Large-Scale Intent Analysis for Identifying Large-Review-Effort Code Changes

Song Wang[a], Chetan Bansal[b], Nachiappan Nagappan[b]

[a] *Electrical Engineering and Computer Science, York University, Canada*
[b] *Microsoft Research, Redmond, WA, USA*

Abstract

Context: Code changes to software occur due to various reasons such as bug fixing, new feature addition, and code refactoring. Change intents have been studied for years to help developers understand the rationale behind code commits. However, in most existing studies, the intent of the change is rarely leveraged to provide more specific, context aware analysis.

Objective: In this paper, we present the first study to leverage change intent to characterize and identify Large-Review-Effort (*LRE*) changes—changes with large review effort.

Method: Specifically, we first propose a feedback-driven and heuristics-based approach to identify change intents of code changes. We then characterize the changes regarding review effort by using various features extracted from change metadata and the change intents. We further explore the feasibility of automatically classifying *LRE* changes. We conduct our study on four large-scale projects, one from `Microsoft` and three are open source projects, i.e., Qt, Android, and OpenStack.

Results: Our results show that, (i) code changes with some intents (i.e., Feature and Refactor) are more likely to be LRE changes, (ii) machine learning based prediction models are applicable for identifying *LRE* changes, and (iii) prediction models built for code changes with some intents achieve better performance

Email addresses: `wangsong@eecs.yorku.ca` (Song Wang), `chetanb@microsoft.com` (Chetan Bansal), `nachin@microsoft.com` (Nachiappan Nagappan)

than prediction models without considering the change intent, the improvement in AUC can be up to 19 percentage points and is 7.4 percentage points on average.

Conclusion: The change intent analysis and its application on LRE identification proposed in this study has already been used in `Microsoft` to provide the review effort and intent information of changes for reviewers to accelerate the review process. To show how to deploy our approaches in real-world practice, we report a case study of developing and deploying the intent analysis system in `Microsoft`. Moreover, we also evaluate the usefulness of our approaches by using a questionnaire survey. The feedback from developers demonstrate its practical value.

Keywords: Change intent analysis, review effort, machine learning

---

1. Introduction

Code changes to software occur due to various reasons such as bug fixing, feature addition, and code refactoring. Often different changes have different motivations on behalf of the developers. For example, it is possible that a code merge, refactoring change, new feature addition all have different risk profiles due to the fact that they are different types of work actions performed by developers. In the past, in the large body of existing work about change analysis [26, 15, 43, 47], changes are all considered to be uniform. It is our goal in the paper to investigate if all changes are equal or if using the change intent for an example scenario of effort prediction, i.e., predicting changes that require large review effort, would result in building better, more accurate context-aware models.

This paper presents the first study to leverage change intent [70, 9] to characterize and understand *Large-Review-Effort (LRE) changes*, i.e., changes that have more than two iterations of code review. Other changes are treated as *regular changes*. In this study, we characterize the uniqueness of the *LRE changes* by using various features collected from the change metadata and the change

intents, and explore the feasibility of automatically identifying the *LRE changes* by building machine learning based classifiers.

First, for understanding the change intent, following existing studies [24, 23, 36, 35], we use heuristics to annotate changes with different change intents by analyzing the commit messages, e.g., changes made for fixing bugs are labelled as 'Bug Fix'. For obtaining accurate change intents, we propose a feedback-driven approach to design and refine the heuristics. Our manual evaluation shows the heuristics for categorizing changes achieve an accuracy higher than 80% on each category. Second, in order to characterize the changes, we have collected various features from the metadata of changes and change intents, such as the process features [86, 26, 85], author information and experience, and the Word2Vec [45, 8] features generated from the commit messages to represent the changes. Third, based on the collected features, we further explore the feasibility of building machine learning based prediction models, i.e., Alternating Decision Tree (ADTree), Logistic Regression (Logistic), Naive Bayes (NB), Support Vector Machine (SVM), and Random Forest (RF), to classify the changes with and without considering the change intents. To evaluate the practical value of our work, we have conducted a case study to discuss the development and deployment of our proposed change intent analysis tool in `Microsoft` and a survey to further evaluate the usefulness and effectiveness of our proposed approach.

This paper makes the following contributions:

- We propose a feedback-driven and heuristics-based approach to classify code changes into different change intents accurately for understanding changes.
- We show that change intents have a strong correlation with the review effort of the changes and changes with some intents (i.e., Feature and Refactor) are more likely to have more review iterations.
- We explore the feasibility of leveraging machine learning models to identify *LRE changes* on one project from `Microsoft` (referred to as `Microsoft`

project) and three open source projects, i.e., Qt, Android, and OpenStack. Experiment results suggest that machine learning based models can be used to identify *LRE changes*. Random Forest, which is the best prediction model in our experiments, achieves AUC values larger than 0.71 on each of the four projects.

- We show that code review effort prediction models built on changes with particular change intents achieve better performance than the general prediction models that do not consider the change intents. **The tool developed in this study has already been used in `Microsoft` to provide the review effort and intent information of changes for reviewers to accelerate the review process.**

- We have conducted a case study to show the deployment of the intent analysis system in `Microsoft`. In addition, a questionnaire survey have been conducted to show the usefulness of our approaches.

- We have released our data and source code to collect features, label changes, and build *LRE* change classification models for facilitating the replication of our study[1].

The rest of this paper is organized as follows. Section 2 presents the background and motivation. Section 3 describes the methodology of our approach. Section 4 shows the experimental setup. Section 5 presents the evaluation results. Section 6 presents our case study on the deployment of the intent analysis system in `Microsoft`. Section 7 discusses open questions and our survey on the usefulness of our proposed approaches. Section 8 shows the threats to the validity of this work. Section 9 presents the related studies. Section 10 concludes this paper. This paper extends our prior publication [84] presented at the 15th International Conference on Predictive Models and Data Analytics in Software

[1] `https://bitbucket.org/wangsonging/ist2020_repo/`

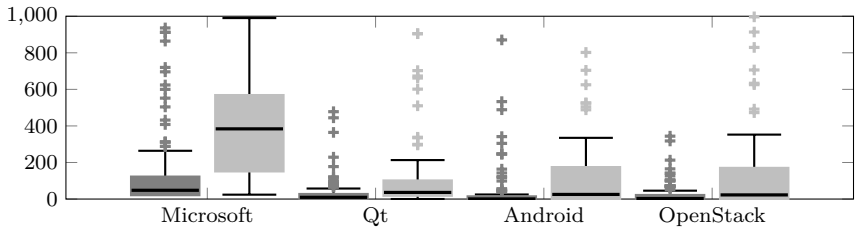


Figure 1: The distributions of review durations (hour) for *regular* and *LRE changes*. Gray bars (●) denote *regular* changes, light gray bars (●) denote *LRE* changes.

Engineering (PROMISE'19). New materials with respect to the conference version include:

- We have conducted a case study to discuss the development and deployment of our proposed change intent analysis tool in `Microsoft`. The tool provides intent labelling service for pull requests with different intent categories for over 200 repositories inside `Microsoft` using the Azure DevOps platform [3] (Section 6). We believe our tool can provide practical guidelines for developing and integrating change intent analysis into real-world software development.

- We have conducted a survey to further evaluate the usefulness and effectiveness of our proposed approach. Feedback shows that more than 90% participants agree with the usefulness of our tool and would like to use it in real practice of code review process (Section 7.4).

- Additional details regarding the experimental design (Section 3), results analysis (Section 5), and related work (Section 9.2) are provided.

2. Background and Motivation

Code changes could be problematic, e.g., they may introduce quality issues such as bugs, improper implementations, and maintenance issues. As a consequence, reviewing them could require much more code review effort. This study focuses on exploring the code review effort of changes. Specifically, based on

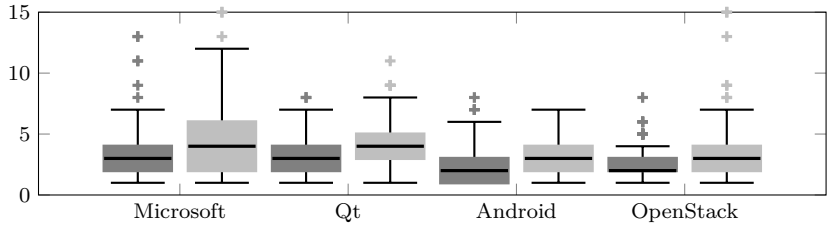


Figure 2: The distributions of #reviewers involved for reviewing changes. Gray bars (●) denote *regular* changes, light gray bars (●) denote *LRE* changes.

the consensus of developers from `Microsoft`, we define a *Large-Review-Effort (LRE) change* as a code change that has more than two iterations of code review, e.g., if a code change cannot pass the first iteration of code review, developers have to conduct a second iteration of code review and even more iterations until they resolve all the review suggestions posted by reviewers. Other changes are treated as *regular changes*.

Code review is a manual inspection of source code by humans, aims at identifying potential defects and quality problems in the source code before its deployment in a live environment [11, 61, 48, 40]. However, such a manual inspection could be a time-consuming and expensive process [75, 61, 48, 40, 27, 62, 79]. For example, to effectively assess a code change, developers are required to read, understand, and critique the code change [62, 79].

In this section, we motivate this study by showing the review effort of *regular* and *LRE* changes, i.e., review duration and the number of reviewers involved. Specifically, for each *regular change* and *LRE change* from the four projects in Table 1, we collect its duration in the code review system and the number of reviewers involved. A reviewer is involved if s/he is in the "Reviewers" field. We use the difference between the submission timestamp and the resolution timestamp of a review request of a change to assess its review duration. Note that we use the review duration to estimate the time effort of reviewing a change, since the exact time cost to review each change is not recorded in the code review systems. We then average the review duration and the number of reviewers involved for all changes for each project.

Table 1: Projects used in this study. **#CR** is the number of changes. **LRERate** is the rate of *LRE changes*.

Project	**Lang**	**First Date**	**Last Date**	**#CR**	**LRERate (%)**
Microsoft	C#	5/05/2015	5/22/2018	>100K	∼15
Qt	C	5/17/2011	5/25/2012	23,041	39.28
Android	JAVA	7/18/2011	5/31/2012	7,120	31.75
OpenStack	Python	7/18/2011	5/31/2012	6,430	43.31

Figure 1 shows the distribution of review duration for each project. As shown in the figure, the average review duration of *LRE changes* could be 10X of that for *regular changes* (in project Android). Figure 2 shows the distribution of the number of reviewers involved to review both the *regular changes* and *LRE changes*. As we can see, on average reviewing the *LRE changes* requires more reviewers to collaborate together than reviewing the *regular changes* in each of the four projects.

We further conduct the Mann-Whitney U test ($p < 0.05$) to compare the differences of review duration and the number of involved between the two groups of changes. The results suggest that the review duration and the number of involved of *LRE changes* are significantly larger than that of *regular changes* respectively. Intuitively, finding *LRE changes* when they are submitted for code review, i.e., pre-merge and pre-deployment, can provide the review effort and intent information of changes for reviewers to accelerate the review process.

3. Approach

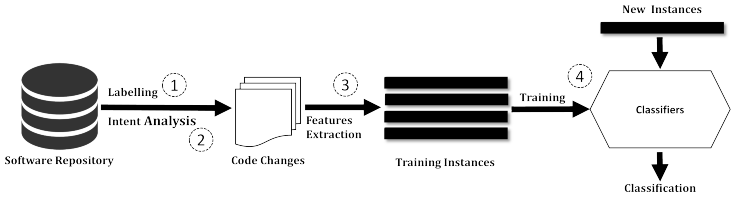


Figure 3: The overview of our *LRE change* prediction approach.

Figure 3 illustrates that our approach consists of four steps: (1) labeling each history change as a *regular change* or a *LRE change* (Section 3.1), (2) analyzing

Table 2: Heuristics for categorizing changes.

Change Intent	**Description**	**Heuristics**
Bug Fix	changes are made to fix bugs	1. the commit message contains keywords: “bug” or “fix” AND 2. the commit message does not contain keywords: “test case” or “unit test”
Resource	changes are made to update non-source code resources, configurations, or documents	1. the commit message contains keywords: “conf” or “license” or “legal” OR 2. if no keyword is matched in step 1, the changed files do not involve any source/test files
Feature	changes are made to implement new or update existing features	1. the commit message contains keywords: “update” or “add” or “new” or “create” or “add” or “implement feature” OR 2. changes in the ‘Other’ category that contain keywords: “enable” or “add” or “update” or “implement” or “improve”
Test	changes are made to add new or update existing test cases	1. the commit message contains the keyword: “test” OR 2. the changed files contain only test files or resource files
Refactor	changes are made to refactor existing code	the commit message contains the keyword: “refactor”
Merge	changes are made to merge branches	the commit message contains keywords: “merge” or “merging” or “integrate” or “integrated” or “integrated”
Deprecate	changes are made to remove deprecated code	the commit message contains keywords: “deprecat” or “delete” or “clean up”
Auto	changes that are committed by automated accounts or bots	the change is submitted by automated accounts or bots
Others	changes that are not in any of the above categories	-

the change intents of all the changes (Section 3.2), (3) extracting features to represent the changes (Section 3.3), and (4) using the features and labels to build and train prediction models and then predicting new changes with the well-trained models (Section 3.4).

3.1. Labeling LRE Changes

The first step of our approach is to label each change as a *regular change* or a *LRE change* based on its code review history. Specifically, for the `Microsoft` project, we extracted its code review database, and checked the code review iteration count for each change, if it has more than two iterations of code review, we labeled it as a *LRE change* (based on the consensus of developers from `Microsoft`) otherwise we labeled it as a *regular change*. For the three open-source projects, since their code review systems do not maintain the code review iteration count, we use a heuristic approach to collect the *regular* and *LRE* changes. Specifically, in their code review system, a code review request of a change may have multiple iterations of code review, for each iteration, developers may submit a patchset to be reviewed, a patchset may have multiple patches. We counted the number of patchset for each code change. If the number of submitted patchsets is larger than two, we labeled the change as *LRE* otherwise we labeled it as *regular*.

3.2. Intent Analysis

Many approaches have been proposed to characterize and classify changes based on the change intents [70, 58, 1, 24]. Most of them consider the high-level change intents, e.g., corrective, adaptive, or perfective [70]. In this study, we leverage the fine-grained change intent categories proposed in Hindle et al. [24], which is shown in its Table 3, to categorize changes. Note that there are more than 20 different categories described in [24]. We started with manual analysis on 200 randomly selected changes from the four projects, and found that some of the categories have very few numbers of changes, e.g., 'Legal', 'Build', 'Branch' have less than three changes. We then group the small categories into larger ones for obtaining more instances. For example, we have grouped 'Legal',

'Data', 'Versioning', 'Platform Specific', and 'Documentation' into '**Resource**', grouped 'Rename' and 'Token Replace' into '**Refactor**', etc. After the grouping, we get eight types of change intents. Note that we also have an '**Other**' category for changes that do not fall into any of the eight categories. In total, we have nine types of change intents.

Table 2 shows the nine types of changes, their descriptions, and the heuristics we used to automatically classify changes. Instead of manually labeling changes, in this work we automate the classification process by using well-refined heuristics. Thus, the accuracy of heuristics could significantly affect the result of this study. To improve the accuracy of the classification of changes, we used a feedback-driven approach to design and refine the heuristics for each type of change intents and the details are as follows:

Step 1: With a randomly selected 200 instances, the first two authors first classify them into the nine categories manually and independently. Specifically, after reading the commit message and checking the changed files of a change, they label the change based on their experience. For the classification conflicts (less than 5%), the third author inspects them independently and the first three authors make a decision for each conflict together. Then they initialize the heuristics for each category. We use $Cohen's Kappa$[2] to measure the inter-coder reliability of Step 1 and the score is 0.91.

Step 2: With the initialized heuristics, we classify all changes into at least one of the nine categories. For each category, we randomly collect 50 instances and the authors work together to check its accuracy manually.

Step 3: If the accuracy of the heuristics designed for a category (on the randomly collected 50 instances from Step 2) is lower than 80%, we further refine the heuristics and then redo **Step 2**, otherwise we keep the heuristics for classifying changes.

Taking the '**Test**' category as an example, in **Step 1**, we found that most changes from it have keywords "test" or "testing" in their commit messages.

[2] https://en.wikipedia.org/wiki/Cohen%27s_kappa

Thus, the initialized heuristics we designed for '**Test**' is that the commit message of a change contains the keywords "test". In **Step 2**, we randomly checked 50 of the collected changes labeled as '**Test**'. Our manual inspection revealed that almost half of them were false positives, and we also found that most of the false positives have irrelevant commit messages, e.g., "... use backup config if test fails ..." and "... send a test message ...". To improve the heuristics, in **Step 3**, we added another heuristic, which is the changed files can only be test files (e.g., file names or paths contain the keyword "test") or resource files. Then we redo **Step 2** again, by using the new heuristics, the accuracy of '**Test**' category is around 90%.

We use the above steps to refine the heuristics of each category to ensure the classification of change intents has higher accuracy. Table 2 shows the final heuristics.

3.3. Feature Extraction

In this study, we use the following features for building machine learning based *LRE change* prediction models.

Change Intent: As code changes could be classified into different categories based on their intents, we assume that changes with different intents have different impacts on the review effort of changes. We use a vector to represent the change intents of a change. Each element in the vector is a binary value, i.e., 1 or 0, representing whether the change has that intent or not. The change intents we considered are listed in Table 2.

Revision History: As presented in previous research [30], the revision history of a file can be a factor to predict its quality. In this study, we also explore the impact of revision history on predicting *LRE changes*. Specifically, given a change, we collect the number of files in this change that have been revised in the last 30 and 90 days, and the number of revision on all the involved files of this change in the last 30 and 90 days.

Owner Experience: This set of features represent the experience of a change's committer. We use a committer's commit history information to represent her/his experience, which includes the total number of changes submitted, the total number of *LRE changes* submitted, and the rate of submitted *LRE changes*. We assume that the committer's experience affects the review effort of the changes s/he submitted.

Word2Vec Features: Word embedding is a feature learning technique in natural language processing where individual words are no longer treated as unique features, but represented as a d-dimensional vector of real numbers that capture their contextual semantic meanings [45]. We train the embedding model by using all data from each project. With the trained word embedding model, each word can be transformed into a d-dimensional vector where d is set to 100 as suggested in previous studies [82]. Meanwhile a code change can be transformed into a matrix in which each row represents a term in its commit message. We then transform the code change matrix into a vector by averaging all the word vectors the code change contains, as described in [82].

Process Features: Various process features have been shown to help predict software bugs [59, 26]. In this study, we use the following process features: code addition, code deletion, number of changed files, and the types of changed files. Note that for the types of changed files, following existing studies [25, 24], we group files into source files, test files, configuration files, scripts, documentations, and others based on their suffixes and file paths. Specifically, we consider files with extensions: .java, .cs, .py, .js, .c, .cpp, .cc, .cp, .cxx, .c++, .h, .hpp, .hh, .hp, .hxx, and .h++, as the source files. Among them, files that contain 'test' in the paths or file names are considered as test files. Files with extensions: .script, .sh, .bash are considered as scripts. Files with extensions: .xml, .conf, .MF are considered as configuration files. Files with extensions: .htm, .html. .css, .txt, are considered as documentation files. The left files are considered as others.

Metadata: In addition to the above features, we also use metadata features of changes. Specifically, given a code change, we collect its commit minute (0,

1, 2, ... , 59), commit hour (0, 1, 2, ... , 23), commit day in a week (Sunday, Monday, ... , Saturday), commit day in a month (0, 1, 2, ... , 30), commit month in a year (0, 1, 2, ... , 11), and source file/path names.

All the features we used are available when the changes are submitted into the code review system.

3.4. Building Models and Predicting LRE Changes

After we obtain the features for changes, we split the data into the training and test datasets. We build and train the machine learning based prediction models on the training dataset and evaluate their performance on the test dataset. Following existing studies [29, 59, 23], we use 10-fold cross-validation to evaluate the prediction models. The process of 10-fold cross-validation is: 1) separating the data set into 10 partitions randomly; 2) using one partition as the test data and the other nine partitions as the training data; 3) repeating step 2) with a different partition as the test data until all data have a predicted label; 4) computing the evaluation results through comparison between the predicted labels and the actual labels of the data. This process helps reduce the bias in the error estimation of classification [28].

4. Experiment Setup

4.1. Research Questions

RQ1: What are the distributions of *LRE* and *regular* changes regarding change intents?

RQ2: Is it feasible to predict *LRE changes* by using machine learning based classifiers with features extracted from changes?

RQ3: Do the specific prediction models (classifiers trained on changes with a particular intent) outperform the general models (classifiers trained on all changes)?

RQ4: Does the performance of predicting *LRE changes* with a single intent differ from that of predicting *LRE changes* with multiple intents?

In RQ1, we aim at understanding the distributions of changes regarding the change intents. In RQ2, we explore the feasibility of predicting *LRE changes*. In RQ3, we aim to explore whether prediction models built on changes with particular intents can generate better performance. In RQ4, we investigate the difference in predicting *LRE changes* with a single intent and changes with multiple intents.

4.2. Experiment Data

To address our research questions, we perform empirical studies on software projects that actively adopt the code review process. We begin with the review dataset of Android, Qt, and OpenStack provided by Hamasaki et al. [18]. The three projects adopt the Gerrit[3] code review system. We also expand the review dataset to include code review data from a large-scale proprietary project from `Microsoft`. It adopts a custom code review system, which shares a similar review process with Gerrit. Details of the projects are in Table 1.

Android[4] is an operating system for mobile devices that is developed by Google. Qt[5] is a cross-platform application and UI framework. OpenStack[6] is an open-source software platform for cloud computing. The last project is a widely used web service from `Microsoft`.

4.3. Evaluation Measures

To measure the performance of predicting *LRE changes*, we use four metrics: *Precision*, *Recall*, $F1$, and AUC. These metrics are widely adopted to evaluate prediction tasks [94, 26, 86, 81, 72, 10, 83]. Here is a brief introduction:

$$Precision = \frac{true\ positive}{true\ positive + false\ positive} \tag{1}$$

$$Recall = \frac{true\ positive}{true\ positive + false\ negative} \tag{2}$$

[3] https://www.gerritcodereview.com/
[4] https://www.android.com/
[5] https://www.qt.io/developers/
[6] https://www.openstack.org/

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \tag{3}$$

Precision and recall are composed of three numbers regarding *true positive*, *false positive*, and *false negative*. True positive is the number of predicted *LRE changes* that are truly *LRE changes*, while false positive is the number of predicted *LRE changes* that are *regular changes*. False negative records the number of predicted *regular changes* that are *LRE changes*. F1 considers both precision and recall.

AUC is the area under the ROC curve, which measures the overall discrimination ability of a classifier. It has been widely used to evaluate classification algorithms in prediction tasks [57, 12, 92, 51, 52]. AUC can evaluates the ability of classifiers in discriminating between defective and clean modules. The AUC score for a perfect model would be 1, for random guessing would be 0.5. A machine learning model is considered applicable to classify a given dataset if the AUC score is larger than 0.7 [74].

5. Results and Analysis

5.1. RQ1: Distribution of Changes

Following the change intent taxonomy approach described in Section 4.1, we automatically label each change from the four projects. As reported in existing studies [46, 13], software changes could be made for multiple purposes, e.g., a change could be made for correcting bugs and refactoring existing code at the same time. Thus, we also label changes with multiple intents. Table 3 shows the number of changes, the percentage of changes with a particular change intent among all changes, and the percentage of *LRE changes* for each change intent in the four projects. In addition, we also show the numbers of changes that have single and multiple intents. Note that the '**Auto**' changes only exist in `Microsoft` project and we find all of them are regular changes, thus we exclude these changes for building change prediction models. Since there exist overlaps

Table 3: Taxonomy of code changes. **#Change** is the number of code changes. **Percent** is the percentage of changes with a particular change intent among all the changes. **LRERate** is the rate of *LRE changes*, which is measured in a percentage. **Single** contains changes that have only one intent. **Multiple** contains changes that have multiple intents. For confidentiality reasons, we did not release the numbers from `Microsoft`.

Change	**Microsoft**		**Qt**			**Android**			**OpenStack**		
Intent	**Percent**	**LRERate**	**#Change**	**Percent**	**LRERate**	**#Change**	**Percent**	**LRERate**	**#Change**	**Percent**	**LRERate**
Bug Fix	∼20	19.11	9,042	39.24	35.43	2,143	30.10	35.88	3,380	52.57	46.95
Resource	∼39	8.98	5,326	23.12	28.56	1,561	21.92	29.66	2,300	35.77	34.26
Feature	∼12	33.55	3,526	15.30	55.05	1,735	24.37	46.63	771	12.00	60.83
Test	∼4	15.14	3,840	16.67	38.75	663	9.31	35.6	1,005	15.63	54.23
Refactor	∼2	41.97	696	3.02	49.28	117	1.64	50.43	195	3.03	65.64
Merge	∼4	14.67	217	0.94	35.02	245	3.44	13.06	31	0.48	48.39
Deprecate	∼6	18.52	3,826	16.61	36.96	752	10.56	37.9	905	14.07	44.75
Auto	∼10	0	/	/	/	/	/	/	/	/	/
Others	∼15	20.96	4,036	17.52	40.21	1,513	21.25	35.23	577	8.97	34.66
Single	∼87	17.12	17,200	74.65	40.69	5,802	81.50	38.12	4,209	65.47	41.29
Multiple	∼13	14.71	5,841	25.35	35.13	1,317	18.50	32.88	2,220	34.53	47.07

among different change intents, the sum of the percentages of change intents is larger than 100.

As shown in Table 3, the distribution of changes regarding intents varies in different projects. We can see that changes are unevenly distributed regarding the intents. For example, changes with intents '**Bug Fix**' and '**Resource**' are dominating across the four projects, i.e., they take up more 50% of all the changes, while the percentages of changes with intents '**Refactor**' and '**Merge**' are less than 4%. Note that the distribution in our dataset is consistent with that of manually categorized changes from existing study [24], in which changes under categories **Corrective** (i.e., addressing failures), **Adaptive** (i.e., changes for data and processing environment), and **Perfective** (i.e., addressing inefficiency, performance, and maintainability issues) are dominating with a percentage higher than 60%, which also confirms the effectiveness of our automated heuristic-based change intent classification (details are presented in Section 3.2).

While '**Feature**' and '**Refactor**' have higher rates of *LRE changes*, this is reasonable since both the '**Feature**' and '**Refactor**' introduce new functionalities or restructure existing code snippets, which are easy to be problematic and require more code review effort. Category '**Resource**' has a lower rate across the four projects. This may be because, compared to all other categories, changes in the '**Resource**' category modify the source code rarely.

> Software code changes are unevenly distributed regarding change intents. Changes with some change intents, i.e., '**Feature**' and '**Refactor**', have a higher probability to be *LRE changes*.

5.2. RQ2: Feasibility of Predicting LRE Changes

This question explores whether machine learning algorithms can learn models that identify among the submitted new changes under review. We use off-the-shelf machine learning algorithms from Weka [17] to build classification models. The used features include change intent, change history, owner experience, Word2Vec features, process features and metadata features (details are in Section 3.3).

Table 4: Comparison of different classifiers on predicting *LRE changes*. The best F1 and AUC values are in bold.

Project	ADTree		Logistic		NB		SVM		RF	
	F1	AUC	F1	AUC	F1	AUC	F1	AUC	F1	AUC
Microsoft	0.40	0.65	0.37	0.72	0.37	0.72	0.41	0.63	**0.46**	**0.76**
Qt	0.53	0.62	0.54	0.72	0.41	0.66	0.34	0.59	**0.57**	**0.71**
Android	0.58	0.68	0.54	0.71	0.58	0.70	0.50	0.65	**0.58**	**0.74**
OpenStack	0.60	0.64	0.64	0.76	0.62	0.68	0.66	0.70	**0.66**	**0.76**

Following existing studies [86, 23, 26, 94], we experiment with five widely used classifiers, i.e., Alternating Decision Tree (ADTree), Logistic Regression (Logistic), Naive Bayes (NB), Support Vector Machine (SVM), and Random Forest (RF). Note that this work does not intend to find the best-fitting classifiers or models, but to explore the feasibility of identifying *LRE changes* by using machine learning algorithms. Existing work [73] showed that selecting optimal parameter settings for machine learning algorithms could achieve better performance, thus we tune each of the classifiers with various parameters and use the ones that could achieve the best AUC value as our experiment settings. For each project, we build classification models and use the commonly used 10-fold cross-validation method to evaluate the prediction models [29, 59, 26, 23].

Table 4 shows the F1 and AUC values of each machine learning algorithm on the four experimental projects. Overall, of the five classifiers, RF consistently outperforms the others on each project. The improvements of RF compared to the other four classifiers range from 5.0 percentage points to 13.0 percentage points in AUC and 5.0 percentage points to 9.0 percentage points in F1. RF achieves similar AUC values on both the `Microsoft` project and open source projects, while it has a significantly lower F1 score (Wilcoxon signed-rank test, $p < 0.05$) on the `Microsoft` project. This may be because the `Microsoft` project has a lower rate of *LRE changes*, e.g., as shown in Table 1, the *LRE* rates of open-source projects are around 20.0 percentage points higher than that of the `Microsoft` project, which makes the data distribution more unbalanced in

the `Microsoft` project. Previous studies showed that the unbalance issue could decline the F1 scores [73]. As revealed in existing work [73], the unbalance issue of a dataset does not impact the AUC measure and they suggested that a machine learning model is considered applicable to classify a given dataset if the AUC is larger than 0.7. Hence, we use the AUC to compare prediction models. We could find that among the five examined machine learning classifiers, two of them, i.e., Logistic and RF, achieve AUC values larger than 0.7 on each of the four experimental projects, which confirms the feasibility of identifying *LRE changes* by using machine learning algorithms.

> Machine learning based prediction models can help predict *LRE changes*. The best model (i.e., RF) achieves AUC values larger than 0.71 on each experimental project.

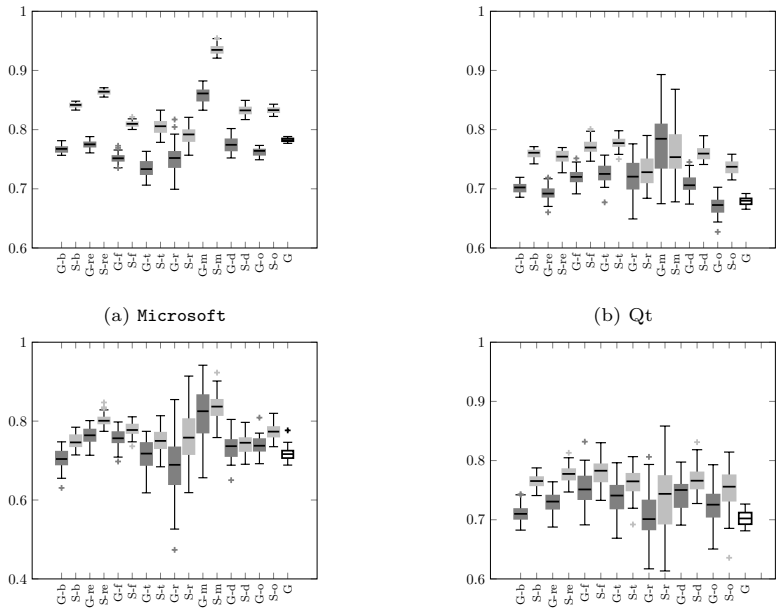
5.3. RQ3: Specific Models vs. General Models

In **RQ2**, we show that it is feasible to leverage machine learning classifiers to identify *LRE changes*. In this RQ, we further explore whether the machine learning classifiers built and trained on changes with a particular change intent, i.e., specific model, could achieve better performance than machine learning classifiers built and trained on all changes, i.e., general model. Specifically, for each project, we build and train the RF-based specific prediction models on changes with one particular change intent. We tune each of the RF-based classifiers with various parameter values and use the ones that could achieve the best AUC value as our experiment settings. In addition, we use 10-fold cross-validation method to evaluate the prediction models. The general model on the project is trained and evaluated on all changes without considering the change intents. Note that we exclude the specific model for category '**Merge**' on the project OpenStack, because it has very few numbers of instances.

Table 5 shows the comparison between the performance of the specific prediction models and the general models. Regarding F1, we can see that at least half of specific models outperform the general models across the four experimental projects. For example, six out of the eight specific models on the `Microsoft`

Table 5: Comparison between machine learning classifiers built on changes having a particular intent and machine learning classifiers built on all changes (i.e., general). Numbers in parenthesis are the differences between the specific models and the general models. Better F1 scores or AUC values that are larger than that of the general models are highlighted in bold. Note that we exclude the specific model for 'Merge' on the project OpenStack, since it only has 31 instances, which is not enough for training a machine learning classifier.

Change Intent	**Microsoft**				**Qt**				**Android**				**OpenStack**			
	P	R	F1	AUC	P	R	F1	AUC	P	R	F1	AUC	P	R	F1	AUC
Bug Fix	0.68	0.40	**0.51**(+0.05)	**0.84**(+0.08)	0.70	0.47	0.57(+0.00)	**0.77**(+0.06)	0.64	0.49	0.55(-0.03)	**0.76**(+0.02)	0.72	0.66	**0.69**(+0.03)	**0.79**(+0.03)
Resource	0.48	0.24	0.32(-0.14)	**0.79**(+0.03)	0.68	0.39	0.50(-0.07)	**0.76**(+0.05)	0.67	0.51	0.58(+0.00)	**0.82**(+0.08)	0.68	0.55	0.61(-0.05)	**0.81**(+0.05)
Feature	0.69	0.55	**0.61**(+0.15)	**0.81**(+0.05)	0.73	0.73	**0.73**(+0.16)	**0.78**(+0.07)	0.67	0.67	**0.67**(+0.09)	**0.77**(+0.03)	0.77	0.85	**0.81**(+0.15)	**0.81**(+0.05)
Test	0.61	0.22	0.32(-0.14)	**0.82**(+0.06)	0.71	0.55	**0.62**(+0.05)	**0.79**(+0.08)	0.62	0.53	0.58(+0.00)	0.74(+0.00)	0.74	0.77	**0.76**(+0.10)	**0.80**(+0.04)
Refactor	0.70	0.62	**0.65**(+0.19)	**0.79**(+0.03)	0.70	0.64	**0.67**(+0.10)	**0.76**(+0.05)	0.70	0.68	**0.69**(+0.11)	**0.76**(+0.02)	0.75	0.81	**0.78**(+0.12)	**0.78**(+0.02)
Merge	0.86	0.60	**0.71**(+0.25)	**0.95**(+0.19)	0.58	0.51	0.55(-0.02)	**0.77**(+0.06)	0.50	0.22	0.30(-0.28)	**0.81**(+0.07)	/	/	/	/
Deprecate	0.66	0.37	**0.47**(+0.01)	**0.84**(+0.08)	0.68	0.51	**0.58**(+0.01)	**0.77**(+0.06)	0.66	0.56	**0.61**(+0.03)	**0.78**(+0.04)	0.72	0.65	**0.69**(+0.03)	**0.80**(+0.04)
Others	0.68	0.50	**0.58**(+0.12)	**0.83**(+0.07)	0.65	0.51	0.57(+0.00)	**0.74**(+0.03)	0.60	0.51	0.55(-0.03)	**0.76**(+0.02)	0.62	0.51	0.56(-0.10)	**0.78**(+0.02)
General	0.51	0.41	0.46	0.76	0.60	0.54	0.57	0.71	0.61	0.55	0.58	0.74	0.68	0.65	0.66	0.76



(a) Microsoft (b) Qt

(c) Android (d) OpenStack

Figure 4: Comparison of perdition performance (AUC) between the specific models and the general models. The model with a prefix "G" means using the trained general model to predict changes with a particular intent. "b" represents 'Bug Fix', "re" represents 'Resource', "f" represents 'Feature', "t" represents 'Test', "r" represents 'Refactor', "m" represents 'Merge', "d" represents 'Deprecate', and "o" represents 'Others'. For example **G-b** means using the trained general model to predict changes with the 'Bug Fix' intent. **S-b** is the specific model trained and evaluated on changes with the 'Bug Fix' intent.

project generate better F1 values than the general model, the improvement is up to 25.0 percentage points and is 6.0 percentage points on average. We observe a similar situation on Qt and OpenStack, i.e., overall specific models are better than the general models, the improvements are up to 16.0 and 15.0 percentage points on Qt and OpenStack respectively. However, we also observe an exception in Android, although five out of the eight specific models generate better (or the same) F1 values than the general model, the overall improvement is negative, the reason is that the '**Merge**' category has an F1 value that is 28.0 percentage points lower than that of the general model. This is because the '**Merge**' category has a much lower *LRE* rate (i.e., 13.1%) than that of

all other categories (range from 29.0% to 50.4%) in Android, which makes the '**Merge**' unbalanced. Regarding AUC, we can observe that all the specific models outperform the general models across the four experimental projects. The improvement could be up to 19.0 percentage points and is 7.4 percentage points on average. Thus, from the comparison shown in Table 5, we conclude that overall the specific models achieve better prediction performance than the general models. Table 6 reports the standard deviation (**SD**) and mean values in our 10-fold cross validation based evaluation. As we can see from the table, the SD values for all the prediction models on the four projects are smaller than 0.1, which suggests the robustness of our proposed approaches.

Above all, we show that the specific models (built on changes with a particular change intent) are overall better than the general models (built on all the changes). One could also argue that using the general models to predict changes with a particular intent may have better performance than the corresponding specific model. To explore this issue, we further examine the performance of leveraging the general models to predict changes with a particular intent. Specifically, for each change intent, we randomly divide its changes into training dataset and test dataset (2/3 for training, 1/3 for test) following existing studies [37, 22]. For the specific model, we train the model on the training data and evaluate its performance on the test dataset. For the corresponding general model, we combine the training data from all specific models, and evaluate its performance on the test dataset of a specific model. We repeat the data splitting, model training, and evaluation 50 times to reduce bias.

Figure 4 shows the boxplots of the 50 times classification for each specific model and its corresponding general model on each project. In addition, we also show the boxplots of overall-general models (i.e., using 2/3 of all changes to train the models and evaluate the models on the left 1/3 changes without considering change intents). Each boxplot presents the AUC distribution (median and upper/lower quartiles) of a prediction model. We use gray (●), light gray (●), and white (○) to represent the specific models, corresponding general models, and the overall-general models respectively. We could observe that overall the

Table 6: The standard deviation (**SD**) and mean values in our 10-fold cross validation based evaluation for prediction models showed in Table 5.

Change Intent	**Microsoft**				**Qt**				**Android**				**OpenStack**			
	SD		**Mean**		**SD**		**Mean**		**SD**		**Mean**		**SD**		**Mean**	
	F1	**AUC**	**F1**	**AUC**	**F1**	**AUC**	**F1**	**AUC**	**F1**	**AUC**	**F1**	**AUC**	**F1**	**AUC**	**F1**	**AUC**
Bug Fix	0.02	0.01	0.49	0.82	0.01	0.01	0.55	0.76	0.03	0.02	0.45	0.70	0.01	0.01	0.67	0.76
Resource	0.01	0.01	0.30	0.78	0.02	0.01	0.48	0.75	0.03	0.01	0.48	0.76	0.03	0.01	0.52	0.78
Feature	0.02	0.01	0.60	0.81	0.02	0.01	0.65	0.72	0.02	0.01	0.62	0.76	0.02	0.02	0.79	0.78
Test	0.02	0.02	0.31	0.80	0.03	0.02	0.53	0.73	0.05	0.03	0.47	0.72	0.03	0.02	0.74	0.76
Refactor	0.03	0.01	0.64	0.78	0.03	0.03	0.64	0.73	0.05	0.02	0.64	0.77	0.03	0.04	0.80	0.71
Merge	0.04	0.03	0.71	0.95	0.07	0.04	0.54	0.76	0.09	0.06	0.30	0.82	/	/	/	/
Deprecate	0.01	0.01	0.46	0.85	0.02	0.01	0.55	0.76	0.04	0.03	0.51	0.74	0.03	0.02	0.65	0.77
Other	0.02	0.02	0.55	0.81	0.02	0.01	0.53	0.74	0.03	0.02	0.49	0.74	0.04	0.03	0.48	0.75
General	0.02	0.01	0.44	0.75	0.01	0.01	0.51	0.76	0.02	0.04	0.49	0.71	0.02	0.01	0.57	0.73

Table 7: Performance of predicting *LRE changes* with single and multiple change intents. Better F1 or AUC values are in bold.

Change Intent	Microsoft		Qt		Android		OpenStack	
	F1	AUC	F1	AUC	F1	AUC	F1	AUC
Single	**0.48**	**0.78**	**0.58**	**0.75**	**0.58**	**0.73**	**0.67**	**0.80**
Multiple	0.41	0.75	0.54	0.70	0.57	0.71	0.66	0.77

specific models outperform the general models on almost all the change intents across the four experimental projects. Specifically, for the `Microsoft` project, the mean AUC values of the specific models are around ten percentage points higher than that of the corresponding general models. For the open source projects, the mean AUC values of the specific modes are around five percentage points higher than that of the corresponding general models. In addition, all the specific models outperform the overall-general models.

Code review effort prediction models built on changes with particular change intents achieve better performance than the general prediction models that do not consider the change intents. Thus, we suggest to build specific models for better *LRE changes* prediction.

5.4. RQ4: Single Intent vs. Multiple Intents

As shown in RQ1 (Section 5.1), in this study we labeled changes with multiple intents. This RQ explores the performance of *LRE changes* prediction models on changes with a single intent and changes with multiple intents. Specifically, for each project, we build and train the RF-based prediction models with changes with only a single intent and changes with multiple intents respectively. We tune each of the RF-based classifiers with various parameter values and use the ones that could achieve the best AUC value as our experiment settings. We also use the 10-fold cross-validation method to evaluate the models.

Table 7 shows the F1 scores and AUC values of the prediction models for changes with single and multiple change intents in the four experimental projects. As we can see, the prediction models for changes with a single intent significantly outperform the prediction models for changes with multiple

intents in both F1 and AUC across the four experimental projects (Wilcoxon signed-rank test, $p < 0.05$). In terms of F1, the improvement could be up to 7.0 percentage points and is 3.3 percentage points on average. For AUC, the improvement could be up to 5.0 percentage points and is 3.3 percentage points on average. One of the possible reasons for this difference is that changes with a single intent are mainly made for one specific purpose, which makes them easier to be distinguished by machine learning classifiers than complex changes with multiple intents.

Machine learning based classifiers generate better performance on changes with a single change intent than changes with multiple change intents.

6. Large-scale Deployment of Intent Analysis

The above change intent analysis and its application on LRE identification proposed in this work has already been used in `Microsoft` to provide the review effort and intent information of changes for reviewers to accelerate the review process. To show how to deploy our approaches in real-world practice, in this section, we discuss the large-scale deployment of code change intent analysis in `Microsoft`. Specifically, the tool provides intent labelling service for pull requests with different intent categories for over 200 repositories inside `Microsoft` using the Azure DevOps platform [3]. The feature has been enabled since November 2018.

6.1. Implementation

We built the change intent analysis tool and integrated it into Azure by using the extensibility mechanisms in Azure DevOps. In this section, We first briefly introduce the Azure DevOps platform (Section 6.1.1) and then discuss the architecture for the intent labelling service (Section 6.1.2).

6.1.1. Background of Azure DevOps

Azure DevOps is a platform for collaborative software development used both inside and outside Microsoft by hundreds of thousands of users. It offers

key functionalities needed for software development such as GIT based version control, boards for work item tracking and build/release pipelines for continuous integration and continuous testing. It is the primary platform used in Microsoft for software development. So, as the logical choice, we decided to enable intent labelling for Pull Requests on the Azure DevOps platform. In order to automatically label Pull Requests, we leverage these extensibility features offered by Azure DevOps:

1. REST APIs - Azure DevOps provides a rich set of APIs which allows 3rd party services to perform CRUD operations for various entities such as Pull Requests, Builds, Tests, etc.

2. Service Hooks - In order to label the Pull Requests in real time, we use the service hooks provided by Azure DevOps. These service hooks allow us to subscribe to various events such as pull request creation, update and completion. So, the intent labelling service can react to these events in real time before the pull request is reviewed.

3. Pull Request Labels - This is a key extensibility feature offered by Azure DevOps for Pull requests. Azure provides labels [4] to each pull request, which can be used to as tags for various purposes such as importance, status, etc. In this work, we use the label feature for tagging the pull requests with the semantic intents. These labels can be added or removed by anyone allowing collaborative tagging. As we discuss later in this section, we rely on the removal of labels as a feedback channel. Note that, we use the intent labels listed in Table 2 to label each pull request.

6.1.2. Architecture of Intent Analysis Service

For each type of intents listed in Table 2, we followed its heuristics to implement its identification process. We implemented the change intent labelling service based on the Microsoft Azure cloud platform. We used Azure SQL as the underlying data store and Azure Cloud Services for building the service. Here are the key components of this service:

1. Data store - We use SQL Azure for the relational data store where we persist pull request meta-data such as title, description, etc. that are used for change intent identification. We also store the results of the intent classification in a separate table for reporting and analytics. This allows us to do experimentation retrospectively as we refine the heuristics for intent classification.

2. Cloud service - We have implemented the intent labelling service based on the multi-tier application architecture as described in [53]. We first use Azure Service Bus to subscribe to pull request messages from Azure DevOps. Then, we implement a cloud service which is triggered by the service bus. Specifically, whenever there is a new message, the intent labelling service will be triggered and further processes the message and fetches the meta-data for the pull requests using the Azure DevOps APIs. After that, it performs the intent classification and adds intent labels to the pull requests again using the Azure DevOps API.

6.2. User Scenarios

The intent labels provide semantic summarization of the pull requests to identify what kind of changes are being proposed in the code review. This information is helpful not just for the code reviewers but even for people like product managers and project leads. We surface the intent labels at two key user interfaces in the Azure DevOps:

1. Pull Request Listing - We surface the intent labels in the pull request listing page in Azure DevOps, as shown in Figure 5. For each pull request, an intent label with gray background is attached. For example, in Figure 5, the pull request titled "Code complexity analysis" is labelled as 'Feature'. Such feature allows users to glance through the list of pull requests and understand the semantic type of pull requests which are currently active or have been completed. In addition,this new feature is especially useful for

architects, tech leads, and product managers to have a high level overview of the software development activity.

2. Pull Request Detail - We also surface the intent labels in the pull request details page in Azure DevOps. For example, Figure 6 shows the detailed page of pull request "Bug fix for fetching timestamp for build ingestion". The change intent of this pull request, i.e., 'BugFix', is also presented in the right bottom panel. This new feature allows developers who are reviewing the pull request to have a quick insight into the intent of the pull request.

6.3. Feedback Loop

In order to gather user feedback, we have designed and implemented an explicit feedback loop. Unlike scenarios such as alerting, we don't directly interact with the end users, i.e., pull request authors and reviewers. Also, feedback channels such as surveys are prohibitive in terms of time and efficiency. So, we rely on the removal of intent labels by the users as a negative feedback mechanism. At the time of the study, we have observed less than 1% of the intent labels being removed. Once we have a significant amount of feedback, we further do a detailed analysis and also use it to improve the intent classifiers as part of future work. We also received offline feedback (over email) about incorrect labelling for pull requests from few repositories.

6.4. Practical Guidelines

In this work, we have used the label feature provided by Azure DevOps[7] and Github[8] to categorize and tag the Pull Requests with the semantic intents. Prior work has focused on using Pull Request comments [32, 41, 88, 90, 39] which can easily get crowded with the comments from the reviewers and the automated

[7] https://docs.microsoft.com/en-us/azure/devops

[8] https://help.github.com/en/github/managing-your-work-on-github/applying-labels-to-issues-and-pull-requests

bots. To the best of our knowledge, this is the first work to use automated labels for improving Pull Request organization and review experience. Here are the key takeaways for researchers and practitioners based on our experience from building and deploying the automated intent labelling service at Microsoft:

1. **Plugin-based Tool Development**: Software projects usually have their workflow. As a tool to help developers accelerate their review process, we were not expected to break the existing code review procedures at `Microsoft`. To achieve this goal, we developed our tool as a plugin, which does not affect the code review process used in `Microsoft`. For researchers who plan to deploy their tools into real-world software projects, we recommend the plugin-based software development.

2. **Bugs due to pull request templates**: Both Azure DevOps[9] and GitHub[10] provide the functionality to create pull request templates. These templates are generally created by repository admins to standardize the pull request descriptions. For instance, at `Microsoft` we have observed templates with placeholders for the pull request authors to fill-in information about the testing and the related bugs, if any. While this is a very useful feature, it can affect any bots which mines the pull request descriptions because the templates often contain keywords such as "bugs", "testing", "feature", etc. We learnt about this important issue from offline feedback from some of our users. For developers who plan to deploy our tool by using the pull request templates provided by GitHub should notice this issue.

3. **Feedback mechanism**: Pull Request comment based bots have several ways to collect explicit feedback such as comment resolution status, replies to the comments and even likes on the comments. However, collecting feedback on labels is non-trivial. Our proposed methodology of mining

[9] https://docs.microsoft.com/en-us/azure/devops/repos/git/pull-request-templates

[10] https://help.github.com/en/github/building-a-strong-community/about-issue-and-pull-request-templates

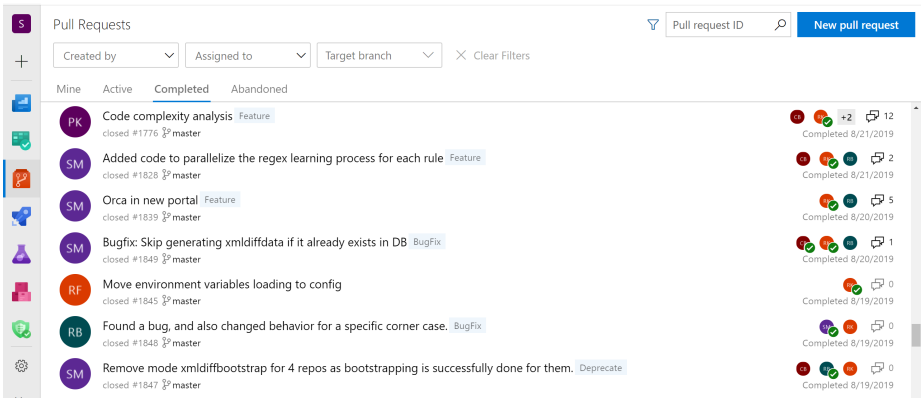


Figure 5: Pull request list in Azure DevOps with intent labels.

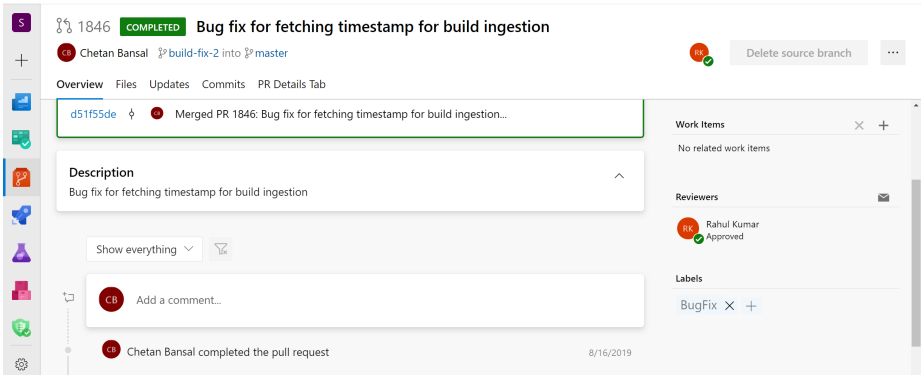


Figure 6: Pull request UI in Azure DevOps with change intent labels.

label edits and removals can be used as a feedback mechanism for other bots which uses the label feature.

7. Discussion

7.1. LRE Rate Analysis

As shown in Table 3, in three of the four projects, changes with a single intent have a higher *LRE* rate than that of changes with multiple intents. To explore the reason behind this phenomenon, we first break down the number of changes with multiple intents from the four projects, which are shown in

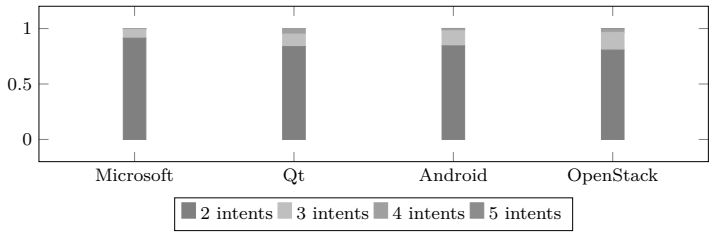


Figure 7: The distribution of changes with multiple change intents.

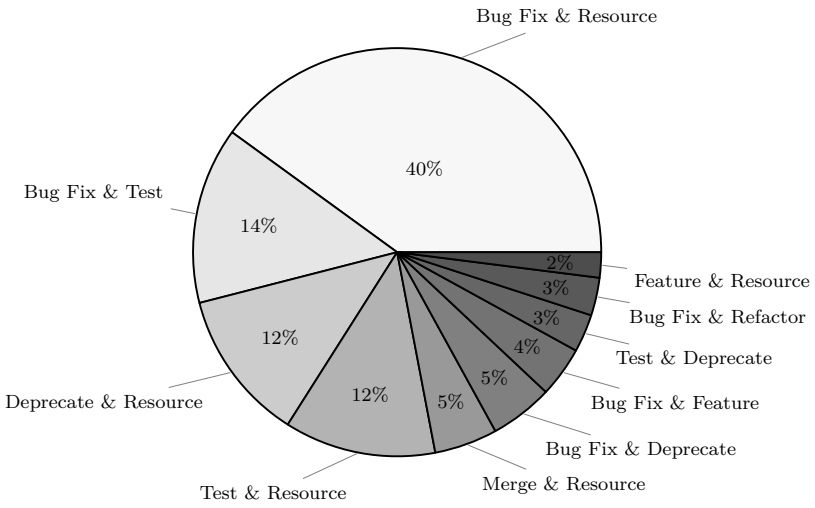


Figure 8: The distribution of code changes with multiple intents.

Figure 7. We could observe that among the changes with multiple intents, changes that have two intents are dominating, i.e., 90% of the changes with multiple intents involve only two different intents. Thus, we narrow down the analysis to explore the distribution of changes with two change intents. To do this, we collect the changes with two intents from the four projects and count the number of different intent combinations among these changes. Figure 8 shows the top ten types of changes with two intents, which cover more than 97% of all the changes with two intents.

As we could see that 'Bug Fix' & 'Resource', 'Bug Fix' & 'Test', 'Deprecate' & 'Resource', and 'Test' & 'Resource' are the dominating combinations. With

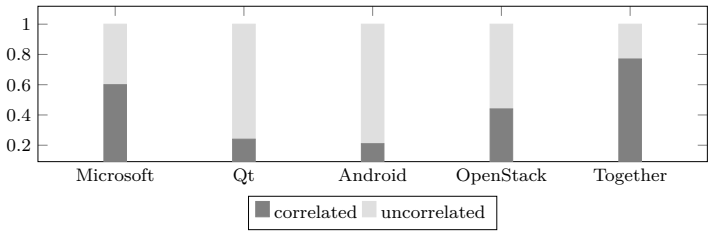


Figure 9: The distributions of correlated and uncorrelated features in each project and across the projects (i.e., Together).

the distribution chart in Figure 8, we infer the reasons why changes with multiple intents have lower *LRE* rates as follows. First, each intent may represent an independent task and developers may modify source code for each intent separately, which provides them more opportunities to check the modified code and eventually improve the quality of the change. Second, some of the combined intents represent the standard software quality assurance process, which can help improve the quality of the changes. Taking changes with double intents 'Bug Fix' & 'Test' as an example, developers may first fix a bug and then modify the test cases to validate the fix of the bug, which makes the changes more reliable.

7.2. Feature Correlation Analysis

Following existing studies [14], we use the Spearman rank correlation [87] to compute the correlations between the metrics described in Section 3.3 and the review effort of changes, i.e., *regular changes* or *LRE changes*. Values greater than 0.10 can be considered a small effect size; values greater than 0.30 can be considered a medium effect size [93]. In this work, we consider the values larger than 0.10 or smaller than -0.10 as correlated, others are uncorrelated.

Figure 9 shows the distributions of correlated and uncorrelated features in each project and across the four projects. As we could observe, in the `Microsoft` project, the percentage of correlated features (around 60%) is larger than that of the open source projects (less than 50%). One of the possible reasons is that compared to the open source projects, the `Microsoft` project has a larger experimental dataset, e.g., around 5X of the size of open source projects, which

provides sufficient data to evaluate each feature and reduce the potential bias. Overall more than 70% features are correlated with review effort of changes, i.e., *regular changes* or *LRE changes*. We further examined the selected features and found that they covered five different feature types, i.e., change intents, revision history, owner experience, Word2Vec features, and process features, while none of the metadata features is selected as correlated. This observation suggests that the commit time of a change does not affect its review effort.

The Spearman correlation analysis shows that most of the collected features are correlated with the review effort of changes, thus the collected features are applicable for identifying *LRE changes*.

7.3. Performance of LRE *Prediction on Other Intent Categorization*

In this work, we proposed a new categorization of change intent based on the existing study [24]. As [24] has also proposed a high-level categorization, which contains six types of changes, i.e., 'Corrective', 'Adaptive', 'Perfective', 'Implementation', 'Non functional', and 'Others'. To examine our proposed *LRE* prediction models on the high-level categorization, we followed their categorization instructions and grouped the nine different types of changes in Table 2 into the six high-level categories, the detailed mapping between the two types of categorization is shown in Table 8. Note that as the '**Auto**' changes only exist in `Microsoft` project, we exclude them in this experiment.

Following our experiment settings in Section 4, we rebuild and evaluate *LRE* prediction models with features described in Section 3.3 on changes labeled with the high-level categorization on the three open source projects. Note that, we only examine Random Forest (RF) based classification models as RF achieves better performance compared to the other four examined classifiers, i.e., ADTree, Logistic, NB, and SVM (details are in Section 5.2). Table 9 shows the F1 scores and AUC values of the prediction models for changes with change intent categorization proposed in [24]. In order to check whether our *LRE* prediction model has different performance on the two different change intent categorizations, We further conduct the Wilcoxon signed-rank test ($p < 0.05$) to

Table 8: Mapping between change intent categories proposed in this study and the high-level change intent categorization proposed in [24].

Change categories in [24]	**Change categories proposed in this work**
Corrective	'Bug Fix'
Adaptive	'Resource', 'Test'
Perfective	'Refactor', 'Deprecate'
Implementation	'Feature'
Non functional	'Merge'
Other	'Other'

Table 9: Performance of *LRE* prediction models with change intent categorization proposed in [24].

Change Intent	**Qt**		**Android**		**OpenStack**	
	F1	**AUC**	**F1**	**AUC**	**F1**	**AUC**
Corrective	0.57	0.77	0.55	0.76	0.69	0.79
Adaptive	0.52	0.77	0.58	0.82	0.63	0.83
Perfective	0.60	0.77	0.61	0.78	0.72	0.81
Implementation	0.73	0.78	0.67	0.77	0.81	0.81
Non functional	0.55	0.77	0.30	0.81	/	/
Other	0.57	0.74	0.55	0.76	0.56	0.78

compare the F1 values and AUC values of the two change intent categorizations on the three open source projects. Our results show that there is no significant difference between the performance of *LRE* prediction models on our change intent categorization and the high-level categorization from [24], which suggests that our proposed approach is generalizable for change intent prediction tasks.

7.4. Survey on the usefulness of our approach

To further assess the usefulness of our approach, we have conducted a a questionnaire survey with 20 developers who have 1 to 5 years' experience with

Table 10: Distribution of participants regarding their experience with code review.

Experience	Num of participants
3-5 years experience in code review	8
1-3 years experience in code review	9
<1 years experience in code review	3

code review. Specifically, these developers were selected from local IT companies during offline technical summits/conferences through the connections of the first author, we approached and discussed with 30 developers and filtered out 10 who either did not have enough background knowledge with code review or did not have time to do our survey (which may take 10 minutes). We collected 20 finished hard-copy questionnaires, the questionnaire is available online[11]. Details of these participants according to their experience are summarized in Table 10. To avoid evaluation bias, none of these participants are from `Microsoft`.

In the questionnaire, we first demonstrates a short description of our approaches, the change intent identification and its performance, and a summarized evaluation result of our *LRE* change identification model on the four evaluation projects. Then it asks four questions shown in Table 11. We provide five options for these questions and also allow respondents freely express their opinion for each of these questions. Specifically, in the first question, we ask developers whether they think finding *LRE* changes is reasonable. Questions 2 and 3 investigate our change intent analysis approach and Questions 4 and 5 investigate our *LRE* change identification model.

As showed in Table 11, of all 20 respondents, 19 of them (95%) agree that finding the *LRE* changes is useful for accelerating code review. This confirms the usefulness of our *LRE* changes identification approach in general. Only 1 (5%) holds a conservation option, i.e., disagreed. The participant also gives his reason for the option, i.e., only finding out *LRE* changes without identifying the

[11] `https://bitbucket.org/wangsonging/ist2020_repo/src/master/survery/`

Table 11: Results of survey

Questions	Strongly Disagree	Disagree	Neither	Agree	Strongly Agree	Total
Q1: Do you think a tool that can find LRE changes is useful for code review?	0	1	0	5	14	20
Q2: Do you think our tool is useful to help identify change intent of changes?	0	0	0	3	17	20
Q3: Would you like to use our tool to help with change intent analysis?	0	0	0	5	15	20
Q4: Do you think our tool is useful to help identify LRE changes?	0	5	1	9	5	20
Q5: Would you like to use our tool to help with your code review process?	0	6	0	11	3	20

reason of *LRE* changes may not enough. This paves the direction for further research. For the change intent analysis, all the participants agree that our change intent analysis is useful and are willing to use it in their development tasks. In addition, for *LRE* change identification, 70% of the participants agree that our *LRE* identification approach could be useful and would like to use our tool to help with their code review process. Meanwhile, 30% participants also hold conservation options. For the reason of disagreement, the participant mainly concerns the performance of our approach on projects with different program languages, as well as the potential risk of false positives, i.e., *LRE* changes that are missed by our approach. In the future, we will explore new approaches to improve the performance of our approach.

8. Threats to Validity

Internal Validity. The main threat to internal validity is about the annotation of change intents, subjectivity of annotation, and miscategorization. The annotation relied on our manually refined heuristics, and although this approach is a common practice, this process contains bias since the authors of this paper are not the developers of these projects. To mitigate this, authors worked independently to annotated the data and refined the heuristics. In addition, we chose a setup that ensures that every heuristic is cross-validated and the classification conflicts have to pass the third inspection. Note that since these heuristics are summarized on changes collected from four different projects, although they come from different domains and use four different program languages, these heuristics may not work well for projects that have special characteristics that do not appear in the four projects studied in this study. In this work, we define *Large-Review-Effort (LRE) changes* based on the consensus of developers from `Microsoft`, the performance of our approach may vary with different thresholds. For labeling *LRE changes* on the three open-source projects, we use the number of submitted patchsets as the threshold, i.e., if the number of submitted patchsets is larger than two, we labeled the change as a *LRE change* otherwise

we labeled it as *regular*. However, such an approach might be inaccurate, the performance of our approach might be different with different thresholds to label *LRE changes* on these open source projects.

In this work, we use the review duration and the number of reviewers involved to measure the review effort for each change since current code review systems (both commercial and open-source) do not support review effort measurement, which might be imperfect. Thus, future research should revisit our study by using more accurate approaches to measure the review effort.

External Validity. In this work, we use a project from `Microsoft` and three open source projects to evaluate our proposed approach. Since they adopt different code review systems, i.e., the `Microsoft` project adopts a custom code review system and the open source projects adopt the Gerrit code review system. Thus, the proposed approach might not work for projects that adopt other code review systems.

In this work, following the existing studies [94, 26, 86, 81, 72, 10, 83], we use the widely adopted Precision, Recall, F1, and AUC to measure the performance of predicting *LRE changes*. The results of this work could be different with different performance metrics, e.g., Matthews correlation coefficient (MCC)[12]. Thus, future research should revisit our study by using different metrics to measure the performance of our approaches.

9. Related Work

In this section, we discuss related work from three different aspects: change intent analysis, software defect prediction, and software code review.

9.1. Change Intent Analysis

In order to understand the change intents of code changes, Swanson [70] first proposed a classification of maintenance activities as corrective, adaptive,

[12] `https://en.wikipedia.org/wiki/Matthews_correlation_coefficient`

and perfective. Along this line, many change analysis models have been proposed [67, 68, 9, 35, 36, 19, 89, 55, 42, 24, 95]. Buckley et al. [9] proposed a taxonomy of software evolution to characterize the mechanisms of changes. Lehnert et al. [35, 36] proposed comprehensive investigations of software change impact analysis. Later, Hassan et al. [19] extended Swanson's categorization by adding three new categories, i.e., bug fixing changes, general maintenance changes, and feature introduction changes. Wilkerson et al. [89] proposed a taxonomy of the types of impacts that can result from source code changes in both procedural and object-oriented code. Paulius et al. [55] proposed a taxonomy of change aspects in the feature modeling domain. Hindle et al. [24] extended Swanson's categorization with two new categories, i.e., feature addition and non-functional. Hassan's categories are not specific enough, which only provided high-level information of categories. Hindle's extended categories contain more detailed types of changes for each category. In this study we adopted the fine-grained change type information provided Hindle's work [24].

In order to classify changes, Mockus et al. [46] and Hassan et al. [19] proposed to classify software changes by using the textual information. Experiment results showed their approaches could produce results similar to manual classifications performed by professional developers. Their work inspired us to heuristically classify changes based on the textual information. Hindle et al. [23] provided another classification model focusing on large code commits only (i.e., the top 1% of the commits in a project ranked by the number of files changed). They took the distribution of terms from commit messages, author, module, and file type as features. They validated the models on 2,000 commits from 9 projects via 10-fold cross-validation, which achieved accuracies consistently above 50%. Yan et al. [91] presented a discriminative Probability Latent Semantic Analysis (DPLSA) model to classify software changes as corrective, adaptive, and perfective. Their experiments showed that the proposed prediction models could achieve an average precision above 70%.

Note that we do not use the machine learning based change classification models proposed by Hindle et al. [23] or Yan et al. [91], the reason is that we

find the refined heuristics provide us relatively acceptable accuracies, i.e., above 80%.

9.2. Software Defect Prediction

Software prediction techniques leverage various software metrics to build machine learning models to predict unknown defects in the source code [20, 94, 59, 34, 50, 49]. Most defect prediction techniques leverage features that are manually extracted from historical defect data to train machine learning based classifiers [44]. Software prediction features can be divided into static code features, process features, semantic features [86], etc. The process features include code deletion, code addition, authors, etc., which are collected during the development process [59]. Moser et al. [49] used the number of revisions, authors, past fixes, and ages of files as features to predict defects. Nagappan et al. [50] proposed code churn features, and shown that these features were effective for defect prediction. Hassan et al. [20] used entropy of change features to predict defects. Lee et al. [34] proposed 56 micro interaction metrics to improve defect prediction. Other process features, including developer individual characteristics [26, 71] and collaboration between developers [44], were also useful for defect prediction. In this study we also use the process features to build the risky change prediction models. Menzies et al. [43] showed that the local defect prediction models, i.e., built on a subset of the data, result in better performance than the global models that built on all the data. We report a similar finding that specific risky change prediction models outperform general models.

Different from the above studies, instead of predicting whether the changes or changed files are fault-prone, i.e., contain bugs, we measure the riskiness of changes regarding the code review effort and aim at finding the *risky changes* that have more than one round code review, which we believe can provide the riskiness regarding the code review effort and intent information of changes for reviewers to accelerate the review process.

9.3. Software Code Review

Code review is a manual inspection of source code by humans, which aims at identifying potential defects and quality problems in the source code before its deployment in a live environment [33, 56, 66, 38, 48, 7, 65, 54]. Many studies have examined the practices of code review. Stein et al. [66] explored the distributed, asynchronous code inspections. They studied a tool that allowed participants at separate locations to discuss faults. Porter et al. [56] reported on a review of studies on code inspection in 1995 that examined the effects of different factors on code inspections. Laitenburger [33] surveyed code inspection methods, and presented a taxonomy of code inspection techniques.

Votta [80] found that 20% of the interval in a "traditional inspection" is wasted due to scheduling. In recent years, Modern Code Review (MCR) has been developed as a tool-based code review system and becomes popular and widely used in both commercial software (e.g., Google [64], Microsoft [2, 32]) and open-source software (e.g., Android, Qt, and OpenStack) [18]. Rigby et al. [61, 62, 60] have done extensive work examining code review practices in OSS development. Hellendoorn et al. [21] used language models to quantitatively evaluate the influence of stylistic properties of code contributions on the code review process and outcome. Sutherland and Venolia [69] conducted a study at Microsoft regarding using code review data for later information needs. Bacchelli & Bird find that understanding of the code and the reason for a change is the most important factor in the quality of code reviews [5]. Some other studies focus on accelerating code review process by recommending reviewers [79, 63, 78, 77], decomposing code review changes with multiple changesets [6, 76, 31, 16].

In this paper, we explore the feasibility of accelerating code review by identifying the risky code changes that require multiple rounds of code review or are reverted.

10. Conclusion

This paper presents the first study of *LRE changes* in code review system, i.e., changes that have more than two iterations of code review by using the change intents. We conduct our study on one large-scale commercial project from `Microsoft`, and three open source projects, i.e., Qt, Android, and OpenStack. Our experiment results show the feasibility of using machine learning based prediction models to identify *LRE changes*. The change intent analysis and its application on *LRE changes* identification proposed in this study has already been used in `Microsoft` to provide the review effort and intent information of changes for reviewers to accelerate the review process. To show how to deploy our approaches in real-world practice, We report a case study of developing and deploying the intent analysis system in `Microsoft`. We have also shared practical guidelines for help other developers who plan to develop the proposed approach for *LRE changes* identification. Moreover, we also evaluate the usefulness of our approaches by using a questionnaire survey. The feedback from developers demonstrates its practical value.

Our work on change intents is just the first step in a large body of work. In the future, we would like to explore if change intent improves the fidelity and accuracy of other prediction tasks, e.g., code reviewer recommendation, software defect prediction, and effort estimation.

11. Acknowledgments

We would like to acknowledge the invaluable contributions and support of B. Ashok, Jim Kleewein, Shawn Martelock, Nitin Sharma and Rahul Kumar.

References

[1] Abdulkareem Alali, Huzefa Kagdi, and Jonathan I Maletic. 2008. What's a typical commit? a characterization of open source software repositories. In *ICPC'08*. 182–191.

[2] Sumit Asthana, Rahul Kumar, Ranjita Bhagwan, Christian Bird, Chetan Bansal, Chandra Maddila, Sonu Mehta, and B Ashok. 2019. WhoDo: automating reviewer suggestions at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 937–945.

[3] Azure. 2019. Azure DevOps Services | Microsoft Azure. `https://azure.microsoft.com/en-in/services/devops/`. Accessed: 2019-12-10.

[4] Azure. 2019. Pull Request Labels (Azure DevOps Git). `https://docs.microsoft.com/en-us/rest/api/azure/devops/git/pull%20request%20labels`. Accessed: 2019-12-10.

[5] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *ICSE'13*. 712–721.

[6] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K Lahiri. 2015. Helping developers help themselves: Automatic decomposition of code review changesets. In *ICSE'15*. 134–144.

[7] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. 2014. Modern code reviews in open-source projects: Which problems do they fix?. In *MSR'14*. 202–211.

[8] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. 2003. A Neural Probabilistic Language Model. *The Journal of Machine Learning Research* 3 (2003), 1137–1155.

[9] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. 2005. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice* 17, 5 (2005), 309–332.

[10] Qiang Cui, Song Wang, Junjie Wang, Yuanzhe Hu, Qing Wang, and Mingshu Li. 2017. Multi-Objective Crowd Worker Selection in Crowdsourced Testing. In *SEKE'17*. 1–6.

[11] Michael Fagan. 2002. Design and code inspections to reduce errors in program development. In *Software pioneers*. 575–607.

[12] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. 2001. *The elements of statistical learning*. Vol. 1.

[13] Ying Fu, Meng Yan, Xiaohong Zhang, Ling Xu, Dan Yang, and Jeffrey D Kymer. 2015. Automated classification of software change messages by semi-supervised Latent Dirichlet Allocation. *IST'15* 57 (2015), 369–377.

[14] Emanuel Giger, Martin Pinzger, and Harald C Gall. 2012. Can we predict types of code changes? an empirical analysis. In *MSR'12*. 217–226.

[15] Todd L Graves, Alan F Karr, James S Marron, and Harvey Siy. 2000. Predicting fault incidence using software change history. *TSE'00* 26, 7 (2000), 653–661.

[16] Bo Guo and Myoungkyu Song. 2017. Interactively Decomposing Composite Changes to Support Code Review and Regression Testing. In *COMPSAC'17*, Vol. 1. 118–127.

[17] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. 2009. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter* 11, 1 (2009), 10–18.

[18] Kazuki Hamasaki, Raula Gaikovina Kula, Norihiro Yoshida, AE Cruz, Kenji Fujiwara, and Hajimu Iida. 2013. Who does what during a code review? datasets of oss peer review repositories. In *MSR'13*. 49–52.

[19] Ahmed E Hassan. 2008. Automated classification of change messages in open source projects. In *Proceedings of the 2008 ACM symposium on Applied computing*. 837–841.

[20] Ahmed E. Hassan. 2009. Predicting Faults Using the Complexity of Code Changes. In *ICSE'09*. 78–88.

[21] Vincent J Hellendoorn, Premkumar T Devanbu, and Alberto Bacchelli. 2015. Will they like this?: Evaluating code contributions with language models. In *MSR'15*. 157–167.

[22] Kim Herzig, Sascha Just, Andreas Rau, and Andreas Zeller. 2013. Predicting defects using change genealogies. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 118–127.

[23] Abram Hindle, Daniel M German, Michael W Godfrey, and Richard C Holt. 2009. Automatic classication of large changes into maintenance categories. In *ICPC'09*. 30–39.

[24] Abram Hindle, Daniel M German, and Ric Holt. 2008. What do large commits tell us?: a taxonomical study of large commits. In *MSR'08*. 99–108.

[25] Abram Hindle, Michael W Godfrey, and Richard C Holt. 2007. Release pattern discovery via partitioning: Methodology and case study. In *MSR'07*. 19.

[26] Tian Jiang, Lin Tan, and Sunghun Kim. 2013. Personalized defect prediction. In *ASE'13*. 279–289.

[27] Yujuan Jiang, Bram Adams, and Daniel M German. 2013. Will my patch make it? and how fast?: Case study on the linux kernel. In *MSR'13*. 101–110.

[28] Ji-Hyun Kim. 2009. Estimating classification error rate: Repeated cross-validation, repeated hold-out and bootstrap. *Computational statistics & data analysis* 53, 11 (2009), 3735–3745.

[29] Sunghun Kim, E James Whitehead Jr, and Yi Zhang. 2008. Classifying software changes: Clean or buggy? *TSE'08* 34, 2 (2008), 181–196.

[30] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. 2007. Predicting faults from cached history. In *ICSE'07*. 489–498.

[31] Patrick Kreutzer, Georg Dotzler, Matthias Ring, Bjoern M Eskofier, and Michael Philippsen. 2016. Automatic clustering of code changes. In *MSR'16*. 61–72.

[32] Rahul Kumar, Chetan Bansal, Chandra Maddila, Nitin Sharma, Shawn Martelock, and Ravi Bhargava. 2019. Building sankie: an AI platform for DevOps. In *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*. IEEE, 48–53.

[33] Oliver Laitenberger. 2002. A survey of software inspection technologies. In *Handbook of Software Engineering and Knowledge Engineering: Volume II: Emerging Technologies*. 517–555.

[34] Taek Lee, Jaechang Nam, DongGyun Han, Sunghun Kim, and Hoh Peter In. 2011. Micro interaction metrics for defect prediction. In *FSE'11*. 311–321.

[35] Steffen Lehnert. 2011. A taxonomy for software change impact analysis. In *IWPSE-EVOL'11*. 41–50.

[36] Steffen Lehnert, Matthias Riebisch, et al. 2012. A taxonomy of change types and its application in software evolution. In *ECBS'12*. 98–107.

[37] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. 2008. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering* 34, 4 (2008), 485–496.

[38] Laura MacLeod, Michaela Greiler, Margaret-Anne Storey, Christian Bird, and Jacek Czerwonka. 2017. Code Reviewing in the Trenches: Understanding Challenges and Best Practices. *IEEE Software* (2017).

[39] Chandra Maddila, Chetan Bansal, and Nachiappan Nagappan. 2019. Predicting pull request completion time: a case study on large scale cloud services. In *Proceedings of the 2019 27th ACM Joint Meeting on European*

Software Engineering Conference and Symposium on the Foundations of Software Engineering. 874–882.

[40] Vahid Mashayekhi, Janet M Drake, W-T Tsai, and John Riedl. 1993. Distributed, collaborative software inspection. *IEEE software* 10, 5 (1993), 66–75.

[41] Sonu Mehta, Ranjita Bhagwan, Rahul Kumar, Chetan Bansal, Chandra Maddila, B Ashok, Sumit Asthana, Christian Bird, and Aditya Kumar. 2020. Rex: Preventing bugs and misconfiguration in large services using correlated change analysis. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 435–448.

[42] Tom Mens, Jim Buckley, Matthias Zenger, and Awais Rashid. 2003. Towards a taxonomy of software evolution. In *Proceedings of the International Workshop on Unanticipated Software Evolution*.

[43] Tim Menzies, Andrew Butcher, Andrian Marcus, Thomas Zimmermann, and David Cok. 2011. Local vs. global models for effort estimation and defect prediction. In *ASE'11*. 343–351.

[44] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. 2010. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering* 17, 4 (2010), 375–407.

[45] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. 2013. Distributed Representations of Words and Phrases and Their Compositionality. In *NIPS'13*. 3111–3119.

[46] Audris Mockus and Lawrence G Votta. 2000. Identifying Reasons for Software Changes Using Historic Databases. In *ICSM'00*. 120.

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

[48] Rodrigo Morales, Shane McIntosh, and Foutse Khomh. 2015. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In *SANER'15*. 171–180.

[49] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. 2008. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *ICSE'08*. 181–190.

[50] Nachiappan Nagappan and Thomas Ball. 2007. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *ESEM'07*. 364–373.

[51] Jaechang Nam, Wei Fu, Sunghun Kim, Tim Menzies, and Lin Tan. 2017. Heterogeneous defect prediction. *TSE'17* (2017).

[52] Jaechang Nam and Sunghun Kim. 2015. CLAMI: Defect Prediction on Unlabeled Datasets. In *ASE'15*. 452–463.

[53] .NET. 2019. .NET multi-tier application using Azure Service Bus. `https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-dotnet-multi-tier-app-using-service-bus-queues`. Accessed: 2019-12-10.

[54] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. 2018. Information needs in contemporary code review. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW (2018), 135.

[55] Paulius Paskevicius, Robertas Damasevicius, and Vytautas Štuikys. 2012. Change impact analysis of feature models. In *ICIST'12*. 108–122.

[56] Adam Porter, Harvey Siy, and Lawrence Votta. 1996. A Review of Software Inspections. *Advances in Computers* 42 (1996), 39–76.

[57] Jens C Pruessner, Clemens Kirschbaum, Gunther Meinlschmid, and Dirk H Hellhammer. 2003. Two formulas for computation of the area under

the curve represent measures of total hormone concentration versus time-dependent change. *Psychoneuroendocrinology* 28, 7 (2003), 916–931.

[58] Ranjith Purushothaman and Dewayne E Perry. 2005. Toward understanding the rhetoric of small source code changes. *TSE'05* 31, 6 (2005), 511–526.

[59] Foyzur Rahman and Premkumar Devanbu. 2013. How, and why, process metrics are better. In *ICSE'13*. 432–441.

[60] Peter C Rigby. 2012. Open source peer review–lessons and recommendations for closed source. (2012).

[61] Peter C Rigby, Daniel M German, and Margaret-Anne Storey. 2008. Open source software peer review practices: a case study of the apache server. In *ICSE'08*. 541–550.

[62] Peter C Rigby and Margaret-Anne Storey. 2011. Understanding broadcast based peer review on open source software projects. In *ICSE'11*. 541–550.

[63] Shade Ruangwan, Patanamon Thongtanunam, Akinori Ihara, and Kenichi Matsumoto. 2018. The impact of human factors on the participation decision of reviewers in modern code review. *Empirical Software Engineering* (2018), 1–44.

[64] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: a case study at google. In *ICSE-SEIP'18*. 181–190.

[65] Davide Spadini, Maurício Aniche, Margaret-Anne Storey, Magiel Bruntink, and Alberto Bacchelli. 2018. When Testing Meets Code Review: Why and How Developers Review Tests. In *ICSE'18*. 677–687.

[66] Michael Stein, John Riedl, Sören J Harner, and Vahid Mashayekhi. 1997. A case study of distributed, asynchronous software inspection. In *ICSE'97*. 107–117.

[67] Xiaobing Sun, Bixin Li, Chuanqi Tao, Wanzhi Wen, and Sai Zhang. 2010. Change impact analysis based on a taxonomy of change types. In *COMPSAC'10*. 373–382.

[68] Xiaobing Sun, Bixin Li, Wanzhi Wen, and Sai Zhang. 2013. Analyzing impact rules of different change types to support change impact analysis. *SEKE'13* 23, 03 (2013), 259–288.

[69] Andrew Sutherland and Gina Venolia. 2009. Can peer code reviews be exploited for later information needs?. In *ICSE-Companion'09*. 259–262.

[70] E Burton Swanson. 1976. The dimensions of maintenance. In *ICSE'76*. 492–497.

[71] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. 2015. Online defect prediction for imbalanced data. In *ICSE'15*. 99–108.

[72] Xinye Tang, Song Wang, and Ke Mao. 2015. Will this bug-fixing change break regression testing?. In *ESEM'15*. 1–10.

[73] Chakkrit Tantithamthavorn, Ahmed E Hassan, and Kenichi Matsumoto. 2018. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *arXiv preprint arXiv:1801.10269* (2018).

[74] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. Automated parameter optimization of classification techniques for defect prediction models. In *ICSE'16*. 321–332.

[75] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. 2012. How do software engineers understand code changes?: an exploratory study in industry. In *FSE'12*. 51.

[76] Yida Tao and Sunghun Kim. 2015. Partitioning composite code changes to facilitate code review. In *MSR'15*. 180–190.

[77] Patanamon Thongtanunam, Raula Gaikovina Kula, Ana Erika Camargo Cruz, Norihiro Yoshida, and Hajimu Iida. 2014. Improving code review effectiveness through reviewer recommendations. In *CHASE'14*. 119–122.

[78] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. 2016. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *ICSE'16*. 1039–1050.

[79] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. 2015. Who should review my code? A file location-based code-reviewer recommendation approach for modern code review. In *SANER'15*. 141–150.

[80] Lawrence G Votta Jr. 1993. Does every inspection need a meeting? *ACM SIGSOFT Software Engineering Notes* 18, 5 (1993), 107–114.

[81] Junjie Wang, Qiang Cui, Song Wang, and Qing Wang. 2017. Domain adaptation for test report classification in crowdsourced testing. In *ICSE-SEIP'17*. 83–92.

[82] Junjie Wang, Mingyang Li, Song Wang, Tim Menzies, and Qing Wang. 2019. Images dont lie: Duplicate crowdtesting reports detection with screenshot information. *IST'19* 110 (2019), 139–155.

[83] Junjie Wang, Song Wang, and Qing Wang. 2018. Is there a golden feature set for static warning identification?: an experimental evaluation. In *ESEM'18*. 17.

[84] Song Wang, Chetan Bansal, Nachiappan Nagappan, and Adithya Abraham Philip. 2019. Leveraging Change Intents for Characterizing and Identifying Large-Review-Effort Changes. In *PROMISE'19*. 46–55.

[85] Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. 2018. Deep semantic feature learning for software defect prediction. *IEEE Transactions on Software Engineering* (2018).

[86] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *ICSE'16*. 297–308.

[87] Arnold D Well and Jerome L Myers. 2003. *Research design & statistical analysis*. Psychology Press.

[88] Mairieli Wessel, Igor Steinmacher, Igor Wiese, and Marco A Gerosa. 2019. Should I Stale or Should I Close? An Analysis of a Bot that Closes Abandoned Issues and Pull Requests. In *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*. IEEE, 38–42.

[89] Jerod W Wilkerson. 2012. A software change impact analysis taxonomy. In *ICSM'12*. 625–628.

[90] Marvin Wyrich and Justus Bogner. 2019. Towards an autonomous bot for automatic source code refactoring. In *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*. IEEE, 24–28.

[91] Meng Yan, Ying Fu, Xiaohong Zhang, Dan Yang, Ling Xu, and Jeffrey D Kymer. 2016. Automatically classifying software changes via discriminative topic model: Supporting multi-category and cross-project. *JSS'16* 113 (2016), 296–308.

[92] Feng Zhang, Quan Zheng, Ying Zou, and Ahmed E Hassan. 2016. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *ICSE'16*. 309–320.

[93] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. 2010. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *ICST'10*. 421–428.

[94] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting defects for eclipse. In *PROMISE'07*. 9–9.

[95] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. 2005. Mining version histories to guide software changes. *TSE'05* 31, 6 (2005), 429–445.