Revisiting the Effect of Branch Handling Strategies on Change Recommendation

Keisuke Isemoto
k_isemoto@se.c.titech.ac.jp
Tokyo Institute of Technology
Meguro-ku, Tokyo, Japan

Takashi Kobayashi
tkobaya@c.titech.ac.jp
Tokyo Institute of Technology
Meguro-ku, Tokyo, Japan

Shinpei Hayashi
hayashi@c.titech.ac.jp
Tokyo Institute of Technology
Meguro-ku, Tokyo, Japan

ABSTRACT

Although literature has noted the effects of branch handling strategies on change recommendation based on evolutionary coupling, they have been tested in a limited experimental setting. Additionally, the branches characteristics that lead to these effects have not been investigated. In this study, we revisited the investigation conducted by Kovalenko *et al.* on the effect to change recommendation using two different branch handling strategies: including changesets from commits on a branch and excluding them. In addition to the setting by Kovalenko *et al.*, we introduced another setting to compare: extracting a changeset for a branch from a merge commit at once. We compared the change recommendation results and the similarity of the extracted co-changes to those in the future obtained using two strategies through 30 open-source software systems. The results show that handling commits on a branch separately is often more appropriate in change recommendation, although the comparison in an additional setting resulted in a balanced performance among the branch handling strategies. Additionally, we found that the merge commit size and the branch length positively influence the change recommendation results.

CCS CONCEPTS

• **Software and its engineering → Software version control**; **Software evolution**;

KEYWORDS

Change recommendation, evolutionary coupling, version control, Git

ACM Reference Format:
Keisuke Isemoto, Takashi Kobayashi, and Shinpei Hayashi. 2022. Revisiting the Effect of Branch Handling Strategies on Change Recommendation. In *30th International Conference on Program Comprehension (ICPC '22), May 16–17, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3524610.3527870

1 INTRODUCTION

As software systems evolve, the amount of codes and the dependencies between the codes increase, thus making the systems more complex. As their complexity increases, it is difficult for developers to understand the extent and effect of their changes. Therefore, necessary changes can leak out, causing bugs in the source code.

Dependencies among source files can be identified based on information about the evolutionary processes of the source files. By using information such as commits recorded in version control systems, we can estimate dependencies that cannot be obtained through static or dynamic analysis, independent of the used programming language.

One approach is change recommendation based on evolutionary coupling [14, 20]. The co-change information can be obtained from the revision history of the source files that have been changed in the same revision. By applying association rule mining to the co-change information, the dependencies among source files can be extracted as association rules such as "*when a file f_1 is changed, another file f_2 should also be changed.*"

Researchers have noted that there are various factors that affect the results of change recommendations based on co-change information [11, 12]. Kovalenko *et al.* reported the influence of branches in history [8]. They compared two history extraction approaches with different branch handling strategies, and reported that obtaining co-change information from commits on a branch slightly improved the recommendation performance compared to omitting to obtain it from history.

However, unlike the result obtained by Kovalenko *et al.* [8], another possible approach when extracting co-change information from history is to obtain such co-change information collectively from a merge commit without traversing commits on a branch. A similar study conducted by Miura *et al.* [9] concluded that changes should be summarized by *work item* in co-change analysis, and it is unclear how the changes on a branch should be handled when applying a co-change analysis. Miura *et al.* investigated the co-change information extracted from the version history of open-source software systems, and found that the changes in commits identified as following the same work, *i.e.*, those related to the same issue, should be grouped together to avoid missing co-change information [9]. Branches in version control systems are primarily used for achieving specific tasks, such as bug fixes and feature additions [21], and changes on a branch are comparable to the same work. Based on this observation, an approach skipping commits on a branch but obtaining their co-change information from a merge commit is possible, and the appropriate treatment of branches when extracting changes during change recommendation is unclear.

Additionally, because branches are an essential element of software development, it is important to investigate the influence of branches in applying repository mining methods to actual repositories. Currently, Git is widely used as a representative version

ICPC '22, May 16–17, 2022, Virtual Event, USA

ACM ISBN 978-1-4503-9298-3/22/05…$15.00
https://doi.org/10.1145/3524610.3527870

control system. In projects that use Git, change management using branches is widely practiced [3]. The branches in Git are easy to create, can be used for tasks such as fixing bugs or adding features [21], and can be used to manage each release. Branch usage also affects the granularity of commits and change recommendations; hence, branches need to be handled appropriately depending on the characteristics of the target repository. Additionally, investigating the types of branches that have affected the change recommendation should be helpful in implementing repository mining methods in actual development.

In this study, we set the following research questions (RQs) to identify the appropriate method for handling branches in change recommendations.

- RQ_1: Does the experimental setting of the study on branch handling strategies influence the change recommendation performance?
- RQ_2: Which branches characteristics affect the performance of change recommendations?
  - $RQ_{2.1}$: What is the relationship between the branch characteristics and change recommendation results?
  - $RQ_{2.2}$: What is the relationship between the branch characteristics and the co-changes extracted from commits in a branch and those from a merge commit?

To answer RQ_1, we compare the change recommendation results with the two branch handling strategies proposed by Kovalenko *et al.* [8] to investigate the effect of the experimental setting that might lead to the results. We used more repositories than Kovalenko *et al.* as well as two implementations. The first implementation reproduced Kovalenko *et al.*'s study, and the other implementation changed the handling of merge commits. We compared the results with different branch handling strategies, and analyzed them through validation based on more evaluation metrics than Kovalenko *et al.*

In RQ_2, we investigate the branches characteristics that affect the change recommendations. However, because differences exist among the branches, defining a uniform treatment for the branches might not be possible. Therefore, we investigate how each branch characteristic affects the change recommendation, such as the length of branches and the size of the merge commits. In $RQ_{2.1}$, we analyze the differences in the recommendation results of different branch handling strategies for each branch characteristic. Because the analysis in answering $RQ_{2.1}$ can be affected by the configuration of the change recommendation method used, in $RQ_{2.2}$ we also analyze the difference in co-change information directly between the commits on a branch and those in a merge commit for each branch characteristic.

The rest of this paper is organized as follows. We first briefly explain the change recommendation approach based on evolutionary coupling in the next section. In Section 3, we explain the study by Kovalenko *et al.* and introduce RQs. We respond to the introduced RQs in Section 4. Discussion and implications, and threats to validity are described in Sections 5 and 6, respecctively. Sections 7 and 8 are for related work and concluding remarks.

2 CHANGE RECOMMENDATION BASED ON EVOLUTIONARY COUPLING

Change recommendation approaches based on evolutionary coupling [8, 14, 20] take a set of files as an input query, and output a ranked list of files as recommendations to be fixed in addition to the input set of files. In this study, we use the "other files" algorithm, which is a change recommendation algorithm implemented by Kovalenko *et al.* [7]. In the following, we briefly explain this as a typical method of co-change-based change recommendation.

This change recommendation algorithm comprises the following three steps:

(1) Collection of commits to be used.
(2) Application of the *Apriori* algorithm [1] to the changesets extracted from the collected commits to generate *association rules*.
(3) Construction of a *recommendation list* of files using the obtained association rules.

The first step is to collect commits from the target change repository using the given branch handling strategy. Each commit is regarded as a *changeset*, *i.e.*, a set of files that were modified at the commit. Initially, commits that meet the following inclusion criteria are collected by following the given branch handling strategy.

- Only commits that contain at least one file in the query are included.
- Assuming that changed files in large commits may not have meaningful relationships [11], commits with more than 10 changed files are excluded.

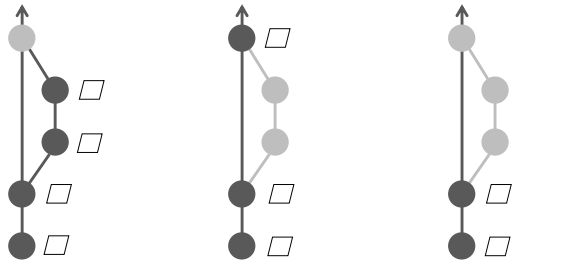
Finally, at most, 100 latest files are passed onto the next step.

The second step is to generate *association rules*. The set of changed files in each collected commit forms a *transaction*; *i.e.*, each pair of files in the changeset is changed together. The transaction database T, which is a set of transactions obtained from the commits, is the input for the Apriori algorithm for generating association rules as the recommendation results. Each association rule follows the form of "*condition part* $\rightarrow$ *conclusion part*" ($x \rightarrow y$), where each part is a set of files. The quality of an association rule is measured by the *support* and *confidence* metrics:

$$\begin{aligned} support(x) &= |\{\, t \in T \mid x \subseteq t \,\}|/|T|, \\ support(x \rightarrow y) &= support(x \cup y), \\ confidence(x \rightarrow y) &= support(x \rightarrow y)/support(x). \end{aligned}$$

In the Apriori algorithm, two parameters, the minimum support (*minsup*) and confidence (*minconf*), are employed to generate the association rules that meet the conditions from the given transaction database. We used the same parameters as those in Kovalenko *et al.*: (*minsup*, *minconf*) = (0.1, 0.1). The generated association rules having more than one file at the conclusion part are filtered out. Finally, at most, 10 association rules having top support values are passed onto the next step.

The third step is to produce a *recommendation list* by applying the given query to the obtained association rules. If all the files in the condition part of an association rule are included in the query, the file at the conclusion part of the rule is adopted as a recommendation result. The results in the recommendation list are ranked in the order of support of the adopted rules.



(a) *Full* (b) *First-Parent*$_{\text{merge}}$ (w/ merging changes) (c) *First-Parent*$_{\text{no\_merge}}$ (w/o merging changes)

Figure 1: Branch handling strategies.

In this change recommendation approach, the number of recommendations is kept low, because the association rules to be used are narrowed down before making the recommendation. Because the recommendation is made from at most 10 association rules, the resulting recommended files are also at most 10.

3 BRANCH HANDLING STRATEGIES

Kovalenko *et al.* [8] studied the effect of handling branches when applying repository mining tasks to a Git repository. They investigated three repository mining applications: recommending reviewers [2], bug prediction [4], and change recommendation [20]. They compared the results after applying two branch handling strategies, *First-Parent* and *Full*, to three repository mining methods, including change recommendation, to evaluate the importance of each branch handling strategy. Figure 1 shows each branch handling strategy. The commits with a parallelogram are the targets to be extracted in the branch handling strategy used. The *First-Parent* strategy selects commits by tracing the first parent when a merge commit occurs and omits the changes obtained from non-main branches, whereas the *Full* strategy selects commits that are reachable by tracing all parent commits. The *Full* strategy excludes the merging change in merge commits, as they are a union of all changes on the associated branch. The *First-Parent* strategy defined by Kovalenko *et al.* also excludes the merging change in merge commits to compare the effect of using the changes in branches. For clarity, we named this strategy as *First-Parent*$_{\text{no\_merge}}$. Kovalenko *et al.* compared these two branch handling strategies, and demonstrated that the *Full* strategy exhibited a slightly better recommendation performance.

Nevertheless, there is another possible approach for handling changes in a branch: extracting all the changes from a merge commit to obtain co-change information of a branch collectively. We name this strategy *First-Parent*$_{\text{merge}}$. In the *Full* strategy, each commit on a branch is treated individually. In the *First-Parent*$_{\text{merge}}$ strategy, the changes on the branch are combined into a merge commit, and a single changeset from the merge commit is used. In contrast to the *First-Parent*$_{\text{merge}}$ strategy, the *First-Parent*$_{\text{no\_merge}}$ strategy excludes merge commits with some exceptions; only the cases where the merge commits contained additional changes are included. The difference of handling merge commits might affect the change recommendation performance, wherein the performance of the *Full* strategy tended to be higher.

Table 1: Target repository sets

Repository sets	# repositories	Avg. # commits
$Apache_K$	10	3,955.6
$Eclipse_K$	8	15,582.9
$Apache_M$	12	2,829.6

Therefore, in RQ_1, we compare the performance of the change recommendation results between *Full* and *First-Parent*$_{\text{no\_merge}}$ and between *Full* and *First-Parent*$_{\text{merge}}$. Through experiments with the two combinations, we confirmed the influence of missing changes on the branches as Kovalenko *et al.* studied. By experimenting with *Full* and *First-Parent*$_{\text{no\_merge}}$, we can experiment with almost the same conditions as Kovalenko *et al.*, leading to a fairer comparison. Additionally, RQ_2 considers the missing changes in the branches of the *First-Parent* strategy to be unsuitable for comparing the branch handling; hence, we target the recommendation results between the *Full* and *First-Parent*$_{\text{merge}}$ strategies.

4 EMPIRICAL STUDIES

4.1 Dataset Preparation

In this study, we used the repository sets of the Apache Software Foundation (ASF) and Eclipse ecosystems used by Kovalenko *et al.* [8] ($Apache_K$ and $Eclipse_K$, respectively) and that of the ASF ecosystem used by Miura *et al.* [9] ($Apache_M$), comprising 30 repositories in total. Table 1 presents the statistics of the used repository sets. Note that two repositories from the Eclipse ecosystem, which failed to clone, and two repositories from ASF, which were unaffected by the branch handling strategy selection, were excluded.

4.2 RQ_1: Does the experimental setting of the study on branch handling strategies influence the change recommendation performance?

4.2.1 Motivation. Kovalenko *et al.* [8] compared the change recommendation results using two branch handling strategies, and they reported that the *Full* strategy slightly outperformed the *First-Parent*$_{\text{no\_merge}}$ strategy. In contrast, according to Miura *et al.* [9], the co-change information should be handled at the task (work item) level in the evolutionary coupling analysis because the revision (commit) level analysis fails to extract the co-change information over revisions within the task. If we assume that the changes made on a single branch are all related to a single task, there may lead to another strategy to extract all the changes in a branch from a merge commit at once: *First-Parent*$_{\text{merge}}$. Miura *et al.* [9] found that missing co-changes could be prevented by employing the task level change treatment, while Kovalenko *et al.* found that the *Full* strategy, which treats commits on a branch separately, produced both a higher recommendation success rate and higher number of recommendations. Because the branches are mainly used as implementing features, *i.e.*, as feature branches [21], if we consider that the changes on a branch are related to a task, summarizing them at the task level is similar to merging the changes on a branch. Although the study by Kovalenko *et al.* compared the performance of the *Full* and *First-Parent*$_{\text{no\_merge}}$ strategies, a different comparison

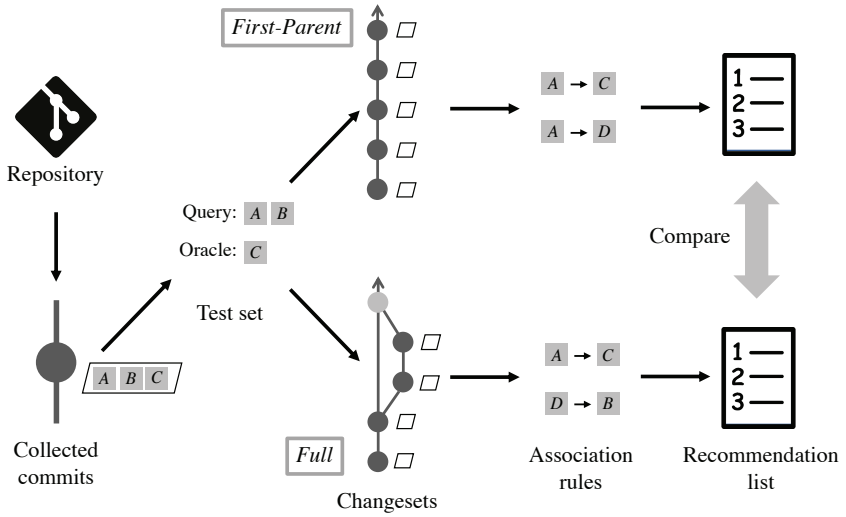


Figure 2: Comparison of two change recommendation results obtained using different branch handling strategies.

with *First-Parent*$_{\text{merge}}$ can lead to understanding branch handling strategies more on their change recommendation performance. The *First-Parent*$_{\text{no\_merge}}$ strategy skips many changes on the branches, which may have resulted in the slightly higher performance of the *Full* strategy. If the performance of the *Full* strategy was higher because of the co-change information being excluded, it does not necessarily mean that the performance of the *Full* strategy is better than that of the *First-Parent*$_{\text{merge}}$ strategy in general. Therefore, we experimented with an implementation that follows Kovalenko *et al.* [8], denoted as the *First-Parent*$_{\text{no\_merge}}$ strategy. We also used another implementation to process all the changes in the collected merge commits, denoted as the *First-Parent*$_{\text{merge}}$ strategy.

Additionally, Kovalenko *et al.* used the success rate and number of recommendations for the evaluation metrics, which might be room for improvement. For example, the success rate and number of recommendations can be kept high even if the number of wrong recommendations increases, while increasing the number of recommendations. We used the mean average precision (MAP) [11–14] and the number of wins and losses [13], which are also used in other change recommendation studies, to compare the performance of the branch handling strategies. The detailed definitions of evaluation metrics will be explained later.

In addition, we compare the results for the repository sets used by Kovalenko *et al.* and those used by Miura *et al.*, to investigate the influence of repository selection in the experiments.

4.2.2 Study Design. We compare the recommendation results between the *Full* and *First-Parent*$_{\text{no\_merge}}$ strategies (*Imp*$_{\text{no\_merge}}$) and those between the *Full* and *First-Parent*$_{\text{merge}}$ strategies (*Imp*$_{\text{merge}}$). An overview of the conducted experiment is shown in Figure 2. For each repository and each commit to be experimented upon, we created a test set by selecting one file in the changeset as the oracle and using the remaining files as the query. With the prepared test set, a change recommendation was performed for each branch handling strategy. Finally, the results based on the branch handling strategies are compared and summarized.

We selected the commits to be used that satisfy the following conditions, which are the same as those used by Kovalenko *et al.*

- At most, 10 files are changed in the commit.
- Different branch handling strategies produce different changesets to ensure that the comparison is meaningful [8].
- At least five changesets are collected in either of the branch handling strategies to be able to produce meaningful rules [8].
- At least one association rule is generated in either of the branch handling strategies [8].

Note that *Imp*$_{\text{no\_merge}}$ and *Imp*$_{\text{merge}}$ have the following three minor differences in their processing, in addition to the branch handling strategy used.

- **Different restrictions on commit extraction.** The commit extraction process is eligible for generating association rules. In *Imp*$_{\text{no\_merge}}$, up to 100 commits, containing query changes as well as at most 10 changed files in descending order of the authored date (newer to older), are targeted. In *Imp*$_{\text{merge}}$, for each change targeted in the query, it extracts at most 100 commits using the `git-log` command with respect to the structure of the commit graph, and commits comprising more than 10 changed files are filtered out. *Imp*$_{\text{no\_merge}}$ fetches commits until it reaches 100, or there are no additional commits with changes matching the query; meanwhile, *Imp*$_{\text{merge}}$ fetches up to 100 commits from the sliced history of each file in the query, and extracts commits comprising at most 10 file changes. We made this change for efficiency in *Imp*$_{\text{merge}}$, as we needed to process large-scale test sets.
- **Different target repository sets**. The implementation of Kovalenko *et al.* excludes repositories containing more than 10,000 commits. In *Imp*$_{\text{no\_merge}}$, we also limited the repositories that the original implementation of Kovalenko *et al.* could process, because we wanted to compare the results of our *Imp*$_{\text{no\_merge}}$ and Kovalenko *et al.*'s implementation to identify the differences.

Table 2: Summary of differences between two studies

	Experimental setting by Kovalenko *et al.*	Replication setting of Kovalenko *et al.* in this paper ($Imp_{\text{no\_merge}}$)	Experimental setting using merging changes (Imp_{merge})
Branch handling strategies	*Full* vs. *First-Parents*$_{\text{no\_merge}}$	*Full* vs. *First-Parents*$_{\text{no\_merge}}$	*Full* vs. *First-Parents*$_{\text{merge}}$
# target repositories	20	24	30
Evaluation metrics	Success rate, # recommendations	Success rate, # recommendations, MAP, # wins in AP	Success rate, # recommendations, MAP, # wins in AP

Effect of implementation (between *Full* vs. *First-Parents*$_{\text{no\_merge}}$ and *Full* vs. *First-Parents*$_{\text{merge}}$)

Effect of repositories (between 20 and 24)

Effect of metrics (between Success rate, # recommendations and Success rate, # recommendations, MAP, # wins in AP)

- **Different adjustment for the number of association rules.** The implementation of Kovalenko *et al.* ($Imp_{\text{no\_merge}}$) included adjusting the number of commits to be used to generate the association rules and the number of association rules to be extracted to ensure that the results of both branch handling strategies produce the same number of recommendations for fairness. This is an unnecessary process in the actual change recommendation process. To align with the actual change recommendation process, Imp_{merge} did not follow such an adjustment. In contrast, $Imp_{\text{no\_merge}}$ continued to perform the same adjustment because this study focused more on replicating the experiments of Kovalenko *et al.*

The obtained recommendation results are thus compared with several evaluation metrics.

- **Success rate** [8] classifies the recommendation results into *success*, *failure*, and *no prediction*, and summarizes their ratios for each branch handling strategy. Note that we have changed the classification criteria differently than Kovalenko *et al.* They regarded a recommendation as a *failure* only when the files included in the query were recommended, whereas we regarded such cases as *no prediction*. In addition to the success rate, the *failure rate* and *no prediction rate* were computed.
- **MAP** [13] is used in two ways. MAP_{all} strictly treats the average precision (AP) of the no-recommendation result as 0, as it assumes that the recommender should always produce at least one recommendation file. In contrast, MAP_{app} excludes such no-recommendation results, as it allows the recommendation system to skip its recommendation results.
- **Number of wins/losses** [13] is calculated by determining the win or loss for each test set. A strategy wins if it produces a higher AP than the other strategy, or if it does not make a false recommendation when the APs of both strategies are 0.

Table 2 summarizes the differences between the study conducted by Kovalenko *et al.* and the results from answering RQ_1.

4.2.3 Results.

Effect of repository selection. We investigated the effect of target repository selection on the performance of the change recommendation. The results for each repository set are shown in Table 3 using the same format as the table in Kovalenko *et al.* 's study, whose columns indicate the implementation used, strategy used, number of test sets to be used (event count), average numbers of recommendations and rules, and evaluation metric values. In $Imp_{\text{no\_merge}}$, both the number of recommendations and success rate were higher than those when using the *Full* strategy, except for the number of recommendations in $Eclipse_K$. The results indicate that using different repository sets ($Apache_M$), rather than the ones used by Kovalenko *et al.* ($Apache_K$ and $Eclipse_K$), did not change the trend of the success rate. Hence, we conclude that *the difference between the two studies was not due to the repository selection.*

Effect of evaluation metrics. To investigate the influence of the evaluation metrics to be used, the results of the success rate, MAP_{all}, MAP_{app}, and the number of wins and losses for all repositories are computed, as shown in Table 4. The rightmost two columns show the number of wins and draws for each branch handling strategy. For $Imp_{\text{no\_merge}}$, *Full* performed better in most evaluation metrics; however, *First-Parent* had more number of wins. *First-Parent* had a higher number of wins for $Imp_{\text{no\_merge}}$ because the number of false recommendations increased along with the number of recommendations in *Full*. We suspect that this is due to the missing changes on the branch in the *First-Parent* strategy, which reduced the variation of recommendations and number of recommendations.

Table 5 summarizes the number of repositories that performed better than the opponent strategy in terms of the success rate, MAP_{all}, and the number of wins/losses. To count the winners in terms of the success rate, we considered it a draw if the rate was equal to that of the other strategy. To count the winners in terms of MAP_{app}, we used the Wilcoxon signed-rank test for the distribution of AP values. The winner was decided when we confirmed statistical significance at the 5% significance level. If we could not conclude statistical significance, we considered such cases as draws. Consequently, for the success rate and MAP_{all}, the performance of the *Full* strategy tended to be higher in $Imp_{\text{no\_merge}}$. The *Full* strategy produced more winning repositories in terms of the number of wins.

The $Imp_{\text{no\_merge}}$ results also indicate that the *Full* strategy produced more false recommendations. However, the performance of the *Full* strategy tended to be higher, and we concluded that the effect of the evaluation metrics was small.

Effect of not using changes in branches. The results of $Imp_{\text{no\_merge}}$ and Imp_{merge} in the tables above were compared to determine the effect of skipping information on the branches. In all results, the branch handling strategy that resulted in higher performance tended to vary. In the results of Table 3, the branch handling strategies that exhibited higher performance differed depending on the repository set used. The *First-Parent* strategy produced a higher recommendation count and success rate for $Apache_K$, whereas the *Full* strategy produced a higher recommendation count and success rate for $Eclipse_K$. For $Apache_M$, the number of recommendations was higher when using the *First-Parent* strategy, and the success rate was higher when using the *Full* strategy. The magnitude of the difference was smaller than that for $Imp_{\text{no\_merge}}$. In Table 4, the difference between the branch handling strategies became smaller in MAP_{all}, and the draw rate in terms of the number of wins increased from 84.7% (36,708/43,329) to 89.9% (130,809/145,529). Moreover,

Table 3: Success, failure, and no prediction rates per repository sets

Implementation	Repositories	Strategy	Events count	Recommendations (average)	Rules (average)	Success rate	Failure rate	No prediction rate
$Imp_{\text{no\_merge}}$	$Apache_K$	*Full*	25,588	**1.005**	7.763	**0.179**	0.438	0.383
		First-Parent$_{\text{no\_merge}}$	25,558	0.954	7.763	0.167	0.428	0.405
	$Eclipse_K$	*Full*	4,405	0.775	6.862	**0.156**	0.372	0.472
		First-Parent$_{\text{no\_merge}}$	4,405	**0.788**	6.862	0.148	0.383	0.470
	$Apache_M$	*Full*	7,063	**0.726**	6.491	**0.104**	0.395	0.501
		First-Parent$_{\text{no\_merge}}$	7,063	0.661	6.491	0.099	0.359	0.542
Imp_{merge}	$Apache_K$	*Full*	33,728	0.787	6.808	0.143	0.350	0.507
		First-Parent$_{\text{merge}}$	33,728	**0.823**	7.172	**0.147**	0.365	0.488
	$Eclipse_K$	*Full*	5,994	**0.740**	6.517	**0.150**	0.350	0.499
		First-Parent$_{\text{merge}}$	5,994	0.739	6.792	0.148	0.349	0.503
	$Apache_M$	*Full*	105,807	0.739	5.391	**0.165**	0.319	0.515
		First-Parent$_{\text{merge}}$	105,807	**0.745**	5.516	0.163	0.325	0.512

Table 4: Study results

Implementation	Strategy	Success rate	MAP_{all}	MAP_{app}	# wins	# draws
$Imp_{\text{no\_merge}}$	*Full*	0.162	0.144	0.247	3,214	36,708
	First-Parent$_{\text{no\_merge}}$	0.152	0.135	0.241	3,407	
Imp_{merge}	*Full*	0.160	0.143	0.294	7,848	130,809
	First-Parent$_{\text{merge}}$	0.159	0.143	0.289	6,872	

Table 5: Number of repositories of higher performance

Implementation	Metric	# *Full* wins	# draws	# *First-Parent* wins
$Imp_{\text{no\_merge}}$	Success rate	15	7	2
	MAP_{all}	8	16	0
	# wins	13	2	9
Imp_{merge}	Success rate	15	3	12
	MAP_{all}	5	20	5
	# wins	16	1	13

Table 5 shows that Imp_{merge} demonstrated better equilibrium than $Imp_{\text{no\_merge}}$ in terms of the number of repositories that the *Full* and *First-Parent* strategies produced higher performance.

In summary, the results of Kovalenko *et al.* [8], which showed that the change recommendation performed slightly better when commits on a branch were handled, were possibly caused by missing change information on branches.

> The change recommendation performance could be affected by the experimental setting in handling merging changes. Regardless of the variations of the target repository sets and the evaluation metrics, the *Full* strategy tended to demonstrate better performance than *First-Parent* strategy in $Imp_{\text{no\_merge}}$; such variations did not affect the conclusion of Kovalenko *et al.* However, this tendency was balanced when using Imp_{merge}, which includes merging changes when using the *First-Parent* strategy.

4.3 RQ_2: Which branches characteristics affect the performance of change recommendations?

4.3.1 Motivation. Branches can be used in multiple ways, and different repositories use them differently. Although most branches are used to implement features or fix bugs [21], they are also used to manage releases. For example, several repositories of interest in this study, such as `apache/accumulo` or `apache/cassandra`, prepare branches for each version, and developers work on them in parallel. Even if a branch is used to implement a feature, the frequency of creating a branch and/or granularity of commits and features might differ depending on the commit policy of the project.

Depending on the characteristics of each branch, there may be different ways of handling the branch that is appropriate for extracting changes in the change recommendations. For example, if changes are frequently split into smaller fragments and commits are kept fine-grained on a branch in a project, it might be better to treat them as a single merge commit, rather than as multiple smaller commits. In contrast, in the case of making multiple large changes on a branch and merging them, it might be better to treat each commit on the branch separately to collect changesets for the change recommendation and avoid unnecessary coupling.

Therefore, in answering RQ_2, we examine the influence of the following two characteristics of a merged branch on the change recommendation performance.

- **Branch length** is the number of commits in the path of a branch. It counts the commits between the *merge commit* and the *merge base*, *i.e.*, the common parent of the merged branch and the parent branch. An illustrative example is shown in Figure 3, which shows the computation of the

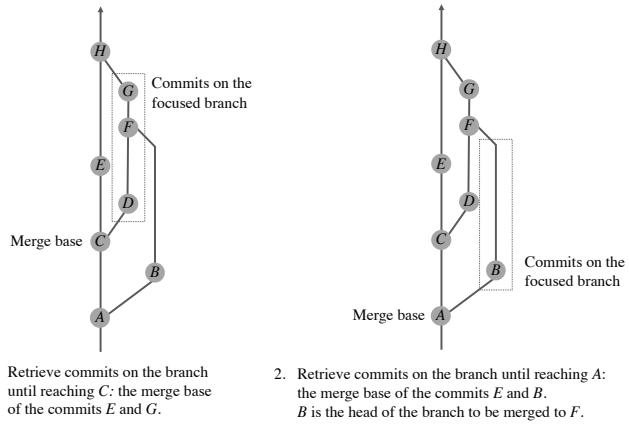


Figure 3: Computation of branch length.

length of the branch merged at commit H. First, the merge base of this branch, C, was computed as the common parent of the head commit of the sinking branch of merge E and source branch of the merge G. The commits on the target branch until the merge base is read, *i.e.*, G, F, and D, are retrieved. When finding another merge commit during this retrieval, we apply this process recursively. Because commit F is a merge commit on the acquired branch, we compute the commits on the branch to be merged at F up to merge base A. Commit B is newly retrieved. Finally, all meaningful commits to be merged at commit H, excluding merge commits, are G, D, and B, and its branch length is computed as 3.

- **Merge commit size** indicates the size of the changeset of a merge commit, *i.e.*, the number of files that have been changed by the merge commit.

We divided this RQ into the following two sub questions:

- $RQ_{2.1}$: What is the relationship between the branch characteristics and change recommendation results?
- $RQ_{2.2}$: What is the relationship between the branch characteristics and the co-changes extracted from commits in a branch and those from a merge commit?

In $RQ_{2.1}$, we examine, for each branch characteristic, the trend in change recommendation results for each characteristic of the partial change history used to generate association rules caused by a branch. In contrast, $RQ_{2.2}$ examines the accuracy of the co-change information of commits on the branch, and the merge commits indicate future changes for each branch characteristic. The analysis for $RQ_{2.1}$ focuses on the branch that caused the difference in commits used for change recommendation, whereas the analysis for $RQ_{2.2}$ directly compares the co-change information of commits on the branch and merge commits. Therefore, the analysis for $RQ_{2.2}$ was unaffected by the implementation of the change recommendation method.

4.3.2 Study Design of $RQ_{2.1}$. We investigated the recommendation results for each branch characteristic used in association rule generation in the change recommendation. The commits used to generate the association rules were replaced by switching the branch handling strategy used. From the replaced commits, we could identify the merge commit that caused the replacement and the commits on the branch that were additionally captured by the replacement. We also analyzed the trend of the recommendation results for those characteristics. Because there may be more than one merge commit that caused the replacement, we compared two typical cases to determine if a similar tendency could be obtained: 1) the results of a single commit and 2) those of six or more commits. Boundary six was used for the third quartile of the number of merge commits that caused the swap of commits in each test set to be analyzed.

To compare the recommendation results obtained from *Full* and *First-Parent*$_{\text{merge}}$ strategies, we used the number of wins/losses [13]. Using the values of the branch characteristics involved, we divided the recommendation results such that the number of samples was as even as possible, and confirmed the branch handling strategy that tends to win. In the case of six or more merge commits, we split the results by the median value to mitigate the effect of outlier instances related to several merge commits.

Note that we limited the number of commits to be investigated to highlight the changes in the commits used between branch handling strategies. Specifically, we targeted commits obtained by limiting the median (53) number of retrieved commits to generate association rules via the *First-Parent* strategy, resulting in 71,832 commits.

4.3.3 Results of $RQ_{2.1}$.

Branch length. The ratio of the winning branch handling strategy to the branch length and the size of the merge commit is shown in Figure 4. Note that tie cases are excluded in this figure. In the case of a single merge commit, the effect on the recommendation result increases with the branch length, from 10 to 24. When the branches become longer than 881, the influence increases. The difference in the win rate between the branch handling strategies, which was less than 2% up to a branch length of 881, becomes 8% higher for the *Full* strategy. If six or more merge commits are handled, the win rate increases until the branch length reaches 5.

Therefore, we conclude that, when the branch is longer, the influence of selecting the branch handling strategy is greater. If there are extremely long branches involved, they might be better treated separately in the commits on the branch, *i.e.*, using the *Full* strategy.

Merge commit size. In the case of a single merge commit, as shown in Figure 4, until the size of the merge commit exceeds 10, there is no winner in approximately 97% of the cases, and the effect of changing the branch handling strategy is small. However, when the size exceeds 10, the win rate increases rapidly from 3% to 10%. We believe that this is because the change recommendation method used in this experiment generates an association rule that excludes commits with more than 11 changes. In the case of six or more merge commits, the rates of winning and losing, rather than draws, tended to increase for winning commits.

As the size of the merge commits increases, the effect of the branch increases. Additionally, in the case of a single branch, we observe a sudden increase in the win rate when the size of the merge commit exceeds 10. This is considered to be an effect of the change recommendation method mentioned above.

4.3.4 Study Design of $RQ_{2.2}$. In answering $RQ_{2.2}$, we compared the co-changed files extracted from a merge commit as a single changeset and those extracted from multiple changesets from the

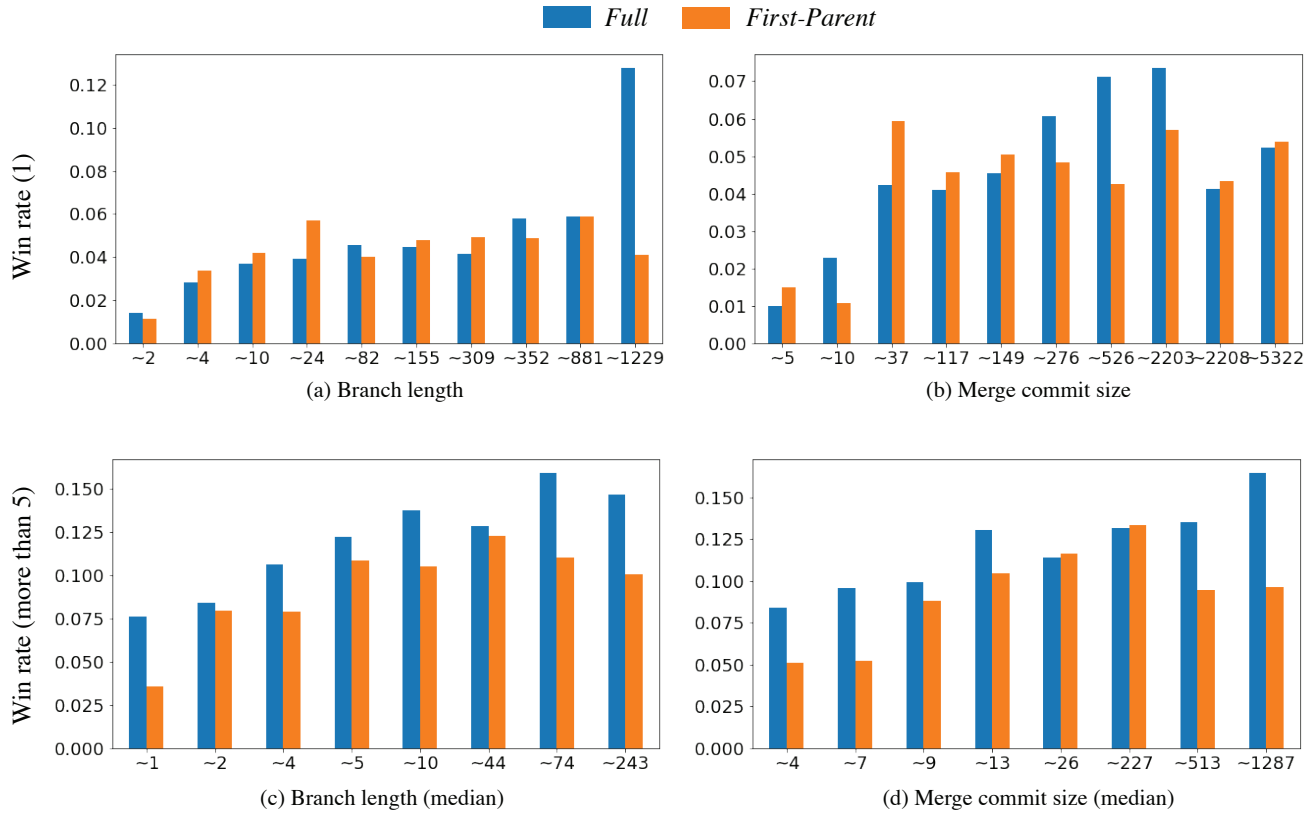


Figure 4: Winning rate of branch handling strategies against branch characteristics.

commits on the related branch. In the analysis, for every single file that is modified by a merge commit or a commit on a branch that is fixed, its co-changed files ($F_{changed}$) are compared. The files that are co-changed with the target file in the reachable future 100 commits from the merge commit are regarded as the oracle (F_{oracle}). By comparing the two file sets, *Precision* is calculated as

$$Precision = \frac{|F_{changed} \cap F_{oracle}|}{|F_{changed}|}.$$

To mitigate the effect of the merge commits containing several files, which should be larger than that of smaller merge commits, we obtained the average for each commit, and determined the distribution and trend for each characteristic of the branch.

We targeted all merge commits on the default branch, *i.e.*, the merge commits that can be reached from the repository's HEAD by following the first parent commit, which can be reachable using both the *First-Parent* and *Full* strategies. We excluded merge commits where the associated branch length is 1, and the change in the changed files in the branch is the same as that in the merge commit, because the difference in the co-change extraction strategies does not affect such merge commits.

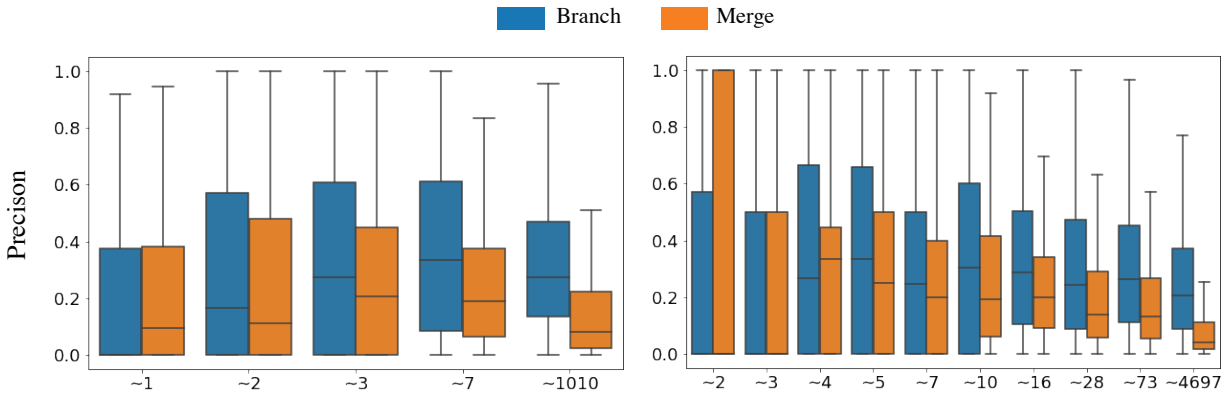
4.3.5 Results of $RQ_{2.2}$.

Branch length. The distribution of the average values per commit of *Precision* for the branch features is shown in Figure 5(a). Except for branch lengths of 1, the distribution exhibits higher precision when using branches than when using merge commits, and the difference between them increases as the branch length increases. In cases with a branch length of 1 or less, the *Precision* was higher when using merge commits because using branches can only outperform if additional changes, or deletion of unnecessary changes, have been made during the merge.

In summary, except for branch lengths 1 or less, the precision values were higher when the changesets were extracted individually with the commits on a branch. When the branch is longer, the difference is larger.

Merge commit size. As shown in Figure 5(b), when the size of the merge commits is small, the median and third quartiles are higher when using merge commits. However, as the size increases, the distribution of precision increases when branches are used. The median and third quartiles are higher when using merge commits, while the merge commit is small. As the merge commit becomes larger, the median becomes higher when using branches. As merge commits become larger, the extracted co-changes are summarized in a merge increase, and the ratio of not useful changes that will not happen in the future increases.

4.3.6 Summary. In both analyses of $RQ_{2.1}$ and $RQ_{2.2}$, when the branch is longer or the merge commit is larger, the evaluation result is better for treating them separately in commits on a branch. In the case of longer branches and larger changes, there might be multiple changes belonging to different intentional tasks, and merging them



(a) Branch length (b) Merge commit size

Figure 5: Distribution of *Precision* over branch characteristics.

together in a single changeset of a merge commit tends to have a negative effect. In the case of shorter branches and smaller changes, a single changeset extracted from a merge commit tends to be better for the source of co-changes. However, these differences were hardly reflected in the recommendation results. We examined the recommendation results where the maximum length of the involved branches was at most 1, and 98% of their cases resulted in a tie, although the *First-Parent* strategy demonstrated a higher performance in general. The need for the branch handling strategy to be changed is minute in repositories where branches are used infrequently, and where they are short, because the influence of only a single short branch is negligible for selecting a branch handling strategy.

A sudden increase in the influence of the branch handling strategies occurred when the number of merges exceeded 10 in the recommendation results. This is because the change recommendation method to be used limited the source commits in generating association rules with at most 10 changes. When analyzing the co-changes ($RQ_{2.2}$), there was no significant differences when the size of the merge commits was approximately 10. The influence of the branch handling strategies increased because the co-change of the merge commits was not handled anymore.

When longer branches and larger merge commits are involved, the effect of the branch handling strategies becomes more significant, and the performance improves when the co-changes are extracted individually from the commits on a branch. In contrast, when shorter branches and smaller merge commits are involved, the performance improves when the co-changes are extracted from a merge commit at once. However, the observed difference was small; the effect was also small. The size parameter of excluding commits in the change recommendation method affected the difference in the recommendation results.

5 DISCUSSION AND IMPLICATIONS

5.1 Discussion

The results of RQ_2 showed that, as the branches become longer and the merge commits become larger, handling co-changes with commits on the branch exhibit better performance. However, if we refer to Figure 4, it is evident that the rate at which the *Full* strategy wins does not increase unilaterally; however, the rate at which the *First-Parent* strategy wins also increases. Therefore, we sampled and investigated a small number (40) of merge commits to determine the co-changes added. Regarding the sampling, we sampled merge commits where the number of co-changes added by the merge is seven or more, to exclude a small number of co-changes that were highly affected by a single co-change and cases where the maximum number of co-changes that a change to three files can have was six or less. A typical case is the co-change with `CHANGE.txt` in `apache/cassandra`, which records the updates. In this repository, 19 out of 20 merge commits corresponded to this. The other repositories also contained three co-changes with configuration files. Additionally, co-changes from newly added changes in the merge commit occurred in 6 of the 40 cases. For merge commits with additional changes, the probability of future co-changes of the merge was higher than that of co-changes on the branch, which is the opposite of the result for the merge commits without additional changes. If there are additional changes in a merge commit, it might be better to handle a single changeset extracted from the merge commit.

5.2 Implications

In change recommendation, it is suggested that handling commits on a branch separately is better, instead of merging them in a merge commit. When the branch is longer, it tends to be better at handling commits individually on the branch. When the branch is shorter, such as those with a length of 1, it is better at handling commits as a merge commit; however, the effect of this treatment is extremely small. Additionally, because long branches contain changes to many different files, there are many opportunities for a single branch to affect the change recommendation. Therefore, when deciding

whether to handle long branches uniformly as multiple commits on a branch or as a single merge commit, the disadvantage of generating co-changes that will not be made in the future leads to a higher risk than the other strategy, which may lack some co-changes.

However, when using a file as a query that has been modified significantly, when branches are rarely used, or when only shorter branches are used in the target repository, the importance of selecting the branch handling strategies decreases.

6 THREATS TO VALIDITY

Internal validity. We studied the change recommendation method based on the "other files" algorithm and the implementation by Kovalenko *et al.* [7]. Because the method was designed to limit its recommendations, the effect of selecting the branch handling strategies might be smaller in observation than in reality.

The analysis conducted in this study assumes that changes on a branch are related to a single task in most cases, which is the most typical usage of a branch. However, this is not always true; the changes in a branch may be related to multiple tasks. In fact, in the repository of `apache/cassandra` in $Apache_K$, commits related to multiple tasks, where multiple issues are linked, were merged at once. Therefore, we believe that the results of this study do not negate the results of Miura *et al.* [9].

External validity. In our study, only the repositories of $Apache_K$ and $Eclipse_K$ used by Kovalenko *et al.* [8] and those of $Apache_M$ used by Miura *et al.* [9], were targeted. The selection of repositories may still be biased; it may have been influenced by their development styles. For example, in `apache/cassandra`, where merging was heavily used, the changes corresponding to an issue were combined into a single commit without recording the details of the fine-grained development history. In such repositories, where the fine-grained development process is performed on branches, using commits on a branch separately, as the source of changesets to be used for the change recommendation, might produce better results.

7 RELATED WORK

Several change recommendation methods that utilize change coupling have been proposed. These include ROSE by Zimmermann *et al.* [20], an FP-Tree-based approach by Ying *et al.* [19], CO-CHANGE by Kagdi *et al.* [6], and TARMAQ by Rolfsnes *et al.* [14]. In these methods, changesets obtained from the commits in a repository are used to generate the recommendation rules. The branch handling strategies studied herein affect the accuracy of these methods.

Moonen *et al.* studied the various factors that affect the change recommendations. They investigated the effect of measures, such as confidence and support, which are used to rank association rules. They also studied the effect of limiting the number of file changes in a commit used to create association rules [11], that of the history length used, and that of using the most recent history [12]. Pugh *et al.* [13] improved the accuracy of change recommendations based on these results to produce shorter partial histories for generating recommendation rules.

Moreover, Rolfsnes *et al.* proposed several approaches to improve the accuracy of change recommendation methods, including the aggregation of multiple recommendation rules [15], filtration of recommendations via machine learning [16], and use of method-level finer-grained co-change information [17]. Mondal *et al.* proposed a combinational approach for analyzing the relationship of program identifiers [10].

Zou *et al.* [21] conducted a study on branch usage in 2,923 repositories, published on GitHub. Further, Shihab *et al.* [18] investigated the influence of branches on software quality for repositories in Microsoft. Bird *et al.* [3] reduced development delays by identifying high-cost branches that require conflict resolution for merging or a long time for validating and deleting branches with high cost, but small contributions. In contrast to their studies, in this study, we analyzed the effect of each branch characteristic on the change recommendations.

8 CONCLUSION

In this study, we analyzed the effect of branches on change recommendation results. In contrast to the study by Kovalenko *et al.* [8] with a slightly higher performance for the *Full* branch handling strategy than the $\textit{First-Parent}_{\text{no\_merge}}$ strategy, the comparison with the $\textit{First-Parent}_{\text{merge}}$ strategy resulted in a balanced performance among the branch handling strategies.

Additionally, the analyses of branch influence on the change recommendation demonstrated that it is better to treat commits on branches separately in the change recommendation. Meanwhile, handling all commits on a branch as a single changeset at once in the related merge commit prevented the leakage of co-changes. Performing this treatment on a large branch caused a negative effect of noise from unnecessary co-changes that will not be changed in the future. We conclude that it is less risky to handle co-changes in the commits on a branch.

Our future work will involve improving the generality of our analysis by experimenting on a wider range of repositories with modern change recommendation methods, such as TARMAQ [14].

The supplemental materials are publicly available [5].

ACKNOWLEDGMENTS

This work was partly supported by JSPS Grants-in-Aid for Scientific Research JP18K11238, JP21K18302, JP21KK0179, and JP21H04877.

REFERENCES

[1] Rakesh Agrawal and Ramakrishnan Srikant. 1994. Fast Algorithms for Mining Association Rules in Large Databases. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB 1994)*. 487–499.
[2] Vipin Balachandran. 2013. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013)*. 931–940.
[3] Christian Bird and Thomas Zimmermann. 2012. Assessing the value of branches with what-if analysis. In *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2012)*. 1–11.
[4] Ahmed E. Hassan. 2009. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*. 78–88.
[5] Keisuke Isemoto, Takashi Kobayashi, and Shinpei Hayashi. 2022. Appendix of "Revisiting the effect of branch handling strategies on change recommendation". Zenodo. https://doi.org/10.5281/zenodo.6402130
[6] Huzefa Kagdi, Shehnaaz Yusuf, and Jonathan Maletic. 2006. Mining sequences of changed-files from version histories. In *Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR 2006)*. 47–53.
[7] Vladimir Kovalenko. 2018. Mining file histories: Should we consider branches?—Online appendix. https://github.com/vovak/branches.

[8] Vladimir Kovalenko, Fabio Palomba, and Alberto Bacchelli. 2018. Mining file histories: Should we consider branches?. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2018)*. 202–213.

[9] Keisuke Miura, Shane McIntosh, Yasutaka Kamei, Ahmed E. Hassan, and Naoyasu Ubayashi. 2016. The impact of task granularity on co-evolution analyses. In *Proceedings of the 9th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2016)*. 1–10.

[10] Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. 2014. Improving the detection accuracy of evolutionary coupling by measuring change correspondence. In *Proceedings of the Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE 2014)*. 358–362.

[11] Leon Moonen, Stefano Di Alesio, David Binkley, and Thomas Rolfsnes. 2016. Practical guidelines for change recommendation using association rule mining. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. 732–743.

[12] Leon Moonen, Thomas Rolfsnes, Dave Binkley, and Stefano Di Alesio. 2018. What are the effects of history length and age on mining software change impact? *Empirical Software Engineering* 23, 4 (2018), 2362–2397.

[13] Sydney Pugh, David Binkley, and Leon Moonen. 2018. The case for adaptive change recommendation. In *Proceedings of the 18th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2018)*. 129–138.

[14] Thomas Rolfsnes, Stefano Di Alesio, Razieh Behjati, Leon Moonen, and Dave W. Binkley. 2016. Generalizing the analysis of evolutionary coupling for software change impact analysis. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*. 201–212.

[15] Thomas Rolfsnes, Leon Moonen, Stefano Di Alesio, Razieh Behjati, and Dave Binkley. 2016. Improving change recommendation using aggregated association rules. In *Proceedings of the 13th IEEE/ACM International Conference on Mining Software Repositories (MSR 2016)*. 73–84.

[16] Thomas Rolfsnes, Leon Moonen, and David Binkley. 2017. Predicting relevance of change recommendations. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. 694–705.

[17] Thomas Rolfsnes, Leon Moonen, Stefano Di Alesio, Razieh Behjati, and Dave Binkley. 2018. Aggregating association rules to improve change recommendation. *Empirical Software Engineering* 23, 2 (2018), 987–1035.

[18] Emad Shihab and Thomas Zimmermann. 2012. The effect of branching strategies on software quality. In *Proceedings of the 6th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2012)*. 301–310.

[19] Annie T.T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. 2004. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering* 30, 9 (2004), 574–586.

[20] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. 2005. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering* 31, 6 (2005), 429–445.

[21] Weiqin Zou, Weiqiang Zhang, Xin Xia, Reid Holmes, and Zhenyu Chen. 2019. Branch Use in Practice: A Large-Scale Empirical Study of 2,923 Projects on GitHub. In *Proceedings of the 19th IEEE International Conference on Software Quality, Reliability and Security (QRS 2019)*. 306–317.