) and 37% (using team specific models), over a random model. We find that the number of lines and chunks of code added by the change, the bugginess of the files being changed, the number of bug reports linked to a change and the developer experience are the best indicators of change risk. In addition, we find that when a change has many related changes, the reliability of developers in marking risky changes is negatively affected. Our findings and models are being used today in practice to manage the risk of software projects.

## Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

## Keywords

Change Risk, Change Metrics, Code Metrics, Bug Inducing Changes

## 1. INTRODUCTION

Risk management plays a crucial part in successful project management. This is especially true for software projects. For example, a survey of 600 firms showed that 35% of them had at least one runaway project [6]. Another study showed that, industry-wide, only 16.2% of software projects are on time and on budget. Of the rest, 52.7% are delivered with reduced functionality and 31.1% are cancelled before completion. The main reason for this large amount of late projects is the lack of proper software risk management (i.e., activities used to manage the possibility of harm or loss) [6, 10].

Due to the importance of risk management in the success of software projects, researchers and industry have become more interested and active in the area of software risk management [13, 23]. One line of work that received a large amount of attention recently is software bug prediction, where code and/or historical metrics are used to predict where bugs might appear in the future (e.g., [26, 35]). In fact a recent literature review showed that in the past ten years more than 200 papers were published on defect prediction alone [17].

However, organizations are interested in effective management of risk in general, which covers more than just bugs. For example, a recent initiative on managing technical debt aims at studying how compromises that developers make today will affect their software in the future [30]. Risky changes could introduce bugs but they could also delay the release of projects, and/or negatively impact customer satisfaction. For example, changes that might have a widespread impact on the code (e.g., switching threading models) or on the user (e.g., making the software application autosave every 1 min instead of 30 seconds, for optimization reasons) are considered risky, regardless of whether or not they introduce bugs. The risk is caused by the uncertainty introduced by the changes.

*A risky change ideally requires additional attention through careful code/design review and possibly more testing*. This is why organizations are interested in identifying risky changes as soon as possible, so that there are enough time and resources available for risk mitigation. Although prior work investigated mitigation strategies (e.g., code reviews [32]) and bug-introducing changes [20], the risk of changes, which is at the core of the software creation process, has rarely been studied.

In this paper, we sought to better understand risk at a fine granularity, i.e., the individual software changes. We conducted a year-long study where developers from a large commercial company were asked to specify, at commit time, whether or not they consider their change to be risky. When assigning a change to be risky they are indicating that they wish additional attention to be considered for that change throughout the organization. The study involved more than 450 developers, spanning over 60 teams.

We use this large, unique data set to understand risky changes and find that:


*SIGSOFT'12/FSE-20,* November 11–16, 2012, Cary, North Carolina, USA.

- The interpretation of risk varies between different developers and teams. Therefore, prediction models that factor in the developer and/or team that made the change perform considerably better than general prediction models that aim to predict risk. This finding has implications on prior work on bug-introducing changes (e.g., [12, 20, 33]) and for future work related to risky and bug-introducing changes.
- We can build high accuracy models to automatically identify risky changes with a recall of 67% and a precision that is 87% (developer models) vs. 37% (team models) better than a random model. Our industrial partner found these models to be of great value in their risk management processes, especially for changes done by inexperienced developers.
- Each developer and team has their own factors that best model change risk. However, in general, the number of lines and chunks of code added by the change, the bugginess of the files being changed, the number of linked bug reports to a change and the developer experience are the best indicators of change risk.
- Risk and bugginess are related, yet different concepts. In general, developers are accurate when classifying buggy changes as being risky changes. However, changes that have many related changes are more likely to be incorrectly classified.

**Organization of the Paper.** Section 2 discusses the related work. Section 3 describes the data used in our study, while Section 4 introduces the case study setup. Section 5 presents our preliminary analysis on change risk assignment, followed by the results of our case study in Section 6. Section 7 discusses the differences between bug-introducing and risky changes. Section 8 reflects on the lessons learned and future work. Finally, Section 9 presents the limitations of our study and Section 10 concludes the paper.

## 2. RELATED WORK

The domains most closely related to this paper are software risk management, bug prediction and bug-introducing change prediction.

**Software Risk Management.** Prior work by Boehm [6] proposed principles and practices of software risk management. In this work, Boehm outlines the six phases (i.e., risk identification, analysis, prioritization, management, resolution and monitoring) of risk management. Dedolph [10] studied the role of software risk management practices at Lucent Technologies in order to understand why risk management is often neglected. He discusses examples of successful risk management. Freimut *et al.* [13] study the implementation of software risk management in an industrial setting. They proposed Riskit, a systematic risk management method, and showed that Riskit provides benefit for the risk management team with acceptable costs.

Our work is different from prior work on software risk management in that we are interested in the risk of one particular change at a time and not the risk of the entire project. Therefore, our work is done at a much finer granularity. Also, prior work on software risk management is concerned about all types of risk in the project, e.g., risk due to technicalities, risk due to personnel and/or risk due to work environment. Although our definition of risk is much wider than bugs alone, we still focus on the risk due to software changes only.

**File level bug prediction.** Researchers in this domain train prediction models to predict bug-prone locations (e.g., files or directories). Complexity metrics (e.g., McCabe's cyclomatic complexity metric [22] and the Chidamber and Kemerer (CK) metrics suite [8]), size (measured in lines of code) [15,19,21], and the number of prior changes and bugs are good predictors of buggy locations [1, 2, 18, 21, 26, 27, 29, 35].

There are some key differences between the aforementioned work and our work. First, our focus is on modelling risk, not only bugs. Bugs are a special case of risk. Second, we perform our modelling at the change level instead of at the file level. This difference is important. Flagging risky changes makes it easier to address the risk since changes can be flagged while they are still fresh in the developer's mind and fixed before they are integrated with the rest of the code base. In contrast, bug prediction flags files later in the release cycle, at which time a developer may have forgotten the issues surrounding their changes [20]. Furthermore, changes can be easily assigned to the developer who made the change, in contrast to files in bug prediction, which are changed by many developers, making it hard to decide who to assign the file to. Finally, changes provide the necessary context to address the flagged issue, whereas in bug prediction in some cases a bug spans many files that are changed together.

**Change level bug prediction.** The majority of the change-level related work aims to predict bug-introducing changes. On the other hand, our aim is to understand and identify risky changes (which are a superset of bug-introducing changes).

Sliwerski *et al.* [33] studied bug-introducing changes in Mozilla and Eclipse. They find that bug-introducing changes are part of large transactions and that bug fixing changes and changes done on Fridays have a higher chance of introducing bugs. Eyolfson *et al.* [12] study the correlation between a change's bugginess and the time of the day the change was committed and the experience of the developer making the change. They perform their study on the Linux kernel and PostgreSQL and find that changes performed between midnight and 4AM are more buggy than changes committed between 7AM and noon and that developers who commit regularly produce less buggy changes. Yin *et al.* [34] performed a study that characterizes incorrect bug-fixes in Linux, OpenSolaris, FreeBSD and a commercial operating system. They find that 14.8 - 24.2% of fixes are incorrect and affect end users, that concurrency bugs are the most difficult to correctly fix and that developers responsible for incorrect fixes usually do not have enough knowledge about the code being changed. Kim *et al.* [20] use change features like the terms in added and deleted deltas, terms in directory/file names, terms in change logs, terms in source code, change metadata and complexity metrics to classify changes as being buggy (i.e., bug-introducing) or clean (i.e, not bug-introducing).

Our work complements prior work on bug-introducing changes in a number of ways. First, we use a more general definition of change risk, assigned by developers who make the changes, which includes bug-introducing changes, i.e., bug-introducing changes are a special case of risky changes. Second, we perform our study on a large commercial system, whereas most of the prior work is performed on open source systems. Third, we quantify the effect of factors that indicate risky changes and compare them to those of bug-introducing changes.

Mockus and Weiss [25] assess the risk of Initial Modification Requests, called IMRs, which are groups of code changes, of the 5ESS commercial project. They predict the potential of an IMR to cause a failure (i.e., introduce a bug) using IMR diffusion, size, interval, purpose and experience metrics. Czerwonka *et al.* [9] present their experiences with CRANE, a tool used within Microsoft for failure prediction, change risk analysis and test prioritization.

Our work complements the work by Mockus and Weiss [25] in a number of ways. First, we provide recommendations at a finer

granularity (i.e., at the individual change level). Second, our unique data set allows us to study the risk as viewed by the developers making the changes, i.e., the risk is assigned by the individual developers making the changes instead of simply using the potential of change to cause a failure. Third, our work studies the impact of more factors and quantifies the effect of the important factors. Also, our work is different than the work by Czerwonka *et al.* [9] in two ways. First, in their definition of risk, the authors use the likelihood of a change to introduce a bug as a proxy for risk. Second, the authors perform their analysis at the granularity of a binary, which is generally made up of hundreds or thousands of files (i.e., equivalent to an IMR). In contrasts, our work is performed at the change level, a granularity much finer than the binary level. *To the best of our knowledge, our study is the only study that focuses on studying the risk of a change assigned by developers.*

# 3. CASE STUDY DATA

In this section, we describe the software system and the data used in our study.

## 3.1 The Software System

Our study is performed on a large, well established commercial software system. The system is written mainly in Java, with lower-level functionality implemented in the C/C++ language, and is used by tens of millions of users across the globe. The system is developed by many different teams, of which more than 60 were involved with our study, and has been in development for over 20 years. The current size of the code base is in the order of several millions of lines of code.

## 3.2 Change Data

To conduct our study, we used change-level information. Changes (or commits) are submitted to the Source Configuration Management (SCM) system by developers to perform maintenance tasks (e.g., fix bugs) or enhance features (e.g., implement new functionality). A change is done by one developer and may touch one or more files. The SCM stores meta-data about each change, such as the change ID, date and time of the change, the developer's name, the change type (e.g., bug fix?), whether the change fixes a previous change, the files touched and how many lines and chunks of code were added, deleted or modified for each file, together with a short description of the change. For the purpose of our study, an optional field (drop-down menu) was provided for developers to indicate whether they consider their change to be risky or not. By default, the drop-down is blank, indicating an unclassified change. Developers are then given the option to assign the right level of change risk.

*The general rule communicated to all developers regarding risky changes is that a change is considered risky if additional attention like careful code reviewing and possibly more testing is deemed necessary*. On the other hand, a non-risky change is one where the change does not need any special treatment in terms of code review or testing. The change can be safely integrated into the code without having any (expected) negative impact.

The change data was extracted and parsed in order to extract different factors that we use to perform our study. The factors are detailed later in Section 4.

## 3.3 Summary of Data

We observed and collected changes made between December 2009 and December 2010. Our final data set contained changes made by over 450 unique developers, spanning more than 60 different teams. For the most part, each team was responsible for one component. In rare cases, large components were assigned multiple teams (e.g., the multimedia component would have two teams, an audio team and a video team). Since assigning the risk to changes is optional, we found that, on average, developers assigned risk to more than 40% of all the changes they submitted. After removing sync and branch changes (which did not modify any code), our final data set contained a total of over 7,000 changes with risk assigned.

Given the fact that such data is rarely available to researchers, we were extremely grateful to have such a rich data set to perform our study. Due to confidentiality restrictions, we are not able to be more specific about the exact numbers or provide any more details about the used data. That said, we believe that the details given provide sufficient context about our study.

# 4. CASE STUDY SETUP

We begin by presenting the various factors used to study the risk of changes. Then, we describe the modelling techniques used and our evaluation criteria for the generated models.

The goal of our study is to better understand risky changes, so that they can be addressed by practitioners and their risk can be mitigated. In particular, we would like to identify risky changes and determine which factors are associated with them. We formalize our study into the following research questions:

**RQ1 Can we effectively identify risky changes?**

**RQ2 Which factors play an important role in identifying risky changes? What is the effect of these factors on the riskiness of a change?**

To answer the aforementioned questions, we study a number of factors that we use to empirically study risky changes.

## 4.1 Studied Factors

Table 1 shows all of the factors used in our study. For each factor, we provide its type (e.g., numeric), an explanation of the factor and the rationale for using the factor in our study.

We group the studied factors into six different dimensions:

**Time:** The main motivation for using this dimension is to study whether the time when changes are made has an impact on their risk, since prior work used time factors to model bug introducing changes in Open Source software [12, 33] .

**Size:** Prior work showed that churn is a good indicator of bugginess at the file [28] and IMR [25] level. IMRs consist of multiple Modification Requests (MR), which are made up of multiple changes, hence they are at a much coarser granularity than our changes. We investigate whether the size of a change (measured in number of added, deleted and changed lines) is also a good indicator of change risk. Furthermore, we use size as a proxy for complexity since prior work showed that complexity measures are highly correlated with size [15]. In addition to counting the number of lines, we also consider the number of locations (i.e., chunks) that these lines were spread over.

**Files:** Prior work showed that process metrics, such as the number of prior bugs, are a good indicator of future bugs [26, 35]. Therefore, this dimension considers the history of the files being modified by the change. For example, if a file has been changed many times in the past, then a change that modifies this file may be more risky. Since most other metrics only provide a snapshot view at the time the change was made, using the history of the files modified by the change is a way of incorporating history into our change risk models.

**Code:** The motivation behind using the code dimension is to study whether the code being modified (e.g., API code) by the change is

a good indicator of whether or not a change is risky. Since the software system under study is written in different programming languages, we introduced four boolean variables that indicate whether or not a change modifies Java code, C++ code, other code (e.g., html or xml pages) or API code. The type of code being modified provides insight into whether the change deals with application layer code or lower level OS code. The file extension was used to determine the type of changed code.

**Purpose:** Prior work showed that the purpose of an IMR (e.g., whether it was a bug fixing change) is a good indicator of its failure potential [25]. We study whether the purpose of a change impacts its risk.

**Personnel:** Prior work showed that the experience of the developers changing an IMR is a good indicator of its risk [25]. Therefore, we also investigate the impact of the developer experience in identifying risky changes.

In total, we extracted 23 different factors that covered six different dimensions. It is important to note that all of our factors can be easily extracted from the change log stored in widely available source code control repositories. This was pointed out by the industrial partner to be a major advantage of this work and makes this work applicable to any company or project that uses source code control repositories.

## 4.2 Logistic Regression Models

In this work, we are interested in identifying risky changes and determining which factors best indicate risky changes. Similar to prior work [35], we use a logistic regression model. A logistic regression model correlates the independent variables (i.e., the 23 factors in Table 1) with the dependent variable (i.e., whether or not a change is risky).

The output of our logistic regression model is a probability (between 0 and 1) of the likelihood that a change is risky. Then, it is up to the user of the output of the logistic regression model to determine a threshold at which she/he will consider a change as being risky. Generally speaking, a threshold of 0.5 is used. For example, if a change has a likelihood of 0.5 or higher, then it is considered risky, otherwise it is not.

However, the threshold is different for different data sets and the value of the threshold affects the precision and recall values of the prediction models. In this paper, we determine the threshold for each model using an approach that examines the tradeoff between type I and type II errors [25]. Type I errors correspond to files that are identified as being risky, while they are not. Having a low logistic regression threshold (e.g., 0.01) increases type I errors: a higher fraction of identified changes will not really be risky. A high type I error leads to a waste of resources since many non-risky changes need to be reviewed in vain. On the other hand, the type II error is the fraction of risky changes that are not identified as being risky when they should be. Having a high threshold can lead to large type II errors, and thus missing many risky changes.

To determine the optimal threshold for our models, we perform a cost-benefit analysis between the type I and type II errors. Similar to previous work [25], we vary the threshold value between 0 to 1, in increments of 0.01, and use the threshold where the type I and type II errors are equal. We report the thresholds used for each model in the results tables of Sections 5 and 6.

Initially, we built the logistic regression model using all 23 factors. Having a large number of factors is beneficial since it allows us to conduct a comprehensive study (i.e., take into account many factors in our models). However, using many factors in our models introduces the risk of having issues due to multi-collinearity. Multi-collinearity is caused by having highly correlated factors in a single model, making it difficult to determine which factors are actually causing the effect being observed and introducing high variance to the corresponding coefficients [7]. To alleviate such collinearity issues, we employ feature selection [14] to remove all redundant (i.e., highly correlated) factors from our models. In particular, we use the `cfs` selector [16], which performs the feature selection based on correlation and entropy. To ensure that the effect of the independent variables is statistically significant, we perform an ANOVA analysis and retain all variables with p-value $< 0.05$. We provide a list of the factors used in our models in Tables 6 and 7.

**Table 2: Confusion Matrix**

| | **True class** | |
|---|---|---|
| **Classified as** | Breakage | No Breakage |
| Breakage | TP | FP |
| No Breakage | FN | TN |

## 4.3 Evaluating the Accuracy of Our Models

We use two criteria to evaluate the performance of the logistic regression models: Predictive Power and Explanative Power.

### *4.3.1 Explanative Power*

Explanative power ranges between 0-100%, and quantifies the variability in the data explained by the model, i.e., how well the model fits the data. When calculating the explanative power, the model is built using all of the data (i.e., we do not split the data into training and testing sets). In addition, we report and compare the variability explained by each factor used in the model, to determine which of the factors are most important. The relative importance of each factor is determined by comparing its explained variability to that of the other factors in the model.

### *4.3.2 Predictive Power*

Predictive power measures the accuracy of the model in modelling the risk of a change. We calculate recall and relative precision based on the classification results in the confusion matrix (shown in Table 2).

**Recall:** is the percentage of correctly classified risky changes relative to all of the changes that are actually risky: Recall $= \frac{TP}{TP+FN}$. A recall value of 100% indicates that every risky change was classified as being risky.

**Relative Precision:** is the improvement in precision by our prediction model over the precision of a baseline model. In our case, the baseline model is a model that randomly predicts risky changes. For example, if a baseline model randomly predicts risky changes and achieves a precision of 20%, while our proposed prediction model achieves a precision of 40%, then the relative precision is given as $\frac{40}{20} = 2X$. In other words, using our model provides twice the precision of the baseline model. The higher the relative precision value the better the model is at classifying risky changes. We use relative precision instead of actual precision for confidentiality reasons, since precision allows one to infer the ratio of risky-changes in our dataset.

When evaluating the predictive power of our models, we employ 10-fold cross validation [11], where the data set is divided into 10 sets, each containing 10% of the data. One set serves as the testing data and the remaining nine sets are used as training data. The model is trained using the training data and its accuracy is tested using the testing data. In our results, we report the average from the 10-fold cross validation.

# 5. PRELIMINARY ANALYSIS

Prior to delving into our case study, we discuss our initial findings concerning change risk assignment. We started our analysis by

**Table 1: Factors Used to Study Risky Changes**

| Dim. | Factor | Type | Explanation | Rationale |
|---|---|---|---|---|
| Time | Hour | Numeric | Time when the change was made, measured in hours (0-23). | Changes performed at certain times in the day, e.g., late afternoons, might be done by over-worked or less aware developers, hence, these changes may be more risky [12]. |
| | Weekday | Numeric | Day of the week (e.g., Mon, Tue) when the change was performed. | Changes performed on specific days of the week (e.g., Fridays) are not as carefully examined and might be more risky [33]. |
| | Month day | Numeric | Calendar day of the month (1-31) when the change was performed. | Changes performed during specific periods, i.e., beginning, mid or end of the month might be rushed to meet end-of-the-month quotas and are likely to be more risky. |
| | Month | Numeric | Month of the year (0-11) when the change was performed. | Changes performed in specific months, e.g., later in the year or during holiday months like December, when less developers and expertise are available, might be more risky. |
| Size | Lines Added | Numeric | The number of lines added as part of the change. | Changes that add more lines add new functionality that has not been tested as thoroughly, therefore, they might be more risky. |
| | Chunks Added | Numeric | The number of chunks (i.e., different sections) added as part of the change. | Changes that add more chunks, i.e., are more spread out, are harder to make and hence are considered more risky. |
| | Lines Deleted | Numeric | The number of lines deleted as part of the change. | Changes that delete more code might remove too much or remove code incorrectly, making the change more risky. |
| | Chunks Deleted | Numeric | The number of chunks (i.e., different sections) deleted as part of the change. | Changes that delete more chunks, i.e., are more invasive, are harder to make and are more risky. |
| | Lines Modified | Numeric | The number of lines modified as part of the change. | Changes that modify more lines have a higher chance of making incorrect changes and are therefore more risky. |
| | Chunks Modified | Numeric | The number of chunks (i.e., different sections) modified as part of the change. | Changes that modify more chunks, i.e., are more invasive, are harder to make and are considered more risky. |
| | Churn | Numeric | The total number of lines added, deleted and modified as part of the change. | Changes that have high churn are harder to make and are considered more risky [28]. |
| Files | Number of Files | Numeric | The number of files modified by the change. | Changes that touch more files require a higher degree of knowledge of the different files and are therefore more risky [18, 33]. |
| | No. file devs | Numeric | The number of unique developers that modified the changed files. If a change modifies multiple files we use the number of developers of the file that has the most developers. | Files that have been changed by many developers are hard to modify. A change that touches a file that has been modified by many different developers is more risky [5]. |
| | No. file changes | Numeric | The number of past changes to the files modified by the change. If a change modifies multiple files, we use the number of changes of the file with the most past changes. | Files that are changed often are hard to modify. A change that touches such a file is more risky [26]. |
| | No. file fixes | Numeric | The number of past bug fixes to the files modified by the change. If a change modifies multiple files, we use the number of bug fixes of the file with the most past bug fixes. | Files that are fixed often tend to be buggy. A change that touches such files is more risky [35]. |
| | File bugginess | Numeric | The ratio of bug fixes to total changes of a file. If a change touches more than one file, we use the value of the file with the largest file bugginess. | Files may be changed often to make additions or general improvements, however if most of those changes are fixing bugs, then a change that touches such files is more risky. |
| Code | Modify Java | Boolean | Indicates whether the change modifies Java code. | Changes that modify code are changing application behaviour and hence are more/less likely to be risky. |
| | Modify CPP | Boolean | Indicates whether the change modifies C++ code. For this project, only low-level functionality was implemented in C++. | Changes that change low-level functionality are more risky. |
| | Modify Other | Boolean | Indicates whether the change modifies anything other than Java and C++ code, e.g., documentation files. | Changes that do not change code are less risky. |
| | Modify API | Boolean | Indicates whether the change modifies any APIs. | Changes that modify APIs can potentially affect all client code using the API, hence they are more likely to be risky. |
| Purpose | Bug Fix? | Boolean | Indicates whether the change fixes a bug. | Changes that fix a bug are more complex and are therefore more risky [33]. |
| | No. of Linked Bug Reports | Numeric | Indicates the number of bug reports that are linked to the change. | Changes that are linked to multiple bug reports need to make larger changes and are therefore more risky. |
| Personnel | Dev. Experience | Numeric | Indicates the experience of the developer who made the change. Experience is measured as the number of previous changes (from the start of the project) done by the developer. | Changes done by experienced developers are less risky [5]. |

**Table 3: Role of Developer and Team Name on Change Risk Classification**

| | Predictive Power | | | Explanative Power |
|---|---|---|---|---|
| **Component** | Precision | Recall | Thresh. | Deviance Explained |
| **All Factors** | 1.32X | 59.4% | 0.482 | 4.4% |
| **All Factors + Team** | 1.42X | 67.5% | 0.496 | 14.2% |
| **All Factors + Developers** | 1.72X | 77.5% | 0.496 | 32.6% |

building a general model based on the changes of all the developers combined, similar to prior work (e.g., [20]). The results of the model are shown in the first row (labeled "All Factors") of Table 3. This table also contains the threshold value used for the logistic regression prediction model that we determined based on the training data set (not the testing data set).

Our findings show that our model achieves good predictive power (i.e., precision and recall), however, the explanative power of the model is very low. We qualitatively examined a random subset of 50 risky changes to try and understand this low explanative power. We found that, although all developers were given the same criteria to label risky changes, the concrete interpretation of risk is ultimately a concept that depends on the individual developers and teams. For example, teams that worked on application-level code were less likely to mark their changes risky unless they were large. On the other hand, members of the UI team would mark their changes risky if they thought that their changes would impact other parts of the code, regardless of the size of the change. The same was observed for developers as well, each had their own criteria for marking risky changes.

Following this finding, we decided to investigate whether or not the team and the developer assigning the risk played a role in the assigned change risk. In our case, a team is composed of multiple developers and each team works on one component. We added the team name to the initial model as an additional factor. The results were much better, as shown in the second row of Table 3 (labeled "All Factors + Team"). Adding the team name improves both predictive and explanative power, indicating that when the risk of a change is considered one needs to discriminate between changes from different teams. Next, we added the developer name to the model containing all factors, as shown in the third row of Table 3 (labeled "All Factors + Developers"). We observe that adding the developer name to the model improves the predictive and explanative power even further.

Our findings here show that, although all developers where given the same rule to classify risky changes, the risk assigned to a change depends on the developer that is assigning the risk and the team that the developer belongs to. Based on these findings we recommend that developer or team specific models should be built when modelling change risk. Building one model to model the risk of *all* changes (i.e., changes from different developers) is not an effective solution.

*Even when all developers are given the same rule to classify risky changes, the risk of a change varies and depends on the developer that is assigning the risk and the team that the developer belongs to.*

# 6. CASE STUDY

In this section, we answer the research questions posed earlier. In particular, we examine the accuracy of our approach in identifying risky changes. Then, we determine the most important factors when identifying risky changes, as well as the specific impact of the factors.

## RQ1. Can we effectively identify risky changes?

**Motivation:** In order to address and assign the proper quality assurance efforts, we need to be able to effectively identify risky changes. Our goal is to examine whether it is feasible to build accurate models that flag risky changes.

**Approach:** In the previous subsection, we showed that the team and the developer play a major role in the accuracy of the change risk models. Therefore, we now build specific models at two levels: the *developer level* and the *team level*. At the developer level, we build a specific model for each developer (instead of one global model with the developer name as an independent variable). At the team level, we build a specific model for each team. Since these models are tailored to the individual teams and developers, we expect them to be more accurate than a global model that does not consider the team or developer.

In order to build the logistic regression models, we need to make sure that enough data was available for each developer. Therefore, we selected developers who made at least 20 changes over the year studied. Since we are building developer-specific models, we also require that a developer has both risky and non-risky changes. This is needed to train our models (i.e., we cannot train a good model using only risky changes or only non-risky changes). Therefore, we required that at least 20% of a developer's changes belong to either class, risky or non-risky. Then, we ranked the developers based on the total number of changes they committed and built models for the top 10. Ideally, we would want to make predictions for the developers with the most committed changes, since a manual risk assessment would be too time consuming for them. For developers that have fewer changes, manual examination might be a viable solution.

For the team models, we aggregated developers based on the team that they belong to. We ranked the teams based on the total number of changes and built models for the top 10 teams. Teams that have the most changes will benefit the most from our models since manual risk assessment of their changes will be a resource intensive task. As mentioned earlier for developers, for teams that have fewer changes, manual examination might be a viable solution.

**Results - Developer Level:** Table 4 shows the predictive and explanative power results for the top 10 developers. In terms of predictive power, our models achieved very promising results. On average the model achieves 1.87X relative precision (or a 87% improvement in precision over the baseline model), while achieving an average recall of 67.7%.

On average, the explanative power of our models is 20.8%. This explanative power is comparable to models that have been built in previous work to predict post-release bugs in files [3, 7].

**Results - Team Level:** Table 5 presents the results for the team level models. On average, the team level models achieve a relative precision of 1.37X and an average recall of 67.9%. In terms of explanative power, the team level models achieve an average explanative power of 13.3%.

As suggested by our preliminary analysis, the developer models outperform the team models in terms of predictive and explanative power. The main reason for this is the fact that the team models are less specific, since they incorporate changes from more developers.

**Table 4: Performance of Developer-Level Change Risk Classification**

| | Predictive Power | | | Explanative Power |
|---|---|---|---|---|
| **Dev.** | Precision | Recall | Thresh. | Deviance Explained |
| Dev1 | 1.58X | 66.8% | 0.464 | 22.6% |
| Dev2 | 1.50X | 55.1% | 0.43 | 22.0% |
| Dev3 | 2.03X | 64.1% | 0.32 | 15.3% |
| Dev4 | 2.76X | 75.5% | 0.302 | 42.6% |
| Dev5 | 1.69X | 76.6% | 0.544 | 12.3% |
| Dev6 | 1.61X | 64.9% | 0.518 | 18.5% |
| Dev7 | 1.28X | 48.0% | 0.394 | 8.0% |
| Dev8 | 1.82X | 65.2% | 0.416 | 19.9% |
| Dev9 | 1.72X | 81.8% | 0.55 | 27.4% |
| Dev10 | 2.72X | 77.8% | 0.482 | 19.0% |
| **Avg.** | 1.87X | 67.6% | - | 20.8% |

However, an advantage of team level models is that they are more practical, since we would need less models to be built (all developers of a team could share the same model). More importantly, team level models can be used by new developers who join the team; this is not possible with developer models.

**Final Remarks:** Another point worth addressing is the fact that relative precision values range between 1.5X - 2.76X for developers, and 1.09X - 1.76X for teams, and recall values range between 48.0 - 81.8% for developers, and 57.2 - 80.9% for teams. This range is due to the fact that different developers and teams have a different distribution of risky to non-risky changes. For example, Dev4 had more changes and a better balance of risky to non-risky changes than Dev7. Therefore, our prediction models were able to provide better accuracy for Dev4 than Dev7. That said, we believe that the average improvements provided by our prediction models are high enough to make them useful in practice.

*We can accurately identify risky changes, achieving average recall of 67% and precision improvement of 87% (for developer models) and 37% (for team models), over a baseline model.*

## RQ2. Which factors play an important role in identifying risky changes? What is the effect of these factors on the riskiness of a change?

**Motivation:** In addition to identifying risky changes with high accuracy, we are interested in knowing which factors are good indicators of risky changes and by how much these factors affect the riskiness of a change. Knowing which and by how much each factor relates to risky changes helps practitioners determine what factors they should consider carefully when determining which changes to carefully examine.

**Approach:** To study the importance of the factors in the prediction models, we perform an ANOVA analysis and examine the relative contribution (in terms of explanative power) of each factor to the logistic regression model.

In addition, similar to prior work [24], we measure the effect of each factor by building a model where all metrics are set to their median values (boolean factors are set to 0). Then, we double the median (boolean factors are set to 1) of one of the factors (while holding all other factors at their median values) and measure the difference in the modeled probabilities. The effect of a factor can be positive or negative. A positive effect indicates that a higher level of a factor corresponds to an increase in change risk, while a negative effect indicates that a higher level of a factor corresponds to a decrease in change risk.

The analysis is done for the models of the top 10 developers and teams mentioned in Tables 4 and 5, respectively. However, due to space limitations, we only show the results for the developers and teams 1, 5 and 10 in details since they represent the high, medium and low range of the top 10 developers and teams. Afterwards, we summarize and discuss all of the models in general.

**Results - Developer Level:** Table 6 shows the most important factors for Dev1, Dev5 and Dev10. Only the factors used in the final model (i.e., after applying feature selection and checking for statistical significance) are shown. The Explanative Power column shows the variability explained by each factor. The higher the deviance explained of the factor, the more important it is to the model. We use this measure as a way of gauging the importance of the factors. For example, for Dev1 the "Chunks Added" factor is the most important factor in determining the risky changes. This means that if a future change is made by Dev1 and there are many "Chunks Added", one has to be cautious about the change since it likely a high risk change.

The Effect column in Table 6 shows the effects of each factor for Dev1, Dev5 and Dev10. All of the factors have a strong positive effect with change risk. Comparing the different factors shows that for Dev1, the number of bug reports linked to a change (e.g., the change addresses a major bug or multiple bugs) has the strongest relationship with change risk. For Devs5 and 10, the number of code lines added also has a strong positive relationship with change risk. In addition, file bugginess has an extremely large positive relationship with change riskiness for Dev10.

**Results - Team Level:** Table 7 shows the most important factors for Team1, Team5 and Team10. In all three models, code additions (either the number of code chunks added or lines added) are strong indicators of risky changes. Once again, we find that all of the factors have a positive effect with risky changes, i.e., higher values indicate higher risk. For Team1, the number of bug reports linked to a change has a strong effect on risky changes. For Team5, we find that the number of fixes to the file modified by the change has the strongest effect. We were not able to calculate the effect for the hour metric, since doubling the median does not make sense (i.e., doubling hour 23 to be 46 does not make sense). For Team10, we find that the number of lines added and file bugginess both have a strong and positive relationship with change risk.

From the aforementioned results, we make two noteworthy observations. First, each developer and team has their own set of factors that best predict the risk of their changes. Second, the models are very simple, containing at most 3 or 4 factors. This simplicity makes these models more attractive to practitioners, who can easily apply and interpret such simple models in practice.

**Results - Summary** In addition to providing the important metrics for developers and teams 1, 5 and 10, we provide a summary of the important factors in each dimension for all of the 10 studied developers and teams in Table 8. The most important factor in each dimension is shown in the column labeled "Most Important Factor". The "Importance" column shows the number of the top 10 developers that a dimension was important for. For example, the lines added factor was the most important factor for 7 of the top 10 developers (as shown in the second row of Table 8). On the other hand, factors in the time dimension were not important for any of the top 10 developers.

From Table 8 we observe that, for both developer and team levels, the most important dimensions are the size and file dimensions,

**Table 5: Performance of Team Level Change Risk Classification**

| | Predictive Power | | | Explanative Power |
|---|---|---|---|---|
| **Component** | Precision | Recall | Thresh. | Deviance Explained |
| Team1 | 1.76X | 57.2% | 0.448 | 6.9% |
| Team2 | 1.57X | 80.9% | 0.524 | 22.6% |
| Team3 | 1.28X | 60.5% | 0.486 | 7.38% |
| Team4 | 1.15X | 77.3% | 0.588 | 9.0% |
| Team5 | 1.14X | 57.7% | 0.502 | 5.6% |
| Team6 | 1.09X | 79.4% | 0.59 | 10.0% |
| Team7 | 1.69X | 65.6% | 0.478 | 15.6% |
| Team8 | 1.43X | 69.9% | 0.508 | 16.6% |
| Team9 | 1.30X | 69.3% | 0.506 | 25.33% |
| Team10 | 1.25X | 71.2% | 0.534 | 13.9% |
| **Avg.** | 1.37X | 67.9% | - | 13.3% |

**Table 6: Most Important Factors for Devs1, 5 and 10**

| Model | Metric | Explanative Power | Effect |
|---|---|---|---|
| **Dev1** | Chunks Added* | 11.7% | 142% |
| | Chunks Deleted** | 5.8% | 120% |
| | Chunks Modified* | 2.8% | 131% |
| | No. of Linked Bug Reports* | 2.3% | 162% |
| **Dev5** | Lines Added** | 12.3% | 274% |
| **Dev10** | Lines Added* | 9.8% | 268% |
| | File Bugginess** | 9.2% | 1114% |

($p < 0.001$ ***; $p < 0.01$ **; $p < 0.05$ *)

with the number of lines of code added, number of chunks of code added, number of files and file bugginess being the most important factors within these dimensions. Purpose (No. of linked bug reports) and Personnel (Dev. experience) factors are the next most important dimensions, with code (modify CPP, for team level) and time (hour, for team level) dimensions being the least important.

*The number of lines and chunks being added, the bugginess of the files being changed, the number of linked bug reports to a change and the developer experience are the most important indicators of risky changes.*

# 7. DISCUSSION

The majority of prior work (e.g., [12, 20]) used bug-introducing changes as a measure of risky changes. However, we argue that risky changes are more than just bug-introducing changes. We believe that risky changes encompass bug-introducing changes as well as other changes that may have a high impact on the software product and/or its users.

To better understand this difference, we compare risky changes to bug-introducing changes by comparing the factors that best indicate risky changes and bug-introducing changes, as well as by analyzing the classification of bug-introducing changes.

## 7.1 Factors used to indicate bug-introducing and risky changes

Similar to the preliminary analysis in Section 5, we build a global model that includes the risky changes from all developers and we compare it to a model that contains the bug-introducing changes from all developers. Table 9 shows the factors in the resulting two models. We find that the number of lines added is an important factor for both bug-introducing and risky changes, having a higher effect for bug-introducing changes. However, for risky changes, two additional factors (i.e., the file bugginess and the number of developers who touched the changed files in the past) are considered to be important. This finding gives an indication that risky changes are likely different from bug-introducing changes, since different (and in this case more) factors are required to identify them.

**Table 7: Most Important Factors for Teams1, 5 and 10**

| Model | Metric | Explanative Power | Effect |
|---|---|---|---|
| **Team1** | Chunks Added** | 3.99% | 137% |
| | No. of Linked Bug Reports** | 1.68 % | 195% |
| | Number of Files* | 1.21% | 174% |
| **Team5** | Chunks Added** | 3.08% | 120% |
| | No. File Fixes** | 1.8% | 125% |
| | Hour* | 0.75% | - |
| **Team10** | Lines Added*** | 11.7% | 134% |
| | File Bugginess* | 2.2% | 138% |

($p < 0.001$ ***; $p < 0.01$ **; $p < 0.05$ *)

## 7.2 Classification of Bug-Introducing Changes as Risky Changes

We now investigate how accurate developers are at identifying bug-introducing changes as risky changes. To do so, we examine the entire set of 7,000 changes that had been assigned a risk value by the developers. We examine all of the changes labeled as *not* risky and determine how many of those changes introduced a bug, i.e., how often bug-introducing changes slipped through without being noticed as risky.

We construct a confusion matrix, similar to the one shown in Table 10. Due to confidentiality reasons, we are unable to show the exact numbers for the confusion matrix, i.e., we can only provide the corresponding ratio of accuracy. We calculate the ratio of accuracy of developers in classifying bug-introducing changes as $\frac{LI}{LI+LNI}$. That is, we measure the ratio of changes labeled as being *not* risky and introducing a bug (i.e., LI) divided by the total number of changes labeled as being *not* risky (i.e., LI+LNI). This value was 3.1%. This means that when developers classify a change as being *not* risky, they are correct 96.9% of the time that the change will not introduce a bug (although it could still cause other issues, such as delay, which are not considered to be bugs). This high level of accuracy is encouraging, showing that developers are good at assessing non-risky changes. However, this now brings up the question: Why are some bug-introducing changes misclassified by developers as being non-risky changes?

## 7.3 Why are Some Bug-Introducing Changes Misclassified?

Our finding shows that in some cases developers incorrectly classified bug-introducing changes, marking them as safe changes. In order to better understand why such changes were incorrectly classified, we compare all the correctly classified (i.e., marked as not risky and not introducing a bug) and all the incorrectly classified (i.e., marked as being not risky and later introducing a bug) changes on the following:

Table 8: Summary of Most Important Factors for Top 10 Developers and Teams

| Dim. | Developer-Level Most Important factor | Importance | Team-Level Most Important factor | Importance |
|---|---|---|---|---|
| **Time** | - | 0 | Hour | 2 |
| **Size** | Lines Added | 7 | Chunks Added | 10 |
| **Files** | No. of Files & File Bugginess | 7 | File Bugginess | 6 |
| **Code** | - | 0 | Modify CPP | 3 |
| **Purpose** | No. of Linked Bug Reports | 2 | No. of Linked Bug Reports | 4 |
| **Personnel** | Dev. Experience | 1 | Dev. Experience | 4 |

- **Cause for the change**: For each change, developers entered a reason for the change. We compared the percentages of each of the eight possible causes (shown in Table 11) between the correctly and incorrectly classified changes. The purpose of this analysis is to investigate whether there is a specific cause of a change that is more likely to be incorrectly classified.
- **Bug fixing change?**: We compare the percentage of bug-fixing changes in the correctly and incorrectly classified changes. The purpose of this analysis is to investigate whether bug fixing changes are more likely to be classified incorrectly.
- **Has related changes**: If a change has other changes related to it (e.g., it requires changes made by others or depends on functionality recently modified by other changes), those changes are explicitly added in the change commit log. We compared the percentage of changes that have related changes for the correctly and incorrectly classified changes. The intuition for looking at related changes is to examine if changes that have related changes are harder to classify.
- **Modifies API**: If a change modifies API code, it is flagged by developers. The main idea is to make other developers aware that this change could potentially affect other code. We examine the difference between correctly and incorrectly classified changes to see whether changes that change API code are more likely to be incorrectly classified.

Table 11 summarizes our findings. The table presents the average percentage of changes in each category. For example, the first row of the table shows that 11.7% of the incorrectly classified changes were due to unclear requirements, whereas 11.5% of the correctly classified changes were due to unclear requirements. In this case, it is clear that a change caused by unclear requirements has no increased chance of being incorrectly classified. We also see a very small difference in classification accuracy for changes made as a side-effect of other changes. Changes due to unclear documentation, due to inadequate testing, due to coding errors, due to design flaws and bug fixing changes are more likely to be correctly classified than not. Changes due to a scope change are slightly more likely to be incorrectly classified. In contrast, changes due to integration errors (i.e., the change was made to fix an integration error) and changes that modify API code are twice as likely to be incorrectly classified. Also, *changes that have related changes are 10 times as likely to be incorrectly classified*. This finding indicates that although developers are aware of the fact that there are related changes, they are not aware of the potential risk of these related changes (i.e., since they are marking them as being not risky, these changes end up introducing bugs later on).

To make sure that our findings are chosen from a representative sample, we measured the number of unique developers responsible for the changes used in this analysis. We found that the incorrectly classified changes were made by more than 60 unique developers and the correctly classified changes were made by more than 370 developers.

Table 9: Most Important Factors When Classifying Bug Introducing Changes

| Model | Metric | Effect |
|---|---|---|
| **Bug-Introducing Changes** | Lines Added*** | +180% |
| **Risky Changes** | Lines Added*** | +128% |
| | File Bugginess*** | +102% |
| | File Devs*** | +131% |

($p < 0.001$ ***; $p < 0.01$ **; $p < 0.05$ *)

Table 10: Risky vs. Bug-introducing Changes

| | Bug-Introducing | |
|---|---|---|
| **Risky** | Yes | No |
| High | HI | HNI |
| Low | LI | LNI |

Based on these findings, we recommend that developers carefully consider their risk assignments for changes that are caused by integration errors, that have related changes and that modify API code.

# 8. LESSONS LEARNT AND FUTURE WORK

After performing our study, we asked the opinions of an experienced development manager in the company about our findings. The manager leads one of the teams studied as part of this paper and is not a co-author of the paper.

The development manager was excited about the findings and suggested that we build a recommendation tool that can be leveraged by him and other team managers to assign quality assurance efforts for risky changes. Based on the prediction models in Section 6, we built a prototype tool that is currently being used by teams within the company to automatically classify their changes. The tool is still in its early stages and features are being added to improve it.

At this early stage, the tool is just starting to be used to classify risky changes. Instead of having to rely on gut feelings, developers can now verify their intuition with a tool that can quantify the risk of a change. Changes that have a mismatch between the tool's classification and the manual classification are being investigated in more detail. Furthermore, our approach is also used to classify the many "unclassified" changes (e.g., during the period of our study, nearly 60% of the changes were unclassified).

As for the developer-specific versus team-level models, the manager shared our belief that team-level models are more practical. However, he suggested that developer-level models would be more beneficial in cases where new developers join a team for short periods of time (e.g., when interns join the development team). This of course assumes that we have enough team history to train the models on.

The manager pointed out that the strength of this work lies in the fact that its findings are simple and easy to understand. A model

that is made up of 4-5 factors can be easily understood by managers, so they will know why changes are being flagged. This makes the model much more appealing than a black-box type solution where changes are flagged without any insight as to the rationale. In addition, he pointed out that it would be desirable for the work to also provide a possible course of action to mitigate the risk of a flagged change. For example, a model that flags a risky change might suggest the reduction in risk that can be achieved if unit testing or code reviews were performed on the change.

**Table 11: Comparison Between Correctly and Incorrectly Classified Changes**

| Category | | Incorrectly Classified | Correctly Classified |
|---|---|---|---|
| Cause | Unclear Requirement | 11.7% | 11.5% |
| | Side-effect of Other Changes | 7.3% | 6.4% |
| | Unclear Documentation | 0.7% | 1.5% |
| | Inadequate Testing | 0.73% | 2.3% |
| | Scope Change | 4.4% | 3.2% |
| | Coding Error | 28.5% | 37.2% |
| | Integration Error | 2.2% | 0.8% |
| | Design Flaw | 10.9% | 11.5% |
| **Bug fixing change** | | 70.1% | 77.9% |
| **Has related changes** | | 70.8% | 7.4% |
| **Modifies API** | | 2.9% | 1.5% |

# 9. LIMITATIONS

**Threats to Construct Validity** consider the relationship between theory and observation, in case the measured variables do not measure the actual factors.

Changes that introduced bugs were manually mapped in this project (i.e., the change that caused a bug was mapped to the change that caused the bug). Although this mapping was done by the project developers themselves, in certain cases, some changes might not have been mapped correctly or not mapped at all.

The risk value used in our study was manually assigned to changes by the developers who made the change. Hence, it is possible that the wrong risk value is assigned. However, our analysis (in Section 7.2) of the percentage of non-risky changes that introduced bugs showed that developers are accurate 96.9% of the time. Also, it is important to note that the risk was not assigned by a manager or any other person. The fact that this risk is assigned by the developer who made the change makes it very credible. Furthermore, we are not aware of any other data set that has manually assigned risk values to changes. That said, it would be ideal to have perfect knowledge of the risky changes and use these changes to perform our study. One possibility is to track changes and see which ones actually required additional review or testing. Another possibility is to do a pilot study in which multiple people assign a risk value to a change and the level of inter-rater agreement can be used to have more confidence in the assigned risk value. Since we do not have such data yet, we refer this research as future work.

When asked to assign the risk to changes, developers assigned risk to 40% of the changes. Our results may be affected by the fact that not all changes were assigned a risk value. However, our response rate of 40% from developers is at least as good as other software engineering studies, which have a response rate in the range of 14 - 33% [4, 31].

During our investigation as to how correct developers are in classifying bug-introducing changes, we looked at how correct developers are when they mark changes as being non-risky. We did not look into how correct developers are when marking a change as risky. The reason is that changes marked as being risky undergo more scrutiny and might be modified before being integrated into the code base. Hence, the link between risky changes and bug-introducing changes is biased. In contrast, our analysis on changes flagged as non-risky does not exhibit such bias.

**Threats to External Validity** consider the generalization of our findings. The studied project was a commercial project written mainly in Java and C/C++, therefore, our results may not generalize to other commercial or open source projects. That said, we do believe that some of the findings may hold for similar projects, especially large, commercial projects from the mobile domain. We also believe that our finding which shows that risk of changes depends on the developer or team who makes the change may hold for other software projects.

Some of the factors used in our study are project specific, e.g., modify Java and modify CPP factors in Table 1. The goal of using these metrics was to examine the effect of upper layer (e.g., application) versus lower layer (e.g., OS) changes on risk. Different projects may use other programming languages or architectural styles, in which case, our metrics cannot be directly used. However, we believe that using factors that differentiate between upper and lower level changes can be done for other projects.

# 10. CONCLUSION

Organizations are strongly interested in managing risk which is a considerably more encompassing concept than bugs which has been extensively studied by the software engineering research community. While a risky change might not introduce bug, it might lead to delays and large cost overruns. In this empirical study, the first of its kind, we looked at a unique data set about the risk of software changes to better understand the characteristics of risky changes. The main findings of our study are:

- When studying risky changes, the developer making the change and the team they belong to need to be considered.
- Risky changes can be effectively identified using factors such as the number of lines and chunks added by the changes, the bugginess of the files being changed, the number of bug reports linked to the change and the experience of the developer making the change.
- We find that developers are accurate 96.1% of the time when identifying bug-introducing changes. However, developers' identification of risky changes is less reliable. Especially, when changes have many related changes.

Our study opens a new avenue for Software Engineering research related to risk management within software organizations, and not only bugs, introduced by changes. We plan (and encourage other researchers) to further develop on the findings of this paper. We see many potential avenues for future work related to risk management as a key and important concept in the production of software today.

Furthermore, one of the key lessons that we learned through this study is that practitioners are willing to get involved in research, as long as their commitment is kept to a minimum and the data collection is done in a non-intrusive manner.


## Acknowledgments

We would like to thank Research in Motion (RIM) for providing support and data access for this study. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of RIM and/or its subsidiaries and affiliates. Moreover, our results do not in any way reflect the quality of RIM's products.

## 11. REFERENCES


[1] Erik Arisholm and Lionel C. Briand. Predicting fault-prone components in a Java legacy system. In *ISESE '06: Proceedings of International Symposium on Empirical Software Engineering*, pages 8–17, 2006.

[2] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, 1996.

[3] Nicolas Bettenburg and Ahmed E. Hassan. Studying the impact of social structures on software quality. In *Proc. Int'l Conf. on Program Comprehension*, pages 124–133, 2010.

[4] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proceedings of Intl Sym. on Foundations of Software Engineering*, pages 308–318, 2008.

[5] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of Sym. and European Conf. on Foundations of Software Engineering*, pages 4–14, 2011.

[6] Barry W. Boehm. Software risk management: Principles and practices. *IEEE Softw.*, 8(1):32–41, January 1991.

[7] Marcelo Cataldo, Audris Mockus, Jeffrey A. Roberts, and James D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. *IEEE Trans. on Softw. Eng.*, 99(6):864–878, 2009.

[8] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[9] Jacek Czerwonka, Rajiv Das, Nachiappan Nagappan, Alex Tarvo, and Alex Teterev. Crane: Failure prediction, change analysis and test prioritization in practice – experiences from windows. In *Proceedings of Intl Conf. on Software Testing, Verification and Validation*, pages 357–366, 2011.

[10] F. M. Dedolph. The neglected management activity: Software risk management. *Bell Labs Tech. Journal*, 8(3):91–95, 2003.

[11] Bradley Efron. Estimating the error rate of a prediction rule: Improvement on Cross-Validation. *Journal of the American Statistical Association*, 78(382):316–331, 1983.

[12] Jon Eyolfson, Lin Tan, and Patrick Lam. Do time of day and developer experience affect commit bugginess. In *Proceeding of Working Conf. on Mining Software Repositories*, pages 153–162, 2011.

[13] Bernd Freimut, Susanne Hartkopf, Peter Kaiser, Jyrki Kontio, and Werner Kobitzsch. An industrial case study of implementing software risk management. In *Proc. of European Software Engineering Conf. and Intl Sym. on Foundations of Software Engineering*, pages 277–287, 2001.

[14] Kehan Gao, Taghi M. Khoshgoftaar, Huanjing Wang, and Naeem Seliya. Choosing software metrics for defect prediction: an investigation on feature selection techniques. *Softw. Pract. Exper.*, 41:579–606, April 2011.

[15] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. of Softw. Eng.*, 26(7):653–661, July 2000.

[16] M A Hall and L A Smith. Practical feature subset selection for machine learning. *Computer Science*, 98:181–191, 1998.

[17] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic review of fault prediction performance in software engineering. *IEEE Trans. on Softw. Eng.*, 99, 2011.

[18] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of Intl Conference on Software Engineering*, pages 78–88, 2009.

[19] Israel Herraiz, Jesus M. Gonzalez-Barahona, and Gregorio Robles. Towards a theoretical model for software growth. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 21, 2007.

[20] Sunghun Kim, E. James Whitehead, Jr., and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Softw. Eng.*, 34(2):181–196, 2008.

[21] Marek Leszak, Dewayne E. Perry, and Dieter Stoll. Classification and evaluation of defects in a project retrospective. *J. Syst. Softw.*, 61(3):173–187, 2002.

[22] Thomas J. McCabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, page 407, 1976.

[23] J.A. Miccolis, K. Hively, and B.W. Merkley. *Enterprise Risk Management: Trends and Emerging Practices*. Institute of Internal Auditors Research Foundation, 2001.

[24] Audris Mockus. Organizational volatility and its effects on software defects. In *Proceedings of Intl Sym. on Foundations of Software Engineering*, pages 117–126, 2010.

[25] Audris Mockus and David M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[26] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of Intl Conf. on Software Engineering*, pages 181–190, 2008.

[27] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 580–586, 2005.

[28] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 284–292, 2005.

[29] Niclas Ohlsson and Hans Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Trans. Softw. Eng.*, 22(12):886–894, 1996.

[30] I Ozkaya, P. Kruchten, R. Nord, and N. Brown. Second international workshop on managing technical debt, 2011.

[31] Teade Punter, Marcus Ciolkowski, Bernd Freimut, and Isabel John. Conducting on-line surveys in software engineering. In *Proceedings of Intl Sym. on Empirical Software Engineering*, pages 80–88, 2003.

[32] Peter C. Rigby, Daniel M. German, and Margaret-Anne Storey. Open source software peer review practices: A case study of the apache server. In *Proceedings of Intl Conf. on Software Engineering*, pages 541–550, 2008.

[33] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of Intl Workshop on Mining Software Repositories*, pages 1–5, 2005.

[34] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of Sym. and European Conf. on Foundations of Software Engineering*, pages 26–36, 2011.

[35] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for Eclipse. In *Proceedings of Intl Workshop on Predictor Models in Software Engineering*, pages 9–15, 2007.