The Impact of Human Discussions on Just-In-Time Quality Assurance

An Empirical Study on OpenStack and Eclipse

Parastou Tourani Bram Adams
Polytechnique Montréal, Canada
{parastou.tourani, bram.adams}@polymtl.ca

***Abstract*—In order to spot defect-introducing code changes during review before they are integrated into a project's version control system, a variety of defect prediction models have been designed. Most of these models focus exclusively on source code properties, like the number of added or deleted lines, or developer-related measures like experience. However, a code change is only the outcome of a much longer process, involving discussions on an issue report and review discussions on (different versions of) a patch. Similar to how body language implicitly can reveal a person's real feelings, the length, intensity or positivity of these discussions can provide important additional clues about how risky a particular patch is or how confident developers and reviewers are about the patch. In this paper, we build logistic regression models to study the impact of the characteristics of issue and review discussions on the defect-proneness of a patch. Comparison of these models to conventional source code-based models shows that issue and review metrics combined improve precision and recall of the explanatory models up to 10%. Review time and issue discussion lag are amongst the most important metrics, having a positive (i.e., increasing) relation with defect-proneness.**

I. Introduction

The later defects are identified and fixed, the more expensive they become [8]. This is why companies try to limit the number of major defects discovered by end users to a minimum, as such defects may degrade a company's reputation and lead to irreversible financial loss. In addition to a well thought-out quality assurance strategy comprising tests, reviews and other activities [13], prediction models of defect-prone files and defect-introducing patches have gained importance.

Initial work on defect prediction considered files or modules as granularity for predictions [17], [18], [30], where models would predict which files have the highest probability of containing defects and hence should be tested more thoroughly. Later studies [12], [21], [22], [29], [37] suggested Just-In-Time (JIT) defect prediction models that focus on software patches instead of on files or modules. Such models are more actionable and provide large effort savings [21], since they locate specific patches as defect-prone, instead of large files or modules. In addition, predictions can be applied at the exact time when a patch is being reviewed and consequently, the responsible developer can intervene quickly.

State-of-the-art approaches for JIT prediction mainly use measures related to the actual code that is being changed, such as the size of the code change (churn) or the developer's track record in the project, but ignore the actual feelings and insights of the stakeholders involved in reviewing a patch or (earlier on) commenting on the bug or feature request that the patch is implementing. For example, when an issue report provokes many comments by different people in a relatively short time frame, this could (amongst others) indicate that the corresponding bug or feature is complicated and hence has a higher risk of introducing bugs than an issue report with only one or two comments. Similarly, a patch requiring multiple revisions before being accepted or yielding many code comments during review could indicate an increased risk, even though the reviewers in the end accepted the patch.

To understand the relation between issue/review discussions and the defect-proneness of a patch, this paper performs a large empirical case study on 10 OpenStack and 5 Eclipse open source projects. We chose these projects, as they are large projects that have adopted code reviews on a large scale and have a reasonable traceability between commits, reviews and issue reports. Using comments in the issue tracking and review repositories as proxy for human discussions, we compare explanatory models containing issue and/or review metrics to a baseline model containing only change-related metrics. Using this comparison, we address the following research questions:

RQ1) How well can issue discussion metrics explain defect-introducing changes?

For five of the project, explanatory JIT models with issue discussion-related metrics show improvements of 3% to 10% in precision or recall.

RQ2) How well can review discussion metrics explain defect-introducing changes?

Models with review discussion-related metrics show similar improvements as RQ1 in precision or recall for 9 projects . Some review discussion metrics figure amongst the most impactful metrics.

RQ3) How well can issue and review discussion metrics explain defect-introducing changes?

For half of the projects, models augmented with both review and issue discussion metrics improve precision and recall by 3% up to 17% (in one case). Review time

and issue discussion lag are amongst the most important metrics in the combined models.

In the remainder of this paper, we first describe the necessary background notions for our work (section II). Next, we describe the case study setup (section III), then present the results of the three research questions (section IV). After discussion and threats to validity (section V and section VI), we discuss related work (section VII) and we finish with conclusions (section VIII).

II. Background

This section provides background information about JIT models and issue/review environments.

A. Source Code-based JIT Models

Various researchers have built JIT models [12], [21], [22], [29], [37]. Here, we focus on the more recent work of Kamei et al. [21]. They considered 14 software change metrics, grouped into 5 dimensions, to explain whether a code change introduces a defect. These metrics, which are derived from the source code repository data of a project, would be measured for a new code change and plugged into a prediction model trained on earlier code changes (for which one knew in the meantime whether they introduced a defect) to obtain a risk probability. Based on a threshold, this probability would then suggest whether or not this new code change is expected to be defect-prone. Developers could then immediately review, test and revise the code change and resubmit.

Table I shows the 14 metrics and 5 dimensions, as well as their rationale. All of them can be calculated using only the project's version control system, such as Subversion or Git. The resulting models performed well on a large set of open and closed source systems. Hence, we use these change metrics to build a baseline model to which we compare our new models that add issue and/or review discussion metrics.

B. Issue and Review Repositories

An issue tracking system (e.g., Bugzilla, Launchpad or Jira) is a repository used by a software organization to enable users and developers to report defects and feature requests. It allows such a reported issue to be triaged and (if deemed important) assigned to team members, to discuss the issue with any interested team member and to track the history of all work on the issue. During these issue discussions, team members can ask questions, share their opinions and help other team members. Some projects also use an issue tracking system to review patches and bug fixes (instead of using a dedicated reviewing environment like Gerrit), however we do not consider such projects in this paper.

According to Rigby et al [36], modern software organizations have embraced lightweight processes for reviewing code changes, i.e., to decide whether a developer's change is safe to integrate into the official version control system. During such lightweight code review, assigned reviewers make comments on a code change or ask questions that can lead to a discussion of the change and/or different revisions of the code change, before a final decision is made about the code change. If accepted, the most recent revision of the code change can enter the version control system, otherwise the change is abandoned and the developer will move on to something else. Modern web apps like Gerrit are used to support this review process.

In this paper, we are interested in understanding whether the characteristics of collaborative group discussions of issue reports and code reviews, such as the discussion volume, intensity or positivity, can provide hints about the defect-proneness of the resulting accepted code change. For example, the behaviour of the discussion participants could show them being uncomfortable with a code change, even though it would pass through review. For this reason, we build explanatory models with issue and/or review metrics to understand why code changes that successfully passed reviewing turn out to be defect-introducing.

III. Case Study Setup

This section explains the methodology used to address our three research questions. We discuss the selection of case study systems, identification of defect-prone code changes, extraction of issue and review metrics, model building and model validation.

A. Selection of Case Study Systems

Our aim is to study whether human discussions of issue reports and code review can provide some indication that a code change that successfully passed review introduces a defect. This model would be used right after code review finished, as a final check before the code change would be integrated into the version control system. The only requirement is that the code change is linked to an issue report and code review, which is readily done by modern software organizations. Therefore, to conduct our empirical study, we required projects with a substantial number of commits linked to issues and reviews.

Openstack[1] and Eclipse[2] are two popular open source ecosystems featuring a large number of sizeable projects that have adopted code reviewing in the last couple of years (issue reports have been adopted by open source projects for a long time). They have adopted modern technology for version control systems (git), issue repositories (Bugzilla and Launchpad) and code review repositories (Gerrit). Furthermore, they are among the pioneers of pushing developers to explicitly link code changes to issues and reviews.

To obtain the version control, issue report and code review data of these ecosystems, we used the data set graciously provided by Gonzalez-Barahona et al. [14]. They developed the MetricsGrimoire tool suite to mine the repositories of OpenStack and Eclipse, then store the corresponding data into a relational database. We used their version control, issue report and code review data sets[3] to perform our study.

[1] http://openstack.org
[2] http://eclipse.org
[3] http://gsyc.es/~jgb/repro/2015-msr-grimoire-data

TABLE I: Overview of change metrics of Kamei et al. [21].

Dimension	Name	Definition	Expected Rationale
Diffusion	NS	#modified subsystems	Modifying more subsystems increases the defect-proneness.
	ND	#modified directories	Modifying more directories increases the scattering of change and the probability of defects.
	NF	#modified files	Modifying more files increases the probability of defects.
	ENT	distribution of changes across all files	Changes affecting multiple files *equally* likely are more risky.
Size	LA	#lines of code added	Larger changes increase the probability of defects.
	LD	#lines of code deleted	Defect-proneness increases when more code is removed, since incorrect code could be deleted accidentally.
	LT	#lines of code in a file before the change	Larger code might be more complex to understand, hence, modifying it is more likely to be defect-prone
Purpose	FIX	is the change a defect fix?	Fixing a defect touches a buggy area of the code, hence the probability of introducing a new defect is higher.
History	NDEV	avg. #developers that changed the files before	Different developers modifying the same file may lead to misunderstanding.
	AGE	average time (#days) since the last change	Recently changed files are more defect-prone than stable code.
	NUC	#unique changes to modified files	The more files have been changed, the more opportunities for defects.
Experience	EXP	#prior commits by the developer	More experienced developers are less likely to introduce defects, unless they make more ambitious changes.
	REXP	#prior commits by the developer weighted by their age	Developers that recently modified a file have more fresh knowledge about the code base.
	SEXP	#prior commits by the developer on a subsystem	Developers that are dominant in a particular subsystem are less likely to introduce a defect there.

TABLE II: Statistics of the Studied OpenStack and Eclipse Projects.

	Project	Total #commits (%defective)	Start Date	Total #reviews (%linked)	#commits linked (%total)	Total #issues (%linked)	#commits linked (%total)	#commits linked to both (%total)
Openstack	cinder	5013 (30%)	2012-05-03	4209 (65%)	2721 (54%)	2057 (78%)	1654 (33%)	1654 (33%)
	devstack	5515 (13%)	2011-09-11	3271 (77%)	2507 (45%)	1088 (70%)	827 (15%)	827 (15%)
	glance	4125 (28%)	2010-08-11	2385 (77%)	1844 (44%)	1521 (75%)	1073 (26%)	1155 (28%)
	heat	7341 (23%)	2012-03-13	5324 (72%)	3859 (52%)	1781 (83%)	1835 (25%)	1982 (27%)
	keystone	6654 (20%)	2011-04-14	4214 (73%)	3084 (46%)	1964 (73%)	1530 (23%)	1996 (30%)
	neutron	8512 (28%)	2010-12-31	6806 (64%)	4384 (51%)	3960 (70%)	2979 (35%)	2979 (35%)
	nova	32998 (25%)	2010-05-30	14699 (79%)	11583 (35%)	8210 (68%)	5940 (18%)	6600 (20%)
	openstack-manuals	8483 (26%)	2011-09-20	7356 (67%)	4951 (58%)	2867 (79%)	2121 (25%)	2290 (27%)
	swift	4333 (17%)	2010-07-12	2203 (72%)	1580 (36%)	1106 (46%)	953 (22%)	1043 (24%)
	tempest	5813 (18%)	2011-08-26	3875 (80%)	3105 (53%)	1619 (70%)	1337 (23%)	1453 (25%)
Eclipse	cdt	24025 (40%)	2002-06-26	1006 (92%)	721 (3%)	13576 (54%)	480 (2%)	9610 (40%)
	egit	4513 (48%)	2009-09-29	4270 (86%)	3655 (81%)	2200 (77%)	1624 (36%)	1850 (41%)
	jgit	3984 (36%)	2009-09-29	3736 (84%)	3187 (80%)	513 (41%)	598 (15%)	756 (19%)
	linuxtools	9712 (41%)	2007-02-02	3925 (86%)	3302 (34%)	1938 (56%)	485 (5%)	1456 (15%)
	scout.rt	3299 (39%)	2010-11-25	1668 (59%)	989 (30%)	1350 (80%)	825 (25%)	2045 (62%)

B. Linking Commits to Bug Reports and Reviews

As mentioned in the previous section, by using Barahona et al.'s exposed databases we got access to the git version control, issue report and Gerrit review data of OpenStack and Eclipse. Then, we used heuristics to identify links between commits and reviews, and also between commits and issue reports.

In particular, links from an accepted review to its corresponding git commit can be identified by searching the Gerrit reviews for the commit identifier of the accepted revision of a patch. These commit identifiers had not been stored in Barahona's exposed databases, hence we modified Metrics-Grimoire to download this additional information from the review repository, then updated Barahona's database with the extracted commit identifiers.

However, only for 60% of the accepted reviews, the commit identifier mentioned in the code review corresponds to the actual commit in the official git repository of a project. The reason is that while a code change is being reviewed, other developers' code changes that are being reviewed in parallel, get accepted and entered in the version control system. Hence, by the time the former code change is accepted, it first needs to be integrated with these newly accepted commits ("rebasing" [6]). As a result, a new git commit with new identifier is created that replaces the accepted code change. Since older versions of Gerrit did not update afterwards the code review with the new identifier, those reviews could not be directly linked with the version control system.

Fortunately, for more than 50% of the missing cases, we either found an additional review comment that has been added after rebasing mentioning the correct Git commit identifier, or we found the identifier of the review in the commit message of a git commit. Hence, using these 3 different techniques, we were able to retrieve the links in most of the Eclipse and OpenStack projects for at least 70% of the reviews.

In order to link commits to issue reports, we noticed after manual inspection of commit messages that Eclipse developers almost consistently mention the issue identifier in their commit message, following these regular expressions (for Openstack, we changed this format a bit, as issue numbers for Openstack just have 6 or 7 digits):

```
(bug|issue)[:#\s_]*[0-9]+
(b=|#)[0-9]+
[0-9]+\b
\b[0-9]+
```

However, after checking whether the identified numbers correspond to actual issue identifiers, just 2% of related issues were found in this way. Instead, 35% of the links were found through issue report comments that mentioned a Git commit identifier, and the remaining links between issues and commits were identified through review info (since we had found sufficient links between reviews and commits). To detect the links between issues and reviews, we referred either to issue identifiers mentioned by review comments or the name of the branch on which a code change had been made, since some of them follow the naming convention "bug/1491511" with "1491511" an issue identifier. On average, more than 76%

of the links (72% for Openstack and 81% for Eclipse) with issues to their commits were identified. To evaluate whether the identified links were correct, we manually examined a random number of them.

After linking commits to reviews and issues, and limiting the analyzed time period to those commits made starting from the time when Gerrit was introduced in a project, we eventually retained those projects with more than 3,000 commits, a reasonable amount of linkage (more than 500 links) and defect-prone commits (more than 10%). Out of 337 Openstack projects, 11 projects with good linkage and sufficient defect-prone commits were selected. For Eclipse, 10 projects out of 612 had more than 3000 commits and more than 1000 reviews, however most of them did not have enough issues. For example, *platform.ui* has more than 17,000 fixed issues, but most of them were reported before the project started using Gerrit (April 2013), and only 500 issues were reported afterwards. Therefore, we eventually selected only 5 Eclipse projects. Table II shows the statistics of our data set.

C. Identifying Defect-introducing Commits

Identifying which commits introduce a defect is not straightforward. Various algorithms and heuristics exist, with the SZZ algorithm [39] still one of the most popular approaches. It automatically locates defect-introducing patches by linking information from a version control system like Git to a bug repository like Launchpad or Bugzilla. It consists of three steps: 1) finding all bug fix commits by identifying for each issue report linked to a commit whether it is a bug, in which case the commit likely is a bug fix; 2) identifying, using a standard command like "git blame" the most recent commits that changed the lines changed by the bug fix; and 3) tagging those commits as potentially defect-introducing.

As such, SZZ assumes that a bug fix only changes the source code lines that had a defect and that the most recent commit(s) that changed those lines were the defect-introducing commit(s). Typically, all commits pointed out by "git blame" are considered to be defect-introducing commits, although heuristics exist to filter out commits that appeared after the bug report was filed. We used the SZZ implementation of Kamei et al. [21] to identify the defect-introducing commits in our case study systems. We customized this implementation to use the mapping from commits to issues obtained in Section III-B instead of a keyword-only approach.

D. Discussion Metrics

In this section, we describe the issue and review discussion metrics that we use in our statistical models, complementing the change-level metrics of Table I. The issue discussion metrics are mined from issue repository comments, hence we calculate them on a per-issue report (and hence per-commit) basis[4]. The review discussion metrics are mined from the Gerrit reviews and comments, and hence are calculated on a per-review basis. We grouped all discussion metrics into four categories, as Table III shows.

a) Focus: During review, reviewers can ask a developer to incorporate certain changes. If the developer agrees, these changes will result in a new revision of the patch that will undergo another round of reviewing by the same reviewers. When reviewers eventually are happy with a revision, that revision will be accepted for integration into the version control system (possibly with rebasing). The more revisions a patch had to go through before acceptance, the riskier, as it could indicate issues with a developer's mastery of or familiarity with the problem at hand, or could mean that the bug fix or feature being implemented is complicated. On the other hand, it could also indicate that the reviewers are serious about their work, as they are pointing out many problems.

Furthermore, Bacchelli et al. [1] and McIntosh et al. [26] found that modern reviewing techniques, such as those supported by environments like Gerrit, do not imply high quality reviews. Indeed, one can "review" a patch by just pointing out typos or without checking how the changed lines integrate with existing code. More experienced reviewers or issue commenters might be more aware of these pitfalls and hence perform more focused reviews, reducing the risk of defects. This is also why we measure *#inline comments*, which corresponds to comments annotating specific code lines in a patch. For example, if line 12 of a patch should be improved, a reviewer could put a comment on that specific line, instead of posting a global review comment. More such comments again means more focused reviewing.

b) Length: Complex issues and patches can be more controversial and hence require more discussion than simple ones. We adopt this idea in our models by measuring the *#comments* and *comment length*. Initially, we also used *#authors*, but we found that this was too highly correlated with *#comments*, which is an easier metric to measure. Note that one cannot blindly count all comments, since some review or issue comments are automatically generated messages that do not constitute human discussion, and hence should be filtered.

c) Time: Through issue and review comments, people discuss, share their ideas and help each other regarding the issue or patch under consideration. Hence, the absence of communication for a prolonged time may indicate miscommunication or even indifference, which is a sign of risk. Hence, we capture these ideas by measuring the *fix time* and *average discussion lag* for issue reports and reviews. The former spans from creation date until the final comment, while the average discussion lag is the average of the time period between each subsequent pair of comments.

d) Sentiment: Until now, all selected metrics focused on quantitative aspects of the discussion, in the sense that they count a volume or measure time. However, issue report and review comments especially contain natural language content capturing the direct opinion of developers or other stakeholders on an issue or patch. Whenever people communicate, their words automatically incorporate certain feelings or sentiment (attitude of towards a subject [27]) to convey their message

[4]For commits linked to multiple issue reports, we randomly select one of the issue reports.

TABLE III: Summary of issue and review discussion measures.

Dimension	Domain	Name	Rationale
Focus	both	commenter experience	The more experienced in leaving comments (for both reviews and issues), the more participative and helpful discussions could be, reducing the risk of defects slipping through.
	issue	reporter experience	The more experienced in issue reporting, the more accurate their reports could be, the more precise the issue could be solved.
	review	reviewer experience	The more experienced in reviewing, the more accurate they are expected to be, hence the higher the chance that any serious risk has been identified and remedied.
		#patch revisions	The more revisions a patch required, the more issues were detected and hence could still linger in the final patch revision.
		#inline comments	The number of comments reviewers made on specific lines of a patch instead of general reviews, as an indicator of the degree of detail of reviewing. The more detail, the lower the risk of remaining defects.
Length	both	#comments	The more comments are posted in a discussion, the more risk might be involved.
		comment length	The number of lines of comments on an issue or review, as a measure of the amount of discussion, may indicate that the discussed commit has a high likelihood of introducing a defect.
Time	review	review time	The total time spent on reviewing, might be related to the risk and defect-proneness of the issue.
	issue	fix time	The total time allocated to fix an issue might be related to the risk and defect-proneness of the issue.
	both	average discussion lag	The average time in between comments could be related to the risk of an issue or review, with risky ones seeing faster replies to comments.
Sentiment	both	Comment Sentiment (max/min/avg/extreme)	Negative sentiment of the participants during issue or review discussions may reveal doubt. Four variations of the sentiment metrics are provided.

or understand other people's reaction. For example, one developer or user might say "We are happy with a simple round robin distribution for new request", or "I am absolutely furious that the application did not say almost anything in its logs". Sentiments are not limited to verbal communication, but are also expressed when using computer-aided communication [41]. Hence, the usage of friendly, positive words is a good indicator that a person is happy with a certain issue or patch, while angry expressions could indicate difficult interactions and stress that might reflect in the quality of the resulting work.

To capture such qualitative opinions, we extract and quantify the sentiments expressed by issue and review comments. In sentiment analysis [35], the "polarity", i.e., positive or negative attitude, and "degree" of a document are measured quantitatively. A larger degree of sentiment represents more positive (or negative) sentiments and attitudes towards a subject, topic, idea or even a person. Most sentiment mining tools generate polarity and degree per sentence or paragraph. Since we need one sentiment score per issue report or review (not per sentence/paragraph), we are interested in the Max, Min and $Average$ of the individual sentiment scores to obtain one sentiment value. In addition, we also include the $Most\ extreme$ sentiment value, which is the sentiment value with the largest absolute value across all sentences/paragraphs of an issue report or review.

In practice, we first filter out any extra data other than human natural language, as comments often contain different kinds of information like source code snippets and stack traces. In our case, a lightweight method based on regular expressions turned out to be the most effective filtering approach [2]. We then applied the SentiStrength tool on the pre-processed comments to obtain sentiment scores from -5 to 5 for each paragraph. SentiStrength is one of the state-of-the-art lexical sentiment mining tool [43], which is easy to use and has been used successfully by several research projects [43]. Alternatively, one could use machine learning-based sentiment mining [40] or, instead of sentiment, one could also measure other types of affect like emotions or polarity [34], however those tools are still in an early stage.

E. Building Explanatory Models

Similar to previous work [3], [9], [21], a logistic regression model is used to build an explanatory model of defect-prone code changes. For each commit, such a classification model returns a probability between 0 and 1 of defect-proneness. Based on a threshold value, one can then classify a commit as defect-prone (probability higher than threshold) or safe (probability lower than threshold). This paper uses the model building scripts of Kamei et al. [21], with a standard threshold of 0.5.

We use logistic regression to build explanatory models, i.e., models that explain defect-proneness of the data set on which they are trained rather than predict defect-proneness of a different data set (like on commits of a subsequent year). The evaluation of such models typically is conducted using 10-fold cross validation [11]. According to this technique, the dataset is randomly divided into 10 folds based on stratified partitioning such that each fold has the same proportion of defect-prone commits as the full data set. Then, one fold is picked as test data for model validation, while the rest of the data is used as training data to build a model. This process is repeated for each of the 10 folds as test data, yielding a confusion matrix, based on which performance measures can be calculated.

Before training a model, we first perform three filtering steps. After collecting the required metrics, we first removed highly correlated factors. For this, we computed the variance inflation factors (VIF) for each metric and removed those metrics with variance inflation factor greater than 5. Then, we also applied Mallow's Cp criterion [25] using a stepwise variable selection technique to remove the rest of the collinear metrics and those that do not affect the model. Step by step, this technique removes the worst metrics, i.e., metrics without any effect, until the deletion of the remaining metrics starts degrading the model.

Second, as Table II shows, our dataset is not balanced: the number of defect-introducing commits is much lower than the number of safe commits. According to Kamei et al. [20], this degrades the performance of statistical prediction models. To avoid this problem, we resample the training set data, similar to previous work [21]. With this approach, non-defect-introducing commits randomly are removed from the training set until we have the same number of defect-introducing and safe commits. Note that we cannot resample the test set, as this would bias the evaluation results (since real-life data is imbalanced, and hence we should evaluate our models as such).

Third, similar to Kamei et al. [21], we applied a standard log transformation to each metric with positive skew (i.e., having a long tail of values towards higher values), to even out the skewing effects on the model.

F. Validation of Model Performance

We validate both the models themselves as well as the importance (impact) of the metrics for the models.

Evaluating the Models: To evaluate the defect-prone classifiers, we use the common *precision*, *recall*, *F1-measure* and *AUC* measures. The first three metrics are derived from a model's confusion matrix. Such a matrix summarizes the four different cases of a classification: a model can correctly classify a commit to be defect-prone (TP, true positive) or safe (TN, true negative), incorrectly classify a defect-prone commit to be safe (FN, false negative) or a safe commit to be defect-prone (FP, false positive).

Using these concepts, *precision* then is the percentage of commits classified as defect-prone by a model that is actually defect-prone, i.e., $\frac{TP}{TP+FP}$. This gives an indication of how often a model's recommendation is correct (lack of false alarms). On the other hand, *recall* measures what percentage of the actual defect-prone commits in the data that can be found by the model, i.e., $\frac{TP}{TP+FN}$. Since there is a tradeoff between precision and recall, the *F1-measure* usually is computed, which is a weighted average of precision and recall: $\frac{(2\times Recall\times Precision)}{(Recall+Precision)}$.

Since the output of a logistic regression model is a probability value between 0 and 1, a threshold needs to be chosen to map the model value to 0 or 1. This means that a model's performance depends on the choice of this threshold. The AUC (Area Under the Curve) of the ROC (Receiver Operating Characteristics) curve is a measure that represents the overall performance of a logistic regression model across all thresholds [24]. The larger the AUC, the better the classification performance. In particular, the larger the AUC is compared to 0.5, the better the model performs than a random classification model.

Evaluating the Metrics: To evaluate which metrics have the largest impact in the models, we compare our models to the code change-based baseline models incorporating the metrics of Table I. For this comparison, we use a hierarchical analysis starting with the baseline model that uses software change measures only. Then, step by step, each dimension of issue or review discussion metrics is added to the model to study its importance in the model. To determine if a dimension provides new knowledge about defect-prone commits, we use an ANOVA test to compare a regression model with the previous one. Given an α value of 0.05, a $p-value$ lower than 0.05 rejects the null hypothesis that there is no significant difference in fit between both models, or in other words a $p-value < 0.05$ means that the new model performed significantly better than the old one. In contrast to the classifier performance using precision and other metrics, for model comparison we use the entire dataset to build one logistic regression model in each step.

After determining the issue or review discussion dimension with the highest impact, one can then analyze which individual metrics have the highest impact in the model using the effect size of Shihab et al. [38]. This approach first takes the full logistic regression model with the median value of each metric as input (the mode value for categorical variables like Change_Type). The model's output value for these inputs is recorded as the "base value". Then, in order to find the impact of each variable on the model, we replace for one metric at a time the median value by the median plus one standard deviation (for categorical values, the second most common value will be used to replace the mode), the rest of the variables stay on their median value. The effect size of this metric on defect-proneness can then be calculated as $\frac{newvalue-basevalue}{basevalue}$. We repeated this method for all significant variables to find their relative impact.

Since the median values represent "common" values for each metric, and adding a standard deviation is a "common" change in metric value, the effect sizes of different metrics can be compared directly to each other in order to find the metrics with the largest impact on defect-proneness. In particular, the effect sizes are independent of the unit of the metrics, in contrast to for example odds ratios [7]. A positive effect size indicates an increase in defect-proneness for an increase in independent variable, while a negative effect size indicates a decrease in defect-proneness.

IV. Case Study Results

RQ1. How well can issue discussion metrics explain defect-introducing changes?

Approach. To understand how much information issue report discussions contain about the risk of their corresponding code change, we build explanatory models for the risk of a code change using the issue discussion metrics of Table III, combined with the software change metrics of Table I. Table IV compares for all analyzed projects the performance metrics of the baseline model (only code change metrics) and the issue metrics model.

To identify the most important metrics in the models of each project, we calculated for all metrics the effect size, as explained in subsection III-F. Then, for each metric, we calculated the median effect size across all Openstack projects (and, separately, across all Eclipse projects) in order to globally rank the metrics from most extreme effect size (either positive or

TABLE VII: Metrics with most extreme effect size (Openstack). Issue metrics are underlined, review metrics in bold.

RQ1 Models		RQ2 Models		RQ3 Models	
Metric (Dimension)	Effect	Metric (Dimension)	Effect	Metric (Dimension)	Effect
FIX (Purpose)	-0.9	LA (Size)	0.21	FIX (Purpose)	-0.73
LA (Size)	0.16	**Reviewer Experience (Focus)**	-0.20	**Review Time (Time)**	0.11
Commenter Experience (Focus)	-0.15	LD (Size)	-0.13	Discussion Lag (Time)	0.08
Comment Length (Length)	-0.10	FIX (Purpose)	-0.12	Sentiment_Avg (Sentiment)	-0.05
Reporter Experience (Focus)	0.08	**Sentiment_Extreme (Sentiment)**	0.08	Sentiment_Extreme (Sentiment)	0.04
Discussion Lag (Time)	0.08	**Discussion Lag (Time)**	-0.05	**Discussion Lag (Time)**	-0.04
SEXP (Experience)	-0.08	NS (Diffusion)	0.05	Commenter Experience (Focus)	-0.02
Number of Comments (Length)	-0.07	**Review Time (Time)**	0.03	AGE (History)	0.01
LD (Size)	-0.05	ENT (Diffusion)	0.03		
ENT (Diffusion)	0.03				
Fix Time (Time)	0.03				
NFC (Diffusion)	0.03				
AGE (History)	0.02				

TABLE VIII: Metrics with most extreme effect size (Eclipse). Issue metrics are underlined, review metrics in bold.

RQ1 Models		RQ2 Models		RQ3 Models	
Metric (Dimension)	Effect	Metric (Dimension)	Effect	Metric (Dimension)	Effect
FIX (Purpose)	-0.9	LA (Size)	0.27	LA (Size)	0.37
LA (Size)	0.39	AGE (History)	0.13	**Review Time (Time)**	0.14
NDEV (History)	-0.31	**ReviewTime (Time)**	0.12	AGE (History)	-0.11
AGE (History)	-0.11	ENT (Diffusion)	0.11	ENT (Diffusion)	0.08
Sentiment_Min (Sentiment)	0.08	**Reviewer Experience (Focus)**	-0.10	**Reviewer Experience (Focus)**	-0.07
ENT (Diffusion)	0.03	LT (Size)	0.07	**Discussion Lag (Time)**	-0.04
		LD (size)	0.04		

TABLE IV: Performance of explanatory models with issue metrics. ^{b} means "base model", increases of $\geq 3\%$ in bold, and decreases of $\leq -3\%$ underlined.

project	prec^b	rec.^{b}	f^b	auc^b	prec	rec.	f	auc
cinder	0.82	0.49	0.62	0.53	0.83	**0.59**	**0.69**	**0.56**
devstack	0.41	0.64	0.50	0.63	0.39	0.66	0.49	0.62
glance	0.64	0.60	0.62	0.57	**0.67**	0.61	0.64	**0.60**
heat	0.73	0.54	0.62	0.58	0.75	**0.61**	**0.67**	**0.61**
keystone	0.61	0.68	0.65	0.70	0.59	0.68	0.64	0.69
neutron	0.65	0.73	0.69	0.71	0.64	0.74	0.69	0.70
nova	0.75	0.64	0.69	0.59	0.77	0.61	0.68	0.61
...-manuals	0.81	0.55	0.66	0.57	0.82	0.57	0.67	0.59
swift	0.46	0.67	0.54	0.64	0.45	0.66	0.54	0.63
tempest	0.58	0.56	0.57	0.53	**0.67**	0.58	**0.62**	**0.61**
cdt	0.73	0.70	0.72	0.72	0.73	0.70	0.72	0.71
egit	0.86	0.67	0.75	0.71	0.85	0.66	0.74	0.71
jgit	0.70	0.65	0.67	0.67	0.72	**0.68**	**0.70**	0.69
linuxtools	0.83	0.61	0.70	0.69	0.84	0.61	0.70	0.70
scout.rt	0.67	0.70	0.68	0.70	0.67	0.71	0.69	0.70

TABLE V: Performance of explanatory models with review metrics. ^{b} means "base model", increases of $\geq 3\%$ in bold, and decreases of $\leq -3\%$ underlined.

project	prec^b	rec.^{b}	f^b	auc^b	prec	rec.	f	auc
cinder	0.66	0.61	0.63	0.60	0.67	0.59	0.63	0.61
devstack	0.40	0.64	0.49	0.67	0.37	**0.69**	0.48	0.66
glance	0.56	0.61	0.58	0.58	0.57	0.59	0.58	0.59
heat	0.48	0.55	0.51	0.58	0.47	**0.60**	0.53	0.58
keystone	0.49	0.61	0.55	0.61	0.46	**0.64**	0.54	0.60
neutron	0.68	0.63	0.68	0.68	**0.72**	**0.71**	**0.71**	**0.71**
nova	0.61	0.65	0.63	0.64	0.61	0.65	0.63	0.64
...-manuals	0.51	0.51	0.51	0.56	**0.56**	**0.59**	**0.57**	**0.61**
swift	0.41	0.60	0.48	0.63	0.42	0.59	0.49	0.63
tempest	0.42	0.59	0.49	0.59	0.40	**0.64**	0.49	0.58
cdt	0.55	0.70	0.62	0.72	0.54	**0.73**	0.62	0.72
egit	0.76	0.65	0.70	0.68	**0.79**	**0.72**	**0.75**	**0.72**
jgit	0.63	0.66	0.64	0.66	**0.66**	**0.73**	**0.69**	**0.70**
linuxtools	0.76	0.67	0.71	0.69	0.76	0.64	0.70	0.69
scout.rt	0.54	0.63	0.58	0.65	0.55	0.65	0.59	0.66

negative) to closest to zero. If a variable did not appear in a project's model, we gave it an effect size of zero for that model. Finally, Table VII and Table VIII only show those metrics with median effect size different from zero, i.e., that appeared in the models of at least 50% of the projects.

Findings.

Five of the projects see an improvement in precision and/or recall of 3% up to 10%. ANOVA analysis showed that models augmented with metrics related to issue discussions significantly improves upon the baseline models (even though for some projects a decrease can be observed). The tempest project sees the largest improvements, with precision increasing by 9% and the AUC by 8% compared to the baseline model. Similarly, cinder and heat improve their recall by 10% and 7%, respectively. glance increases its precision by 3%, while jgit improves its recall by the same amount. On the other hand, we notice that nova loses 3% of precision and recall, respectively.

The most important metrics are FIX, LA, NDEV and Issue Commenter Experience. The type of a change has a huge impact on the models of both Eclipse and OpenStack, twice with an effect size of -0.9. This negative value means that when the type of a commit changes from bug fix to no bug fix, the probability of defect-proneness increases, which might indicate that when developers fix bugs in those projects they pay more attention. The third most important metrics (Issue Commenter Experience and NDEV) also have a negative effect size, which means that the more experience

TABLE VI: Performance of explanatory models with issue *and* review metrics. ^{b} means "base model", increases of $\geq 3\%$ in bold, and decreases of $\leq -3\%$ underlined.

project	prec^b	rec.^{b}	f^b	auc^b	prec	rec.	f	auc
cinder	0.83	0.49	0.62	0.54	**0.86**	**0.57**	**0.69**	**0.60**
devstack	0.41	0.59	0.48	0.64	0.39	0.56	0.46	0.63
glance	0.64	0.58	0.61	0.57	**0.76**	**0.71**	**0.74**	**0.71**
heat	0.72	0.51	0.60	0.56	**0.76**	**0.58**	**0.66**	**0.61**
keystone	0.72	0.61	0.66	0.63	0.73	0.60	0.66	0.64
neutron	0.64	0.74	0.69	0.71	**0.71**	**0.80**	**0.75**	**0.77**
nova	0.75	0.61	0.67	0.58	**0.78**	0.60	0.68	**0.62**
...-manuals	0.81	0.58	0.68	0.55	**0.85**	**0.62**	**0.72**	**0.62**
swift	0.46	0.67	0.55	0.64	**0.49**	0.62	0.55	0.65
tempest	0.60	0.44	0.50	0.53	**0.70**	**0.61**	**0.65**	**0.63**
cdt	0.59	0.68	0.63	0.70	0.58	0.66	0.62	0.69
egit	0.85	0.70	0.77	0.71	**0.89**	**0.75**	**0.82**	**0.77**
jgit	0.75	0.65	0.70	0.69	0.74	**0.71**	**0.73**	0.70
linuxtools	0.80	0.68	0.73	0.69	0.80	0.67	0.73	0.69
scout.rt	0.59	0.65	0.62	0.66	**0.62**	**0.70**	**0.66**	**0.69**

the issue reporter has in commenting on issues (OpenStack) or the more developers have changed the modified files before (Eclipse), the lower the risk that the discussed code change will be defect-introducing. This seems intuitive for OpenStack, whereas for Eclipse the finding seems contradictory. The only metric in the top three with a positive effect size, is LA. Hence, the larger a code change is in terms of added lines of code, the higher the probability that it will be defect-introducing.

Models augmented with issue discussion metrics improve upon code-change baseline models. For 5 out of 15 projects, precision and/or recall improve 3% up to 10%.

RQ2. How well can review discussion metrics explain defect-introducing changes?

Approach. We follow a similar approach as for RQ1, but this time use the review metrics of Table III instead of the issue metrics. The model performance metrics are shown in Table V, while the most important metrics are shown in the second column of Table VII and Table VIII.

Findings.

9 projects improve their precision and/or recall by 3% to 8%. Except for heat, tempest and jgit, 6 of these 9 projects did not see an improvement for the issue metric models of RQ1, i.e., the improvements of review-based models seem complementary to those of issue-based models. The largest improvements can be found in terms of recall, with jgit, egit and openstack-manuals improving their recall by 6%, 7% and 8% respectively. horizon, despite its increase in precision by 4%, saw a drop in recall by 8%. Three other projects (devstack, keystone and linuxtools) saw a drop in precision or recall of 3%.

The most important variables are LA/LD, Reviewer Experience, AGE, FIX and Review Time. While LA again scores high (with positive effect size), LD also was found to be important (third place for OpenStack), but with a negative effect size. Hence, while code changes that add many lines of code are more risky, so are code changes that do not remove too much code. Reviewer Experience, as expected, also has a negative impact (i.e., less risk with more experienced reviewers), similar to FIX (cf. RQ1). Eclipse has two other top metrics with positive effect size, i.e., the more time since the last change of the modified files or the longer reviewing takes, the higher the risk of a code change.

Review metrics also improve upon code change-based models, with 9 projects improving their precision and/or recall by 3% to 8%.

RQ3. How well can issue and review discussion metrics explain defect-introducing changes?

Approach. To understand which of the issue and review models has the strongest link with defect-proneness of commits, RQ3 first builds the models of RQ1, then hierarchically adds the metrics of the review dimensions. The performance of the resulting models is shown in Table VI, while the most important metrics are listed in the third column of Table VII and Table VIII.

Findings. 8 of the projects improve both precision and recall by 3% up to 17%, while 2 more projects improve either precision or recall. Compared to RQ1 and RQ2, we observe large improvements in precision of 12% (glance), 7% (neutron) and 10% (tempest), while for recall we observe improvements of 8% (cinder), 13% (glance) and 17% (tempest). Other improvements are from 3% to 5%. This results in increases in AUC of up to 6%, 7% or even 10% (tempest). In other words, the combination of issue report and review discussions metrics seems to have a major link with defect-proneness. That said, devstack and swift see a drop in recall of 3% and 5%, respectively.

The most important metrics overall are FIX, LA, Review Time, AGE and Issue Discussion Lag. Similar to the issue metric-based models of RQ1, FIX again has a large, negative effect size (but did not appear in the Eclipse models) and LA a large, positive effect. Hence, both top metrics are code-level metrics. The second (both projects) and third (for OpenStack) highest metrics, Review Time and Issue Discussion Lag, mostly have a positive effect size, indicating that the longer reviews have taken or the longer it took people to reply to each other's issue comments, the higher the risk of a code change. Both seem intuitive. In particular, a longer lag between issue discussions could be due to the complexity of the problem, or unavailable project members (too busy). Finally, AGE is the final top metric, this time surprisingly with a negative effect size, i.e., the longer since the last change to a file, the lower the risk of defects. We suspect this difference with the RQ2 model for Eclipse is either due to our calculation of median effect sizes or interaction with other metrics.

We note that the top 3 metrics in Table VII and Table VIII contain 3 code change metrics, 1 issue metric and 1 review metric. This indicates that the performance of the final model

depends on all metrics together, with different metrics of the three studied domains having an important link with defect-proneness of code changes. Given that the performance of the explanatory models also improved substantially by adding both review and issue discussion metrics, this suggests that one should consider adding their top metrics to any JIT defect model.

Combined models substantially improve precision and recall for half of the projects, with the most important metrics related to the type/size of a change, the time taken to review a change, the time since the last change and issue discussion lag.

V. Discussion

From the metrics related to the Focus dimension, especially the experience-based metrics turned out to have a major link with defect-proneness, either in the review (Reviewer Experience) or the issue (Issue Commenter Experience) domain. Their negative effect sizes indicate that the more experienced the person, the less risky the commit might be. These findings seem consistent with the results of Kononenko et al. [23]. They also found reviewer experience to be a good indicator of review quality, with less experienced reviewers more likely to overlook potential problems.

Sentiment-related metrics seem to play a smaller role than experience, but still feature in more than half of the models of OpenStack and/or Eclipse. Minimum and extreme sentiment metrics tend to have a maximum effect size of 0.08. Given that the minimum sentiment is a negative metric, a positive effect size actually means that more negative sentiment has an increasing effect on defect-proneness. Extreme sentiment is harder to interpret, since it could be a very negative value or very positive value. Average sentiment (effect size of -0.05) tends to be positive, which again implies that higher sentiment is linked with lower defect-proneness. This seems intuitive, and somehow relates to the findings of Ortu et al. [33] that more positive emotions are linked with shorter issue fixing time, however, more research is needed to fully understand these observations.

We have not seen a significant impact between software quality and the number of comments (Length dimension), except for RQ1 (Number of Comments and Comment Length had negative effect size). These results support the observations of Bavota et al. [4], as they also were not able to find any link between the number of comments and the chance of defects slipping through review.

VI. Threats to Validity

Threats to internal validity concern confounding factors that might influence the obtained results. Although we studied both change measures and human discussion measures, there are likely other unknown factors that impact defect-inducing probabilities that we have not measured yet.

Due to the elaborate filtering that we performed in order to link three repositories (version control, bug repository, and code review) , we finally selected 15 huge projects from OpenStack and Eclipse projects, while for example Eclipse has around 549 projects. While just 10 Eclipse projects have more than 1,000 merged reviews in Gerrit, there were not enough links between issue reports and commits to use the projects for RQ1 and RQ3 (yielding 5 analyzed Eclipse projects in Table II).

Construct validity considers the accuracy of observation measurements. First of all, the heuristics used to find the links between the three repositories are not 100% accurate, however we used the state-of-the-practice linking algorithms at our disposal. Recent features in Gerrit show that clean traceability between version control and review repositories is now within reach of each project, hence the available data for future studies will only grow in volume.

Furthermore, there are some inaccuracies and limitations related to the algorithms and tools that we have used. For example, we used the SZZ algorithm to identify defect-inducing changes. The SZZ algorithm has some limitations, since it looks for special keywords in commit messages to link bug fixing commit and to bug-introducing commits. If a fixing commit message does not contain the keywords used by the algorithm, that commit and its bug-introducing commit will be ignored. Similarly, for sentiment analysis of comments, we used SentiStrength, which is a lexical-based tool that has its own limitations and is not 100% precise. For example, it uses indirect affective terms that can increase inaccuracies. Indirect affective terms such as “feel” or “like” associate with sentiment, but do not directly express it. Hence, their sentiment value depends on the context [42].

Another threat to construct validity is that we assumed that defects had the same weight. This assumption is primarily because of unreliable assigned priorities and severities in issue tracking systems [28], [19]. However, in reality, some defects are more severe and take more time to be resolved. Nonetheless, each defect that we considered was at least severe enough to be fixed and integrated into the system.

Threats to external validity correspond to the generalizability of our experimental results. We studied 15 large open source projects, but we have no evidence to claim that these results are representative of all projects out there. Hence, replication studies should confirm whether our findings generalize to other similar open and closed source projects.

VII. Related Work

There are many studies on defect prediction that use various metrics captured from version control systems and bug databases. Nagappan et al. [32] presented a set of related code churn measures like total lines of code, file churn, file count as highly effective predictors of defect density. Hassan [18] showed in a large case study that change complexity metrics are better predictors in comparison to code complexity metrics. We refer to D’Ambros et al. [10] for a detailed survey and evaluation of these models.

In contrast to prior work, for which the granularity were files or packages, Kamei et al. [21] proposed Just-In-Time(JIT)

defect prediction models that predict "defect-prone" software changes using the 14 factors related to software changes described in subsection II-A. Their study was conducted over six open source and five commercial projects and the proposed change risk model has an accuracy of 68 percent while it also may reduce the effort in resolving the most risky changes. Later, Fukushima et al. [12], through a case study on 11 open source projects, evaluated the performance of the JIT work in a cross-project context. They showed that strong within-project performance of a JIT model does not imply it also will perform well in a cross-project context. These studies build on earlier work by Kim et al. [22] and Mockus et al. [29].

In our study, we focused on the impact of human review and issue discussions on software quality. There are several other studies that investigated the impact of human factors on software quality. Here we briefly mention the most relevant ones for our work.

Wolf et al. [45] conducted a study of the relation between communication, coordination and software quality integration. They used the IBM Jazz repository to investigate the relationship between communication structures and instances of coordination during code integration. By applying social network analysis metrics, measurable communication characteristics are computed. Results show that developer communication plays an important role in the quality of software integrations.

Bettenburg et al. [5] investigated how social interaction measures affect software quality in the form of post-release defects. They discovered that a consistent information flow in discussions decreases the probability of defects significantly and consequently increases software quality. Sliwerski et. al [39] presented the impact of work dependencies on software quality.

Graziotin et al. [15] conducted a study with 42 student participants and investigated the correlation of affect with creativity and analytical problem solving performance of software developers. Their results showed that happier software developers are more productive in problem solving performance. Murgia et al. [31] found that software artifacts like issue repository comments can convey emotional information of developers. In this research, we use the sentiment of repository comments as one possible metric of emotions in issue or review discussions.

Tourani et al. [44] studied the usage of automatic sentiment analysis on open source mailing lists and showed that development mailing lists carry both positive and negative sentiments. Guzman et al. also applied sentiment analysis in their work [16] to analyze sentiment in topics extracted from software collaboration artefacts. In contrast to these studies, the work presented in this paper does not focus on sentiments of emails or comments, but rather on investigating the link between sentiment and defect-proneness. Lately, Ortu et al. [33] studied the impact of human emotions, sentiment and politeness on issue fixing time and found that extremely positive or negative issue comments are linked with lower issue fixing time. Hence, understanding the way in which human activities taking place before a code change relate to the defect-proneness (and hence quality) of that change, is important.

Bavota et al. [4] empirically showed on three large systems that code reviewing significantly decreases the chance of defect introduction, while it also substantially increases the readability of the reviewed code. Similar to us, they first had to link commits to reviews to determine which commits were reviewed and which ones not. They used this data to compare defect-proneness of reviewed commits to non-reviewed commits. They also analyzed whether defect-proneness changes have lower review participation degree (i.e., number of reviewers and review comments).

Kononenko et al. [23] performed an empirical study on four large systems to quantitatively investigate factors that might affect code review quality. They analyzed technical properties of contributions, personal characteristics of developers, and some attributes of the team involved in review process. Finally, they found that some technical factors and developer-related factors like review experience and review load can influence code review quality. Note that the studied systems used the Bugzilla issue repository technology for reviewing, which means that some of the review metrics were impossible for us to obtain from the Gerrit review technology used by OpenStack and Eclipse.

Instead of explicitly evaluating the quality of reviews, we focus only on commits that are linked to an issue report and/or review in order to build full-fledged models with 14 code change, 12 review and 10 issue metrics to understand the link between human discussions and defect-proneness of changes. Furthermore, we studied 15 large systems of 2 major ecosystems.

VIII. Conclusion

In this paper, we empirically studied the impact of human discussion metrics on "Just-In-Time" defect prediction models through an extensive study on 15 large open source projects. We categorized the identified human discussion metrics into four dimensions and measured them on both the issue report and/or review linked to a code change. Explanatory models built using these metrics show a strong connection between human discussion metrics and defect-prone commits, complementing traditional change-based metrics. In particular, while code change metrics related to the size, type (bug fix or not) and time since last change are the most impactful metrics, review time and issue discussion lag have shown to have a substantial positive (i.e., increasing) effect on defect-proneness. Some models also included the experience of issue reports and reviewers as important metrics, which have a negative (i.e., decreasing) effect on defect-proneness. We believe that our study opens up a variety of research opportunities to continue investigating the impact of collaborative characteristics on quality assurance.

References

[1] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proc. of the 2013 Intl. Conf. on Software Engineering (ICSE)*, pages 712–721, 2013.

[2] A. Bacchelli, M. Lanza, and R. Robbes. Linking e-mails and source code artifacts. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 375–384, New York, NY, USA, 2010. ACM.

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

[4] G. Bavota and B. Russo. Four eyes are better than two: On the impact of code reviews on software quality. In *Proc. ICSME*, pages 81–90. IEEE, 2015.

[5] N. Bettenburg and A. E. Hassan. Studying the impact of social interactions on software quality. *Empirical Softw. Engg.*, 18(2):375–431, Apr. 2013.

[6] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining git. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories (MSR)*, pages 1–10, 2009.

[7] J. M. Bland and D. G. Altman. Transformations, means, and confidence intervals. *BMJ*, 312(7038):1079, Apr. 1996.

[8] B. W. Boehm and P. N. Papaccio. Understanding and controlling software costs. *IEEE Trans. Softw. Eng.*, 14(10):1462–1477, Oct. 1988.

[9] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. *IEEE Trans. Softw. Eng.*, 35(6):864–878, Nov. 2009.

[10] M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Softw. Engg.*, 17(4-5):531–577, Aug. 2012.

[11] B. Efron and R. Tibshirani. Cross-validation and the bootstrap: Estimating the error rate of a prediction rule. Technical report, Stanford University, 1995.

[12] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi. An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 172–181, New York, NY, USA, 2014. ACM.

[13] D. Galin. *Software Quality Assurance: From Theory to Implementation*. Pearson, June 2003.

[14] J. M. Gonzalez-Barahona, G. Robles, and D. Izquierdo-Cortazar. The metricsgrimoire database collection. In *12th Working Conference on Mining Software Repositories (MSR)*, pages 478–481, May 2015.

[15] D. Graziotin, X. Wang, and P. Abrahamsson. Happy software developers solve problems better: psychological measurements in empirical software engineering. *PeerJ*, page e289, 3 2014.

[16] E. Guzman, D. Azócar, and Y. Li. Sentiment analysis of commit comments in github: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 352–355, New York, NY, USA, 2014. ACM.

[17] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Softw. Eng.*, 31(10):897–910, Oct. 2005.

[18] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88, Washington, DC, USA, 2009. IEEE Computer Society.

[19] I. Herraiz, D. M. German, J. M. Gonzalez-Barahona, and G. Robles. Towards a simplification of the bug report form in eclipse. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 145–148, New York, NY, USA, 2008. ACM.

[20] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K. ichi Matsumoto. The effects of over and under sampling on fault-prone module detection. In *ESEM*, pages 196–204. IEEE Computer Society, 2007.

[21] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Trans. Software Eng.*, 39(6):757–773, 2013.

[22] S. Kim, E. J. Whitehead, Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Softw. Eng.*, 34(2):181–196, Mar. 2008.

[23] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey. Investigating code review quality: Do people and participation matter? In *Proc. ICSME*, pages 111–120. IEEE, 2015.

[24] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Trans. Softw. Eng.*, 34(4):485–496, July 2008.

[25] C. L. Mallows. Some comments on Cp. *Technometrics*, 15:661–675, 1973.

[26] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proc. of the 11th Working Conf. on Mining Software Repositories (MSR)*, pages 192–201, 2014.

[27] N. Mishra and C. K. Jha. Article: Classification of opinion mining techniques. *International Journal of Computer Applications*, 56(13):1–6, October 2012. Published by Foundation of Computer Science, New York, USA.

[28] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, July 2002.

[29] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5:169–180, 2000.

[30] J. C. Munson and T. M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Trans. Software Eng.*, 18(5):423–433, 1992.

[31] A. Murgia, P. Tourani, B. Adams, and M. Ortu. Do developers feel emotions? an exploratory analysis of emotions in software artifacts. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 262–271, New York, NY, USA, 2014. ACM.

[32] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 284–292, New York, NY, USA, 2005. ACM.

[33] M. Ortu, B. Adams, G. Destefanis, P. Tourani, M. Marchesi, and R. Tonelli. Are bullies more productive? empirical study of affectiveness vs. issue fixing time. In *Proceedings of the 12th IEEE Working Conference on Mining Software Repositories (MSR)*, Florence, Italy, May 2015.

[34] M. Ortu, G. Destefanis, M. Kassab, S. Counsell, M. Marchesi, and R. Tonelli. Would you mind fixing this issue? an empirical analysis of politeness and attractiveness in software developed using agile boards. In *XP2015, Helnsiki*, page in press. Springer, 2015.

[35] B. Pang and L. Lee. Opinion Mining and Sentiment Analysis. *Foundations and Trends in Information Retrieval*, 2(1-2):1–135, Jan. 2008.

[36] P. C. Rigby and C. Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 202–212, New York, NY, USA, 2013. ACM.

[37] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 62:1–62:11, New York, NY, USA, 2012. ACM.

[38] E. Shihab, Y. Kamei, B. Adams, and A. E. Hassan. Is lines of code a good measure of effort in effort-aware models? *Information and Software Technology*, 55(11):1981–1993, 2013.

[39] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM.

[40] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Y. Ng, and C. Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1631–1642, Stroudsburg, PA, October 2013. Association for Computational Linguistics.

[41] M. Thelwall. Heart and soul: Sentiment strength detection in the social web with sentistrength (summary book chapter). In press.

[42] M. Thelwall, K. Buckley, and G. Paltoglou. Sentiment strength detection for the social web. *J. Am. Soc. Inf. Sci. Technol.*, 63(1):163–173, Jan. 2012.

[43] M. Thelwall, K. Buckley, G. Paltoglou, D. Cai, and A. Kappas. Sentiment in short strength detection informal text. *J. Am. Soc. Inf. Sci. Technol.*, 61(12):2544–2558, Dec. 2010.

[44] P. Tourani, Y. Jiang, and B. Adams. Monitoring sentiment in open source mailing lists -– exploratory study on the apache ecosystem. In *Proceedings of the 2014 Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*, Toronto, ON, Canada, November 2014.

[45] T. Wolf, A. Schroter, D. Damian, and T. Nguyen. Predicting build failures using social network analysis on developer communication. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.