Assessing time-based and range-based strategies for commit assignment to releases

Felipe Curty do Rego Pinto
Instituto de Computação
Universidade Federal Fluminense
Niterói, RJ, Brazil

Bruno Costa
Instituto Federal de Educação,
Ciência e Tecnologia do Rio de Janeiro
Rio de Janeiro, RJ, Brazil

Leonardo Murta
Instituto de Computação
Universidade Federal Fluminense
Niterói, RJ, Brazil

***Abstract*—Release is a ubiquitous concept in software development, referring to grouping multiple independent changes into a deliverable piece of software. Mining releases can help developers understand the software evolution at coarse grain, identify which features were delivered or bugs were fixed, and pinpoint who contributed on a given release. A typical initial step of release mining consists of identifying which commits compose a given release. We could find two main strategies used in the literature to perform this task: time-based and range-based. Some release mining works recognize that those strategies are subject to misclassifications but do not quantify the impact of such a threat. This paper analyzed 13,419 releases and 1,414,997 commits from 100 relevant open source projects hosted at GitHub to assess both strategies in terms of precision and recall. We observed that, in general, the range-based strategy has superior results than the time-based strategy. Nevertheless, even when the range-based strategy is in place, some releases still show misclassifications. Thus, our paper also discusses some situations in which each strategy degrades, potentially leading to bias on the mining results if not adequately known and avoided.**

I. Introduction

Release engineering is a discipline that integrates the source code independently developed by the contributors into a coherent software product, which includes libraries and other resources needed for the software deployment and execution [1]. A release groups multiple independent changes into a deliverable piece of software targeted at specific stakeholders.

Some of the key applications of release mining include automatically composing release notes [2, 3, 4, 5] and comparing releases [6, 7, 8, 9]. Release notes summarize release information and are important to enable end-users, product owners, integrators, and developers to understand the changes that occurred in the software since its previous releases. Such changes may include, for example, new features, bug fixes, architectural changes, and changes to licenses [4]. On the other hand, releases comparison allows contrasting specific characteristics of the release engineering process with the observed outcome. For instance, Khomh et al. [8, 9] compare releases to understand the impact of rapid release on software quality, and Clark et al. [6] compare releases to understand the implications of rapid releases on software security. Furthermore, developers fixing bugs may narrow the searchable code base by identifying the commits of the release that inserted the bug.

A common initial step of the release mining approach consists of identifying which commits belong to each release. Although tags are frequently used to indicate the last commit of a release, version control systems such as Git do not provide built-in support to obtain the remaining commits of a given release. Hence, stakeholders aiming at release mining in Git must adopt a strategy to identify the commits that belong to each release. The most common strategies to assign commits to a release are time-based and range-based. The time-based strategy assumes that any reachable commit in a specific time interval belongs to a release. For instance, the Git command `git rev-list --since 2020-6-1 1.0.4`, which implements the time-based strategy, lists all commits of a given release that occurred in June, 2020. On the other hand, the range-based strategy assigns the commits in the change path between two tags to the release. For instance, the Git command `git rev-list 1.0.3..1.0.4`, which implements the range-based strategy, lists all commits of release 1.0.4.

As recognized by existing release mining works [3, 4], those strategies are subject to potential false positives and false negatives. Depending on the software development history, irrelevant commits may be accounted into a release, and relevant commits may not be accounted. This threat can potentially affect the target release mining application. For instance, it may lead to release notes with information not related to a given release or not including important information. However, to the best of our knowledge, we still have no quantitative evidence in the literature on the impact of such a threat according to the adopted strategy.

In this work, we assessed the precision and recall of time-based and range-based strategies for commit assignment to releases. In our context, precision is the fraction of relevant commits among all commits assigned to a given release, and recall is the fraction of relevant commits assigned to the release among the total amount of relevant commits. Moreover, we investigated whether the number of developers and base releases influence the precision and recall results, depending on the adopted strategy. We also investigated the effect of including all the commits available in a specific time interval to the time-based strategy analysis. To do so, we implemented both strategies and executed them over 13,419 releases and 1,414,997 commits from 100 relevant open source software

projects. We contrasted the results with a baseline to compute the precision and recall metrics. The baseline considers the whole project history and the exact moment each release was created to identify which release delivers each commit for the first time.

Our study is organized into three research questions. In the following, we list them and briefly discuss the main findings:

RQ1. How do time-based and range-based strategies compare in terms of precision and recall? In this research question, we investigate the effectiveness of the time-based and range-based strategies. The answer to this research question enables stakeholders to select the best strategy based on their effectiveness. We found that the time-based and range-based strategies have equivalent precision ($\mu = 98.58\%$ *vs.* $\mu = 98.62\%$, respectively). However, the time-based strategy has a statistically significantly lower recall ($\mu = 91.89\%$) than the range-based strategy ($\mu = 100\%$), with a large effect size.

RQ2. How do the number of developers and the number of base releases influence the precision and recall of the time-based and range-based strategies? In this research question, we investigate factors that may influence the effectiveness of the time-based and range-based strategies. The answer to this research question helps stakeholders choose the most effective strategy depending on the project's characteristics. We found that increasing the number of developers has little influence on the precision of both strategies (with a negligible effect size). It increases the recall of the time-based strategy, comparing releases with many developers ($\mu = 96.82\%$) and releases with few developers ($\mu = 89.18\%$), with a large effect size, but does not influence the recall of range-based strategy. Also, we found that increasing the number of base releases does not influence precision in either strategy. Moreover, it reduces the recall of the time-based strategy, comparing releases with multiple base releases ($\mu = 71.39\%$) and releases with a single base release ($\mu = 95.13\%$), with a large effect size, but does not influence the recall of the range-based strategy.

RQ3. How does the inclusion of all the commits available in a specific time interval influence the precision and recall of the time-based strategy? The time-based strategy may be inadvertently parameterized to consider all commits in the time interval, disregarding being reachable by the release under analysis. The answer to this research question helps stakeholders know the impact of running the time-based strategy with this alternate parameterization. We found that this approach jeopardizes both precision ($\mu = 98.58$ *vs.* $\mu = 64.37$, with a large effect size) and recall (with a negligible effect size) of the time-based strategy.

This paper is organized into six other sections besides this introduction. In Section II, we explain some key version control concepts. In Section III, we detail the materials and methods of our research. In Section IV, we answer each research question and discuss the findings of our paper. In Section V, we discuss the threats of the validity of our results. In Section VI, we present the related work. Finally, in Section VII, we conclude our work and highlight some future work.

II. Background

This section presents some basic knowledge about version control system concepts. More specifically, we focus on Git's concepts, as it is the version control system used by all projects in our corpus.

A. Commits, Tags, and Releases

Git is a distributed version control system that represents the changes to the software as commits. A commit is an object uniquely identified by a SHA1 hash that references the state of the code in a given moment. When a developer checks out a commit, Git retrieves the versions of the software artifacts stored in the repository when the commit was made. Each commit also stores metadata, including the message explaining the change, the author, the committer (the developer who applied the change to the repository), and the timestamp. A commit *c* also references its parents, i.e., the commits that commit *c* was based on. Generally, a commit has only one parent. A commit with more than one parent is a merge commit.

Like most version control systems, Git provides tags, which consists of a mechanism to reference a single object stored in the repository (generally a commit). The tags allow the developers to name, describe, and timestamp the releases. Hence, it is possible to label a commit as a release using tags, e.g., labeling a commit with hash "1d35..." as release "1.0.1".

Finally, release names may have semantics, such as those that use Semantic Versioning [10]. The Semantic Versioning labels the release with three numbers separated by dots (e.g., 1.2.3), respectively, the major, minor, and patch versions. The major version represents releases that are backward incompatible with previous ones; the minor version represents releases that introduce backward compatible new features; and the patch version represents releases that only fix bugs without introducing new features. For instance, according to semantic versioning, the difference between release "1.0.0" and "1.0.1" is just bugfixes.

B. Software and Release Development History

Although a commit is atomic and collects all the software artifacts versions available at the time it was made, sequencing commits enables stakeholders to retrieve the software's incremental evolution. Therefore, the commits and their parents describe the software development history and may be represented as a directed acyclic graph, in which the nodes are the commits and the edges are the parent relationships. Fig. 1 present two examples of a commit graph, which show all changes sequenced in the inverse order they were introduced in the repository.

In essence, the commits of a given release are those introduced in the release development. The developers start the development from a commit that belongs to a previous release, i.e., its base release. Gradually, they include new commits to the release as the development progresses. Eventually,

developers may integrate the code of other releases into the release under development, adding these as base releases of the current release.

C. Branch, Merge, and Rebases

On Git, when developers need to work in parallel, they may use branches to isolate their changes. Eventually, they may need to integrate the parallel work. Such an integration, also known as merge, produces a commit with more than one parent, each one being the last of commit of each branch. Git also offers the rebase command. Roughly speaking, it removes the commits from a given branch and reapplies them at the end of another branch, changing the project development history.

III. Materials and Methods

In this section, we explain in more detail the research questions that guided this study and the approach we used to answers them. Additionally, we present the strategies to assign commits to releases, the corpus we used to evaluate our work, the process to mine releases, the baseline used to compare the strategies, and the dependent variables.

We provide a replication package of our study online[1]. The replication package includes the repositories and the scripts we used to run our experiment. We implemented the strategies to assign commits to releases in a release mining tool named Releasy[2].

A. Research Questions

1) How do time-based and range-based strategies compare in terms of precision and recall? (RQ1): In this research question, we assess time-based and range-based strategies to identify which selects commits with higher precision and recall. The answer for this question could help researchers, developers, and product owners choose the best strategy to support their release mining process. Once defined the strategy, this answer may provide information about the errs that the commit assignment to releases may introduce to their target application. For instance, developers using a low recall strategy to generate release notes must be aware that the resulting release notes may be incomplete. We mined the releases using both strategies and tested the following null hypothesis:

$\mathbf{H_0^1}$: *There is no significant difference between the mean of precision and recall on commit assignment to release using time-based and range-based strategies.*

We adopted $\alpha = 0.05$ for all tests. First, we run the Shapiro test to check whether the data follows a normal distribution. Since our data do not fit a normal distribution (p-value < 0.0001), we run the non-parametric Wilcoxon paired test to check whether there is a significant difference between the mean of the distributions of precision and recall of each strategy. We used the paired test because our distributions originate from the same projects, distinguishing themselves only by the strategy used to assign commits to releases. Finally, we used the Cliff's Delta test to calculate the effect size of the difference, i.e., the magnitude of the difference.

2) How do the number of developers and the number of base releases influence the precision and recall of the time-based and range-based strategies? (RQ2): In this research question, we want to discover factors that may influence the precision and recall of the time-based and range-based strategies. This question's answer may reveal that one strategy is better than the other in specific scenarios, such as those involving higher parallelism.

We choose to investigate metrics related to parallel work. We used two factors: the *number of developers* and the *number of base releases*. A higher *number of developers* involved in a release, very likely, may represent that more parallel work has happened during the development of the release. Also, a higher *number of base releases*, very likely, may represent the development of multiple releases in parallel. We do not expect to define guidelines to compose the project team or the development process, but we intend to provide awareness about each strategy's limitations regarding these factors.

We calculated the Spearman correlation of the *number of developers* with the number of commits in a release and could observe that it is high ($\rho = 0.7410$), according to Hinkle et al. [11]. Hence, we normalized the *number of developers* per the number of commits in the release to reduce the effect of the release size in the analysis. Since the releases have much more commits than developers, we opted to multiply the metric by 100, avoiding small decimal numbers. We also calculated the Spearman correlation of the *number of base releases* with the number of commits in a release and could observe that it is low ($\rho = 0.3396$), according to Hinkle et al. [11]. Hence, we did not normalize the *number of based releases*. Thus, we adopted the following metrics, extracted per release under analysis:

- *Number of developers per 100 commits*: the number of unique developers that made at least one commit to the release, divided by the total commits of the release, times 100.
- *Number of base releases*: the number of base releases of the release.

We calculate both metrics for all releases. Then we calculated the mean of the *number of developers per 100 commits* ($\mu = 25.27$) and the *number of base releases* ($\mu = 1.33$) considering all releases. Next, we used the mean of the *number of developers per 100 commits* to divide the releases into two groups (two treatments): the group with *fewer developers* (fewer than the mean) and the group with *many developers* (bigger than or equal to the mean). Likewise, we used the mean of the *number of base releases* to divide the releases into two groups (two treatments): the group with *single base releases* (only one) and the group with *multiple base releases* (more than one).

We mined the releases of each group using both strategies. Then, we assess the strategies comparing the groups fewer *vs.* many developers, and single *vs.* multiple base releases. We tested the following null hypothesis:

[1]https://github.com/gems-uff/release-mining

[2]https://github.com/gems-uff/releasy

$\mathbf{H_0^{2a}}$: *There is no significant difference between the mean of precision and recall on commit assignment to releases with few and many developers.*

$\mathbf{H_0^{2b}}$: *There is no significant difference between the mean of precision and recall on commit assignment to releases with single and multiple base releases.*

We followed the same statistical approach of RQ1, but instead of comparing the strategies, we compared the groups, testing the treatments of each factor independently for each strategy. The Shapiro test shows that all the distributions do not fit a normal distribution (p-value < 0.0001). We did not test the recall of the range-based strategy because it achieved 100% recall for all treatments.

3) How does the inclusion of all the commits available in a specific time interval influence the precision and recall of the time-based strategy? (RQ3): In Section III-B1, we explained the time-based strategy. We introduced that a developer may run the strategy considering all the commits available in a specific time interval, independently of the commit being reachable by the release. In this question, we want to assess the impact of this variant on the precision and recall of the time-based strategy.

We mined the releases using the time-based strategy considering the reachable commits and considering all the commits. We tested the following null hypothesis:

$\mathbf{H_0^3}$: *There is no significant difference between the mean of precision and recall on commit assignment to release using time-based strategy considering only reachable commits or all commits available in a specific time interval.*

We followed the same statistical approach of RQ1. The Shapiro test shows that all the distributions do not fit a normal distribution (p-value < 0.0001).

B. Release Mining Strategies

We found works in the literature that assign commits to releases considering the commit timestamps [3, 9, 7, 4, 12], called time-based strategies, and considering the range of commits [13, 14, 15], called range-based strategies.

The strategies prune the project's commit graph to retrieve the sub-graph of commits that belong to a given release. Both strategies use the release's tag to assign the first commit that composes the sub-graph and a reference to recognize which commits must belong to the sub-graph. The reference accomplishes the stop condition of their algorithms and is generally related to a base release. The time-based strategy uses the base release's timestamp as a reference, and the range-based strategy uses the base release tag as reference. Since a release may have more than one base release, the commits assigned may vary according to the choice. Also, choosing a wrong base release may lead to incorrect results.

We explain the time-based and range-based strategies in the following sections.

1) Time-based Strategy: The time-based strategy assigns the commits to releases based on the commits' timestamp, i.e., when the commits were inserted into the version control system. It starts assigning the commit referenced by the tag of the current release. Then, it walks through the repository history, assigning all reachable commits made after the reference timestamp. The Git command `git rev-list --since 2020-6-1 1.0.4` implements this strategy. It retrieves all commits reachable by release 1.0.4 made after June 1st, 2020.

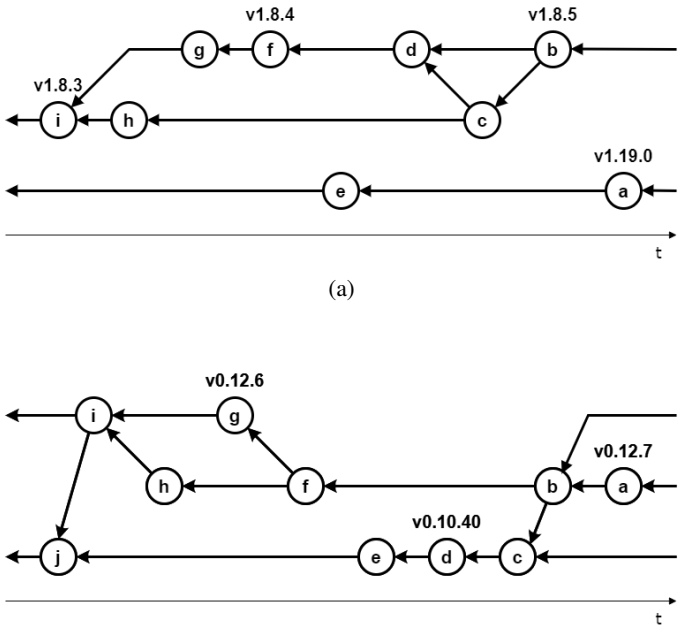


Figure 1: Part of the commit graph of D3.js (a) and Node.js (b) projects, excluding some consecutive commits. The circles represent commits, positioned from left to right according to their commit timestamp, i.e., the commit from the left is older than the commit from the right. The arrows point from a commit to its parents.

For instance, Moreno et al. [4] use this approach to generate release notes. The authors state that the dates between the release and its base release are approximations because developers may be working on other releases or even start developing the next release before finishing the current release.

In Fig. 1a, when analyzing the release "v1.8.5" and using the timestamp of the release "v1.8.4" as reference, the time-based strategy would assign the commits $\{b, c, d\}$ to the release "v1.8.5". The strategy would not assign the commits $\{f, g, h, i\}$ because they were made before the release "v1.8.4" timestamp. Moreover, the strategy would not assign the commits $\{a, e\}$ because they are not reachable by the tag "v1.8.5". In Fig. 1b, when analyzing the release "v0.12.7" and using the timestamp of the release "v0.12.6" as reference, the time-based strategy would assign the commits $\{a, b, c, d, e, f\}$ to the release "v0.12.7". The strategy would not assign the commits $\{g, h, i, j\}$ because they were made before the release "v0.12.6" timestamp.

A developer may inadvertently run the time-based strategy considering all the commits (including those unreachable). The inclusion of all commits would assign commits from branches unrelated to the current release development. In Fig. 1a, considering all the commits, the time-based strategy would inappropriately include the commit $\{e\}$ to release "v1.8.5". The strategy would not include the commit $\{a\}$ because it was made after the release "v1.8.5". The Git command `git rev-list --since 2020-6-1 2020-6-30 --all` implements this variant of the time-based strategy by assigning all the commits made in June, 2020 to the release.

Finally, in Git, the commit timestamp is obtained from the committers' computer, and there is no validation on the repository regarding the correctness of this information. Also, developers can create a tag with a timestamp different than the commit's timestamp. Hence, commits and tags accidentally reported with the wrong timestamp may influence the time-based strategy results.

2) Range-based strategy: The range-based strategy assigns the commits to releases based on the repository history instead of the time. The strategy selects the commits reachable by a given release that are not reachable by its base release. This strategy identifies all reachable commits of both the release under analysis and its base release by walking through the transitive closure of the tagged commits. Then, the strategy subtracts the set of the base release's commits from the set of commits of the release under analysis. The remaining commits are the ones that the strategy assigns to the release under analysis. For instance, the Git command `git rev-list 1.0.3..1.0.4` implements this strategy. It retrieves all commits that are reachable by release "1.0.4" but are not reachable by release "1.0.3".

For instance, GitHub [15] uses this approach to compare releases. Also, Chacon and Straub [14] explains the use of double dot notation to determine the commits reachable by one release that are not reachable by another.

In Fig. 1a, when analyzing the release "v1.8.5" and using the release "v1.8.4" as reference, the range-based strategy would assign the commits $\{b, c, d, h\}$ to the release "v1.8.5". The strategy would not assign the commits $\{f, g, i\}$ because they are reachable by the release "v1.8.4". Moreover, the strategy would not assign the commits $\{a, e\}$ because they are unreachable by the release under analysis. In Fig. 1b, when analyzing the release "v0.12.7" and using the release "v0.12.6" as reference, the range-based strategy would assign the commits $\{a, b, c, d, e, f, h\}$ to the release "v0.12.7". The strategy would not assign the commits $\{g, i, j\}$ because they are reachable by the release "v0.12.6".

C. Project Corpus

When selecting the corpus for our study, we aimed at mature and relevant open source projects. We chose to search for projects hosted on GitHub because it hosts millions of projects and provides ease of search through APIs. In GitHub, users can assign stars to track projects they like or find interesting. Hence, we consider the number of stars of a project as a good approximation of GitHub's project relevance (more is better) [16]. Thus, we sorted the search results descending by the number of stars of the projects and selected those with the higher number of stars.

We selected projects from multiple programming languages, considering the top 10 popular programming languages from the 2019 Stack Overflow Survey[3]. In 2019, nearly 90,000 developers answered this survey, which revealed the most popular technologies. The survey ranks programming, script, and markup languages. We choose to discard HTML and CSS because they are not programming languages. We also discarded SQL, Shell, and PowerShell because they are languages targeted at specific purposes, such as interacting with a database and running operating system commands. Our selection consists of the following programming languages: JavaScript, Python, Java, C#, PHP, C++, TypeScript, C, Ruby, and Go.

On September 17th, 2020, we searched GitHub to populate our corpus. We run a GraphQL query for each programming language to retrieve the 35 projects with the most stars. Then, we applied seven filters to avoid perils [17, 18] and produce a balanced corpus, preventing including projects too small that would be irrelevant to the analysis or too big that would dominate the analysis. We describe the filters in the following:

1) Inactive projects: we removed projects without commits made in the last 180 days to avoid working with abandoned projects;
2) Small projects: we removed projects with less than 2,000 commits to avoid immature or toy projects;
3) Non-software projects: we removed non-software projects, such as documentation projects;
4) Big projects: we removed projects that alone represent more than 5% of our corpus's total number of commits to preventing the characteristics of such projects from taking precedence over the characteristics of other projects;
5) "Monorepo" projects: we removed projects that host releases of different products on the same Git repository;
6) Few releases: we removed projects with fewer than ten final releases to avoid working with projects that are still in their initial releasing phase;
7) Non-semantic versioning projects: we removed projects that use non-semantic versioning releases, such as projects that use dates to label their releases.

We iterative applied the filters. First, we selected 100 projects, the 10 with the most stars for each programming language, and applied the filters to them. After executing a given filter, we added the next projects from our query results to top up ten projects per language again. Then, we reapplied the same filter. We repeated this process until the filter stop removing projects. Then, we applied the next filter using the same process (filtering and topping up). After applying all the filters, we repeated the process from the beginning until all filters stop removing projects, which happened after the third

[3]https://insights.stackoverflow.com/survey/2019/#technology

Table I: The project filtering results per round

Filter	Round 1	Round 2	Round 3	Total
Inactive	2	8	2	12
Small	34	24	4	62
Non-software	9	3	-	12
Big	7	3	-	10
"Monorepo"	33	6	-	39
Few releases	4	2	-	6
Non-semantic versioning:	1	0	-	1

Table II: The project characteristics per programming language ($n = 10$ per programming language)

Language	# Stars		# Commits		# Releases	
	Min.	Max.	Min.	Max.	Min.	Max.
C	12,189	48,821	2,055	48,773	24	530
C#	7,388	18,451	2,201	18,402	20	195
C++	21,113	86,283	3,998	30,739	18	506
Go	23,744	47,145	3,659	27,005	30	338
Java	22,463	51,561	2,066	55,176	27	249
JavaScript	54,145	173,633	3,127	35,292	49	593
PHP	14,385	61,920	2,391	42,497	12	584
Python	30,376	52,756	2,215	50,771	32	393
Ruby	11,024	31,669	2,280	39,546	68	184
TypeScript	27,952	64,910	2,030	31,206	49	352

round. Table I shows the order we applied the filters and the removals per round. In total, the filters removed 142 projects.

Our final corpus comprises 13,419 releases, excluding pre-releases, such as betas and release candidates, and 1,414,997 commits from 100 relevant open source projects developed using ten different programming languages. Table II show the corpus' characteristics per programming language, including the number of stars, commits, and releases.

D. Data processing

We cloned all the Git repositories of our corpus on October 11th, 2020, and implemented the whole mining process, including the range-based and time-based strategies.

First, we identified the releases on the Git repository applying a regular expression that finds the version number in the tags' name. The regular expression comprises three parts and separates the tag name into prefix `(?<prefix>(?:[^\s]*?)`, version number `(?<version>(?:[0-9]+[\._])*[0-9]+)`, and suffix `(?<suffix>[^\s]*)`. We included the positive look ahead `(?=(?:[0-9]+[\._]))` to handle releases with number in the prefix and the alternative `|(?:[^\s]*?)` to handle releases with a single version number (e.g., "r1"). For instance, for the release tag "v1.0.0beta" our regular expression would assign "v" as prefix, "1.0.0" as version number, and "beta" as suffix.

Next, we identified the tags that correspond to pre-releases, e.g., "v1.0.0beta". For most projects, we considered pre-releases the tags with a suffix. The exception were the projects "spring-projects/spring-boot", "spring-projects/spring-framework", "netty/netty", and "godotengine/-godot", which use suffix for all releases. For these projects, we manually analyzed the suffix to identify the pre-releases. We discarded all pre-release tags because the commits of the pre-releases also belong to the final release, e.g., the commits of the release "v1.0.0beta" belong to the final release "v1.0.0".

In addition, we analyzed the remained tags to check whether they contain any conflicts, i.e., different tags representing the same semantic versioning number or referencing the same commit. We found 98 release conflicts. We discarded one of the two tags to resolve the issue. We found 55 pairs of tags with the same semantic versioning number: some tags representing the same version but with a missing bugfix number (e.g., v1.0 and v1.0.0) and other tags with different prefixes (e.g., v1.0.0 and 1.0.0). Surprisingly, despite the same semantic, these tags do not always reference the same commit. We applied the following heuristics to address this conflict: first, we removed the tags using the least common prefix adopted by the project; then, we removed the tags not using semantic versioning pattern. We also found 43 tags with different version referencing to the same commit. We removed the tag with the higher semantic versioning number to address this issue.

Finally, we mined the repositories with the time-based and range-based strategies. For both strategies, we choose the base release as the previous semantic version available when the release in analysis was created. In Fig. 1a we would use the release "v1.8.4" as the base release of the release "v1.8.5", and in Fig. 1b we would use the release "v0.12.6" as the base release of the release "0.12.7".

E. Baseline

We need to establish the baseline containing the set of commits that actually belongs to the release, to check whether the commits assigned by a strategy are correct or not. Since there is no predefined list of the commits belonging to each release in our corpus, we devised an algorithm to define the baseline.

Our algorithm processes the whole Git repository and find the release that first delivered a given commit. The algorithm first orders all the releases available in the repository by time. Then, for each release in ascending time order, the algorithm assigns all the commits that are reachable by the release but were not previously assigned to other releases. In Fig. 1a, we would assign to release "v1.8.5" the commits $\{b, c, d, h\}$. We would not assign the commits $\{f, g\}$ and $\{i\}$ because they would have been previously assigned to releases "v1.8.4" and "v1.8.3", respectively. Moreover, we would not assign the commits $\{a, e\}$ because they are not reachable by the release "v1.8.5". In Fig. 1b, we would assign the commits $\{a, b, c, f, h\}$. We would not include the commits $\{g, i, j\}$ and $\{d, e\}$ because they would have been previously assigned to releases "v0.12.6" and "v0.10.40", respectively.

Unfortunately, no Git command implements this algorithm, and the algorithm demands analyzing the whole repository history, which is impractical to be conducted manually. This possibly explains why the existing work in the literature opted to use time-based or range-based strategies instead of this algorithm for assigning commits to releases.

F. Dependent variables: precision and recall

We assessed the effectiveness of the strategies using precision and recall (dependent variables). The precision and recall of commits assigned to a release were calculated with their traditional formula: $precision = TP/(TP + FP)$ and $recall = TP/(TP + FN)$. We compare the set of commits C_i^s assigned by a given strategy s to the release r_i with the set of commits C_i of the baseline for the same release r_i. Then, we identify the false positives $FP = C_i^s \setminus C_i$, false negatives $FN = C_i \setminus C_i^s$, and true positives $TP = C_i^s \cap C_i$. When a strategy completely errs the analysis, generating zero true positives and false positives, and thus a potential division by zero, we set the precision to zero. We also calculated the F-measure for the releases, which is the harmonic mean of the precision and recall.

Even after applying the filter to remove big projects, the corpus still contains some projects with few releases ($min = 12$) and others with many releases ($max = 593$). The difference in the number of releases may happen because the development process adopted by the project may influence the number of releases [19]. For instance, the project "d3/d3" comprises 4,282 commits and 265 releases. In comparison, the project "tesseract-ocr/tesseract" comprises 4,600 commits but only 18 releases. Thus, we calculated the dependent variables per release and used the macro averaged mean [20] to aggregate the metrics per project, i.e., the project's precision is the mean of the precision of its releases, and the project's recall is the mean of the recall of its releases. Finally, we calculated the overall precision as the macro averaged mean of all projects' precision and the overall recall as the macro averaged mean of all projects' recall.

IV. Results and Discussion

In this section, we answer the research questions, present the results, and discuss our findings.

A. How do time-based and range-based strategies compare in terms of precision and recall? (RQ1)

The Wilcoxon test returned p-value $= 0.0851$ for precision and p-value < 0.0001 for recall. Thus, we could not reject H_0^1 for precision but could reject for recall. The Cliff's Delta test returned $d = 1.0000$, which represents a large effect size. Thus, we could conclude that the strategies have similar precision but significantly different recall. Table III shows the macro averaged mean precision, recall, and F-measure for each strategy. The range-based strategy has higher precision and recall than the time-based strategy, suggesting that the range-based strategy is the most appropriate strategy to assign commits to releases.

Fig. 2 shows the distributions of precision and recall of the projects. We could observe more projects with higher precision with the time-based strategy (median $= 100.00\%$) than with the range-based strategy (median $= 99.88\%$). However, the test revealed that the difference is not statistically significant. We could also observe that all the projects achieved 100.00% recall using the range-based strategy, which means that the strategy did not miss any commit in our experiment.

Table III: The overall precision, recall, and F-measure of the *corpus* according to each strategy.

Strategy	Precision	Recall	F-measure
Time-based	98.58%	91.89%	92.94%
Range-based	98.62%	100.00%	98.93%

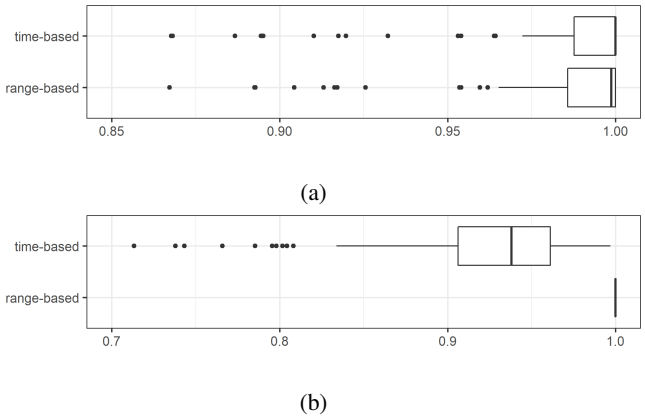


Figure 2: The comparison of the macro averaged mean of precision (a) and recall (b) for the time-based and range-based strategies.

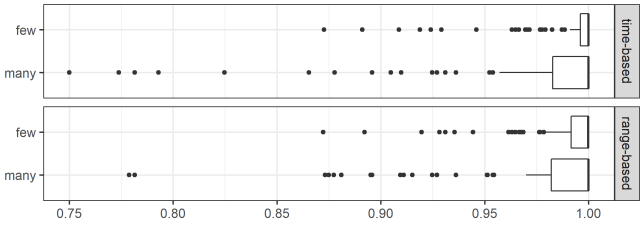
RQ1. How do time-based and range-based strategies compare in terms of precision and recall?
Answer: The time-based and range-based strategies have equivalent precision ($\mu = 98.58\%$ *vs.* $\mu = 98.62\%$, respectively), but the time-based strategy has lower recall ($\mu = 91.89\%$) than the range-based strategy ($\mu = 100\%$).
Implications: In general, stakeholders in charge of mining release should consider the use of range-based strategy instead of time-based. The range-based strategy include a similar small amount of false positives when compared to the time-based strategy, but without false negatives. In practice, in a release notes generation, for instance, all features and bug fixes actually implemented by the release would be listed, but unfortunately, some few features or bug fix that belong to other releases would be inappropriately listed too.

B. How do the number of developers and the number of base releases influence the precision and recall of the time-based and range-based strategies? (RQ2)

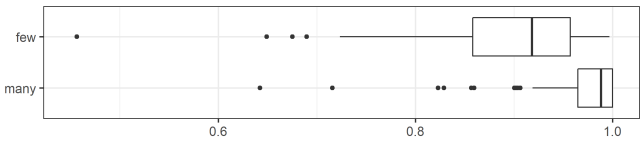
In the analysis of the factor *number of developers* and the treatments *few developers* and *many developers*, the Wilcoxon test returned p-value < 0.0001 for the precision and recall of the time-based strategy, and p-value $= 0.0023$ for the precision of the range-based strategy. We did not test recall for the range-based strategy because both have $\mu = 100\%$. Thus,

Table IV: The overall precision, recall, and F-measure of the factors *number of developers* and *base releases*.

Factor	Strategy	Treatment	Precision	Recall	F-measure
Developers	Time-based	few	99.01%	89.19%	90.96%
		many	97.59%	96.82%	96.31%
	Range-based	few	98.92%	100.00%	99.20%
		many	97.82%	100.00%	98.19%
Base releases	Time-based	single	98.64%	95.13%	95.41%
		multiple	96.92%	71.39%	76.12%
	Range-based	single	98.83%	100.00%	99.02%
		multiple	97.45%	100.00%	98.32%



(a)



(b)

Figure 3: The comparison of the macro averaged mean of precision (a) and recall (b) for the time-based and range-based strategies considering the factor *number of developers*.

we could reject H_0^{2a} for the precision and recall of the time-based strategy, and for the precision of range-based strategy. The Cliff's Delta test returned $d = 0.0649$ (negligible effect size) for the precision of the time-based strategy, $d = 0.6979$ (large effect size) for the recall of the time-based strategy, and $d = 0.0333$ (negligible effect size) for the precision of range-based strategy. We could conclude that the factor *number of developers* has negligible influence in the precision of both strategies but a large influence in the recall of time-based strategy. Table IV show the precision, recall, and F-measure of each strategy considering each treatment. We could observe that the F-measure of the time-based strategy is higher with the treatment of *many developers*. Fig 3 shows the distributions of precision and recall of the projects according to the number of developers working in the releases. Although it is negligible, we could observe that the precision of both strategies is a little lower with the treatment *many developers*. Moreover, we could observe that the recall of the time-based strategy is higher with the treatment *many developers*.

In the analysis of the factor *number of base releases* and the treatments *single base release* and *multiple base releases*,

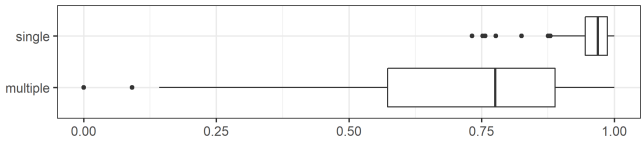


Figure 4: The comparison of the macro averaged mean of recall for time-based strategy considering the factor *number of base releases*.

the Wilcoxon test returned $p\text{-value} = 0.6529$ for precision of the time-based strategy, $p\text{-value} < 0.0001$ for the recall of the time-based strategy, and $p\text{-value} = 0.0898$ for the precision of the range-based strategy. Again, we did not test the recall of the range-based strategy because both have $\mu = 100\%$. We could reject H_0^{2b} just for the recall of the time-based strategy. The Cliff's delta test returned $d = 0.7667$ (large effect size) for the recall of the time-based strategy. We could conclude that the factor *number of base releases* has no influence on the precision of both strategies but has a large influence in the recall of the time-based strategy. Also, the factor *number of base releases* does not influence the range-based strategy. In Table IV, we could observe that the F-measure of the time-based strategy is lower with the treatment of *multiple base releases* than with the treatment *single base releases*. Fig. 4 shows the distribution of recall of the projects according to the number of base releases. We could observe that the recall of the time-based strategy with the treatment *multiple base releases* is lower than the recall of the time-based strategy with the treatment *single base release*.

RQ2. How do the number of developers and the number of base releases influence the precision and recall of the time-based and range-based strategies?
Answer: We found that releases with many developers have little influence on the precision of both strategies (with negligible effect size) but raises the recall of the time-based strategy ($\mu = 96.82\%$) when compared to releases with few developers ($\mu = 89.19\%$), with large effect size. On the other hand, for the time-based strategy, releases with multiple base releases have a lower recall ($\mu = 71.39\%$) than releases with a single base release ($\mu = 95.13\%$), with large effect size.
Implications: Stakeholders in charge of mining releases using the time-based strategy may achieve higher recall when analyzing releases with *many developers*. However, they should avoid using the time-based strategy on releases with *multiple base releases*.

C. How does the inclusion of all the commits available in a specific time interval influence the precision and recall of the time-based strategy? (RQ3)

The Wilcoxon test returned $p\text{-value} < 0.0001$ for precision. Although only 13% of the projects presented differences in

Table V: The overall precision, recall, and F-measure of the *corpus* according to time-based strategy, including just reachable commits and including all commits available in a specific time interval.

Strategy	Precision	Recall	F-measure
Time-based (reachable commits)	98.58%	91.89%	92.94%
Time-based (all commits)	64.37%	91.84%	66.71%

the recall, the Wilcoxon test returned $p\text{-value} = 0.0017$ for recall. Thus, we could reject H_0^3. The Cliff's Delta test return $d = 0.9512$ (large effect size) for precision and $d = 0.0085$ (negligible effect size) for recall. Table V shows the macro averaged mean of precision, recall, and F-measure for the strategy. The time-based strategy, including all commits, has lower F-measure than the time-based strategy, including only reachable commits, suggesting that the inclusion of all available commits in a specific time interval jeopardizes the precision of the time-based strategy.

Fig. 5a shows the distribution of the projects' precision using the time-based strategy, including just the reachable commits and including all commits available in a specific time interval. We could observe that the precision is lower for the majority of projects using the time-based strategy, including all commits. The reduction in precision happens because the variant strategy considers commits unrelated to the release under analysis, such as the commit $\{e\}$ shown in Fig. 1a.

Fig. 5b shows the distribution of the recall of the projects, and we could observe that the distributions are similar. We were intrigued by the fact that some releases have different recall using time-based strategy, including just the reachable commits and including all commits available in a specific time interval. Therefore, we sampled some of these releases and we could observe issues with the timestamp of the releases, entailing in the incorrect removal of commits. For instance, the timestamp of release "v0.35.6" of the project "electron/-electron" was made on 12/25/2015, but the referenced commit was made on 01/11/2016, surprisingly after the release. This issue does not happen in the time-based strategy that just consider reachable commits because the algorithm starts at specific commit instead of a timestamp of a tag.

RQ3. How does the inclusion of all the commits available in a specific time interval influence the precision and recall of the time-based strategy?
Answer: We could observe that the inclusion of all available commits in time interval introduce error on the analysis and reduce the precision ($\mu = 98.58$ *vs.* $\mu = 64.37$, with a large effect size) and recall (with a negligible effect size) of the time-based strategy.
Implications: Stakeholders using the time-based strategy should be careful not to include unreachable commits in the analysis.

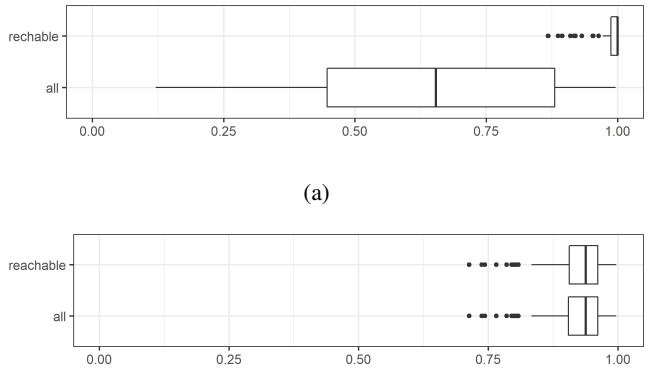


Figure 5: The comparison of the macro averaged mean of precision (a) and recall (b) for the time-based strategy, including just reachable commits and including all commits available in a specific time interval.

V. Threats to Validity

Although we have taken care to avoid bias, some situations may have affected the results. In this section, we discuss these situations, observing the guidelines from Wohlin et al. [21].

Internal validity. We have no control over the development of the projects in our *corpus*. The projects may introduce techniques that change the commit graph and impair our results. For instance, developers may have applied the rebase operation, which creates a more linear commit graph. Moreover, they may have committed unreachable code, such as code enclosed by a feature toggle intended to be delivered in a future release.

Construct validity. We used the previous semantic version number available when the release was created as its base release for both the time-based and range-based strategies. We choose this reference because it is reasonable and straightforward for a developer to adopt. However, other references could have been used, such as the previous release in time or the result of the command `git describe`. The use of other references may change the results of precision and recall for both the strategies.

Finally, we found 98 semantic conflicts in our *corpus*, i.e., two tags representing the same semantic versioning number (e.g., "v1.0.0" and "1.0") or referencing the same commit. We discard one tag for each conflict, which can introduce errors in the analysis. However, the event is rare, representing just about 0.70% of our corpus' releases, thus not imposing a relevant impact to our analysis.

Conclusion Validity. Although we individually calculated precision and recall for the releases, we run the hypotheses test using per project precision and recall. We used the macro averaged mean to aggregate the metrics per project, which give equal weight to all project's release [20]. Hence, releases with few commits will have the same weight in the analysis as releases with many commits.

External validity. We used relevant open source projects to compose our *corpus*. These projects may have unique characteristics not present in an industry project, such as a team composed of hundreds of voluntary collaborators. Furthermore, we applied filters to select the projects, limiting the programming language, size, and release pattern. Hence, the result found in this work cannot be generalized for projects with characteristics different from our *corpus*.

VI. Related Work

In this section, we present studies that are related to our work. The studies are grouped according to their assignment strategy, that is, (i) commit assignment to issues, (ii) issue assignment to releases, (iii) commit assignment to dependent commits, (iv) commit assignment to releases. Considering, specifically, the objective of our study, we did not find research efforts focused on the impact of assign commits to releases, and, to the best of our knowledge, our work is the first attempt investigating this subject.

Commit assignment to issues. The following studies aim at bridging the version control system and issue tracking system. Le et al. [22] evaluates RCLinker, a tool that links commits to release using contextual information and summarizing techniques. They evaluated six projects and achieved overall 50,91% of precision and 89,27% recall. Furthermore, the authors compared their tool with MLink [23], which achieved overall 56.40% precision and 17.96% of recall.

Sun et al. [24] evaluates FRLink, an approach that focuses on recovering the missing issue-fix relationships. They compared their technique with the RCLinker and reported that FRLink could outperform RCLinker in F-measure by 40.75%. They also discuss that, in their context, recall is more relevant than precision.

Issue assignment to releases. Moreno et al. [3, 4] mine Git using the time-based strategy to generate release notes. They select the commits that belong to a release to link them with the issue tracker system. Then, they generate release notes. They evaluate the importance and completeness of their approach regarding identify the features that belong to the release notes but do not evaluate the error introduced by incorrectly choosing the commits that belong to a given release. Abebe et al. [2] studied nine factors that influence the likelihood of an issue being listed in a release note. They first identify the issues present in the release notes, then they search the commits with a message with an issues' identification. They reported they could link 89% of the issues to commits.

Commit assignment to related commits. Dhaliwal et al. [25] propose two different approaches to identify dependencies among commits, aiming at creating groups of dependent commits. In the context of software product lines, features are added to the common components of a software product family and, after, integrated into products following a selective code integration product. By identifying groups of dependent commits, the authors help developers link the commits to the features to enable the selective integration of the features. The approaches achieve precision up to 95% and recall up to 82%.

The identification of related commits from software repositories is also the objective of Hammad [26]. This study presents an approach to identify related and similar source code modifications automatically. The textual contents of commits are used to recover the related commits. However, they do not discuss the threats to validity, considering the precision and recall when assigning the commits to the related ones.

Commit assignment to releases Shobe et al. [13] explain how to mine releases on the Subversion version control system using an approach similar to the range-based strategy. However, they do not discuss the precision and recall of their method.

We found studies comparing traditional and rapid releases of the Firefox browser, which use the Mercurial version control system. Mäntylä et al. [12, 7] compare the releases regarding software testing. They use the time-based strategy but only assign the commits to Firefox's major releases and explain that it is hard to link commits to specific releases. Khomh et al. [8, 9] compare the releases regarding quality. In the first work [8], they check out the versions and compare them externally, like comparing only the last commits of the releases. In the second work [9], they use the time-based strategy to extract information about the developers who commit the changes, the number of commits, and their size. Clark et al. [6] compare the releases regarding the security of the produced code. They check out the code and use only the last commit. Souza et al. [27, 28] analyze patch backouts (i.e., reversing commits) on rapid releases using the time-based strategy.

VII. Conclusion

In this paper, we assessed the time-based and range-based strategies to assign commits to releases. To do so, we created a *corpus* with 100 relevant open source projects, comprising 13,419 releases and 1,414,997 commits. We implemented both strategies in an open source tool and ran the strategies on all the releases to compute each strategy's precision and recall.

We could observe that the range-based strategy has similar precision but higher recall than the time-based strategy. Thus, stakeholders in charge of mining releases should consider adopting the range-based strategy instead of the time-based. If, for some reason only the time-based strategy is available, stakeholders must know that its recall improves for releases with many developers. However, its recall reduces for releases with multiple base releases. Moreover, stakeholders using the time-based strategy must be careful not to include unreachable commits in the analysis because it drastically reduces the strategy's precision.

As future work, we intend to study the effects of changing the base release and investigate the releases that achieved low F-measure with either strategy to understand what caused the issue. Hence, we may propose enhancements to the strategies to achieve better results. We also intend to perform an in-depth study on the effect of the different strategies in release applications, such as release notes generation.

References

[1] B. Adams, S. Bellomo, C. Bird, B. Debic, F. Khomh, K. Moir, and J. ODuinn, "Release Engineering 3.0," *IEEE Software*, vol. 35, no. 02, pp. 22–25, Mar. 2018.

[2] S. L. Abebe, N. Ali, and A. E. Hassan, "An empirical study of software release notes," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1107–1142, 2015.

[3] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, A. Marcus, and G. Canfora, "Automatic Generation of Release Notes," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. ACM, 2014, pp. 484–495.

[4] L. Moreno, G. Bavota, M. Penta, R. Oliveto, A. Marcus, and G. Canfora, "ARENA: An Approach for the Automated Generation of Release Notes," *IEEE Transactions on Software Engineering*, vol. 43, no. 02, pp. 106–127, Feb. 2017.

[5] F. Curty, T. Kohwalter, V. Braganholo, and L. Murta, "An Infrastructure for Software Release Analysis through Provenance Graphs," in *VI Workshop on Software Visualization, Evolution, and Maintenance*, São Carlos, SP, Brazil, 2018.

[6] S. Clark, M. Collis, M. Blaze, and J. M. Smith, "Moving Targets: Security and Rapid-Release in Firefox," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 1256–1266.

[7] M. V. Mäntylä, B. Adams, F. Khomh, E. Engström, and K. Petersen, "On rapid releases and software testing: A case study and a semi-systematic literature review," *Empirical Software Engineering*, vol. 20, no. 5, pp. 1384–1425, Oct. 2015.

[8] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams, "Do faster releases improve software quality? An empirical case study of Mozilla Firefox," in *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, Jun. 2012, pp. 179–188.

[9] F. Khomh, B. Adams, T. Dhaliwal, and Y. Zou, "Understanding the impact of rapid releases on software quality," *Empirical Software Engineering*, vol. 20, no. 2, pp. 336–373, Apr. 2015.

[10] T. Preston-Werner, "Semantic Versioning 2.0.0," https://semver.org/, 2013.

[11] D. E. Hinkle, W. Wiersma, and S. G. Jurs, *Applied Statistics for the Behavioral Sciences*. Houghton Mifflin College Division, 2003, vol. 663.

[12] M. V. Mäntylä, F. Khomh, B. Adams, E. Engström, and K. Petersen, "On Rapid Releases and Software Testing," in *2013 IEEE International Conference on Software Maintenance*, Sep. 2013, pp. 20–29.

[13] J. F. Shobe, M. Y. Karim, M. B. Zanjani, and H. Kagdi, "On mapping releases to commits in open source systems," in *Proceedings of the 22nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: Association for Computing Machinery, Jun. 2014, pp. 68–71.

[14] S. Chacon and B. Straub, *Pro Git*. Springer Nature, 2014.

[15] GitHub, "Comparing releases - GitHub Docs," https://docs.github.com/en/free-pro-team@latest/github/administering-a-repository/comparing-releases, 2020.

[16] H. Borges and M. Tulio Valente, "What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform," *Journal of Systems and Software*, vol. 146, pp. 112–129, Dec. 2018.

[17] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git," in *2009 6th IEEE International Working Conference on Mining Software Repositories*, May 2009, pp. 1–10.

[18] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The Promises and Perils of Mining GitHub," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. ACM, 2014, pp. 92–101.

[19] K. Beck, *Extreme Programming Explained: Embrace Change*. addison-wesley professional, 2000.

[20] C. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[21] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer Science & Business Media, Jun. 2012.

[22] T. B. Le, M. Linares-Vasquez, D. Lo, and D. Poshyvanyk, "RCLinker: Automated Linking of Issue Reports and Commits Leveraging Rich Contextual Information," in *2015 IEEE 23rd International Conference on Program Comprehension*, May 2015, pp. 36–47.

[23] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Multi-layered approach for recovering links between bug reports and fixes," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: Association for Computing Machinery, Nov. 2012, pp. 1–11.

[24] Y. Sun, Q. Wang, and Y. Yang, "Frlink: Improving the recovery of missing issue-commit links by revisiting file relevance," *Information and Software Technology*, vol. 84, pp. 33–47, 2017.

[25] T. Dhaliwal, F. Khomh, Y. Zou, and A. E. Hassan, "Recovering commit dependencies for selective code integration in software product lines," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, Sep. 2012, pp. 202–211.

[26] M. Hammad, "Identifying related commits from software repositories," *International Journal of Computer Applications in Technology*, vol. 51, no. 3, pp. 212–218, 2015.

[27] R. Souza, C. Chavez, and R. A. Bittencourt, "Do Rapid Releases Affect Bug Reopening? A Case Study of Fire-

fox," in *2014 Brazilian Symposium on Software Engineering*, Sep. 2014, pp. 31–40.

[28] ——, "Rapid Releases and Patch Backouts: A Software Analytics Approach," *IEEE Software*, vol. 32, no. 2, pp. 89–96, Mar. 2015.