Noname manuscript No.
(will be inserted by the editor)

On the assignment of commits to releases

Felipe Curty do Rego Pinto · Leonardo Gresta Paulino Murta

the date of receipt and acceptance should be inserted later

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.

Keywords release · release mining · commit assignment · release analysis

1 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

Felipe Curty do Rego Pinto
Instituto de Computação - Universidade Federal Fluminense
Niterói - RJ - Brazil
E-mail: felipecrp@id.uff.br

Leonardo Gresta Paulino Murta
Instituto de Computação - Universidade Federal Fluminense
Niterói - RJ - Brazil
E-mail: leomurta@ic.uff.br

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, release 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 the adoption of rapid releases on software quality, and Clark et al. [6] compare releases to understand the implications of the adoption 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 since June 1st, 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. Also, the lack of standards and tools makes retrieving the release history difficult and inaccurate[10]. 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 unrelated to a given release or missing important information. Hence, the release mining application will report inaccurate information in such cases, jeopardizing the analysis results.

We searched in Stack Overflow and found that developers raised doubts about how to assign commits to releases [11, 12, 13, 14]. We observed that the provided answers use Git commands to inspect the project history, but did not address the issues reported in this paper. Moreover, we conducted a systematic literature review to find works that analyzed rapid and traditional software releases [15]. We conducted our systematic literature review using the snowballing approach and the guidelines proposed by Wohlin [16]. We found 22 relevant papers (out of 492 analyzed papers), but none discussed the impact of wrong commit assignment in their results. We also ad hoc searched for other release mining work on Google Scholar and did not find any work or discussion regarding the accuracy of the commit assignment to releases. Thus, this work is the first attempt to provide quantitative evidence on the impact of such a threat according to the adopted strategy to assign commits to releases.

In this work, we assessed the precision and recall of time-based and range-based strategies for the 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 unique developers and base releases influence the precision and recall results, depending on the adopted strategy. Regarding the time-based strategy, we also investigated the effect of including all the commits available in a specific time interval and using a more accurate reference. 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. We devised an algorithm to define the baseline, which 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 four 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 unique developers, the number of base releases, and the cycle time 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 unique 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. The cycle time has no influence on both strategies.

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.

RQ4. How does the use of a more accurate reference influence the precision and recall of the time-based strategy? In the previous research questions, we assessed the strategies considering the release's base release as a reference. However, Moreno et al. [3, 4] has suggested that a developer may choose the best time interval to analyze the release. In this research question, we investigate if using the release's first commit influences the precision and recall of the time-based strategy. We found that using release's first commit as reference achieves a significantly lower precision ($\mu = 91.32\%$ vs. $\mu = 98.59\%$, with a large effect size), but a significantly higher recall ($\mu = 91.89\%$ vs. $\mu = 100.00\%$, with a large effect size).

Nevertheless, neither the time-based strategy nor the range-based strategy achieves perfect results. Hence, we implemented our baseline algorithm in a free and open-source release mining tool named Releasy. Releasy may allow developers and researchers to use the baseline algorithm as a strategy to assign commits to releases.

Finally, we discuss each strategy's limitations and explain why they may include errs in the commit assignment to releases. Furthermore, we discuss how developers should choose the strategies and what they should be aware of when using a given strategy.

This paper extends our previous work on assessing the time-based and range-based strategies [17]. In particular, the novel contributions are:

1. We included the *cycle time* as a new factor in RQ2;
2. We included the RQ4 to investigate whether it is possible to achieve better results adjusting the reference of the time-based strategy;
3. We evaluated accuracy and performance of the algorithm we devised to create our baseline.
4. We included a discussion about the limitations of each strategy;

5. We included a discussion to guide stakeholders in choosing a strategy; and
6. We conducted a qualitative analysis in the releases that achieved low precision or recall to discover patterns that may jeopardize the strategies to assign commits to releases.

This paper is organized into eight other sections besides this introduction. In Section 2, we explain some key version control concepts. In Section 3, we detail the materials and methods of our research. In Section 4, we show our results and answer our research questions. In Section 5, we present the *Releasy* tool, which implements the baseline algorithm, and evaluate its execution time. In section 6, we discuss the findings of our paper. In Section 7, we discuss the threats of the validity of our results. In Section 8, we present the related work. Finally, in Section 9, we conclude our work and highlight some future work.

2 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.

2.1 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 [18]. 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.

2.2 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.

2.3 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.

3 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.

3.1 Release Mining Strategies

We found works in the literature that assign commits to releases considering the commit timestamps [3, 9, 7, 4, 19], called time-based strategies, and considering the range of commits [20, 21, 22], called range-based strategies.

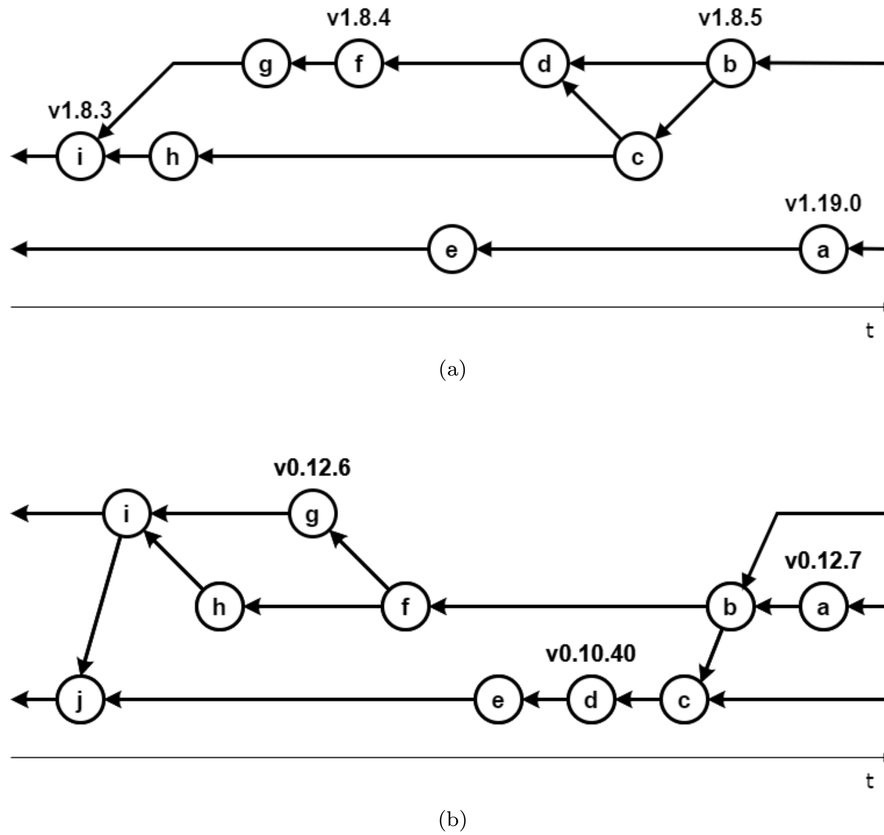
The strategies prune the project's commit graph to retrieve the sub-graph of commits that belong to a given release. Both strategies use the release's tag to assign the first commit that composes the sub-graph and a reference to recognize which commits must belong to the sub-graph. The reference accomplishes 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.

3.1.1 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. Since we provided the timestamp as a stop condition, Git does not need a tag range because it will stop assigning commits as soon as it reaches a commit with an older timestamp.

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



(a)

(b)

Fig. 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 --until 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.

3.1.2 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 [22] uses this approach to compare releases. Also, Chacon and Straub [21] 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".

3.2 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) [23]. 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[2]. 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 general-purpose programming languages [24]. 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.

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

Table 1: 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	1	0	-	1

On September 17$^{\text{th}}$, 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 [25, 26] and produced a balanced corpus, preventing including projects that would be irrelevant to the analysis or 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. Projects with few commits: we removed projects with less than 2,000 commits to avoid immature projects;
3. Non-software projects: we removed projects comprising only documentation and websites, which we identified by manually analyzing the repository;
4. Projects with too many commits: 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, which we identified by manually analyzing the release prefixes on projects using more than one release prefix;
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 round. Table 1 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 2 show the corpus' characteristics per programming language, including the number of stars, the number of commits in the main branch, and the number of releases.

3.3 Data processing

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

Table 2: 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

First, we identified the releases on the Git repository by applying a regular expression that finds the version number in the tags' names. 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 semantics, 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 the semantic versioning pattern. We also found 43 tags with different versions referencing 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 the 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".

3.4 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. The algorithm retrieves all the release tags and sorts them ascending by date. Next, it selects the older release to start assigning the commits. It assigns all the reachable commits to the release and removes those commits from the commit graph. Hence, once the algorithm assigns a commit to a release, it will not assign that commit to other releases. The algorithm continues assigning commits, observing the release timestamps order until there is no other release to assign commits.

For instance, there are four releases in Fig. 1a. The algorithm selects the release “v1.8.3” because it is the oldest release considering its tag's timestamp. Then, it assigns the reachable commits $\{i\}$ and removes those commits from the commit graph. The algorithm selects the next release, which is the “v1.8.4”. It assigns the remaining reachable commits $\{f, g\}$. It does not assign the commit $\{i\}$ because it was already assigned to the previous release and was removed from the commit graph. The algorithm selects the release “v1.8.5” and assigns the remaining reachable commits $\{b, c, d, h\}$. It does not assign the commit $\{f, g, i\}$ because they were already assigned to the previous releases and were removed from the commit graph. Finally, the algorithm selects the release “v.19.0” and assigns the remaining reachable commits $\{a, e\}$.

Similarly, there are three releases in Fig. 1b. The algorithm selects the release “v0.12.6”, which is the oldest, and assigns the reachable commits $\{g, i, j\}$. Then, it selects the release “v0.10.40” and assigns the remaining reachable commits $\{d, e\}$. It does not assign the commit $\{j\}$ because it was already assigned to the previous release and was removed from the commit graph. Finally, the algorithm select the release “v0.12.7” and assigns the remaining reachable commits $\{a, b, c, f, h\}$. It does not assign the commit $\{d, e, g, i, j\}$ because they were already assigned to the previous releases and were removed from the commit graph.

Unfortunately, no Git command implements this algorithm, and the algorithm demands analyzing the whole repository history, which is impractical to be conducted manually. It 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.

Nevertheless, we implemented the baseline algorithm to mine all the releases in our *corpus* and generate the list of commits considering each strategy and the baseline. Thus, we could compare the strategies using our baseline.

Since we devised the baseline algorithm and it may be prone to errors, we assessed its reliability to compare the strategies to assign commits to releases. The experiment manually assesses releases from our *corpus* and compares the results with the baseline. We created two lists of commits for each assessed release: the list of commits manually assigned and the list of commits assigned by the baseline algorithm. Then, we compared the lists to check the accuracy of the baseline algorithm.

Since our *corpus* comprises 13,419 releases, it would not be feasible to assess all releases manually. Hence, we sampled releases from our *corpus*. We used Cochran's equation to calculate the sample size [27]. Assuming maximum variability ($p = 0.5$), 95% confidence level, and ±10% error, we calculated the sample size as 96 releases, which we rounded to 100 releases. It is worth noticing that a similar sampling strategy has been used in another software engineering study [28]. Finally, we randomly selected one project for each programming language in our *corpus*, and we randomly selected ten releases for each project. Table 3 shows the selected releases.

The sample comprises 12,366 commits according to the baseline algorithm. Also, the sample comprises releases with and without false-positives and false-negatives commits, considering the time-based and range-based strategies and our baseline as a reference. We calculated the precision, recall, and F-measure of the time-based and range-based strategies using our sample and presented the results in Table 4. The results were coherent with the overall results presented in Table 5. Hence, we could conclude that we have a proper sample to evaluate the baseline algorithm.

We created the manually assigned list of commits by exploring the commit graph. We identified the release's boundaries and defined the list of commits that belong to each release. We used the *gitk* command to visualize the commit graph and to help the analysis. We also used the *git log* command to support the analysis and ensure we assign all the commits visualized through the *gitk* command. It is worth noting

Table 3: Releases manually assessed

Project	**Language**	**Releases**
tmux/tmux	C	0.8, 1.0, 1.1, 1.2, 1.3, 1.5, 2.3, 2.4, 2.6, 3.0
MaterialDesignInXAML/ MaterialDesignInXamlToolkit	C#	2.6.0, 3.1.1, v1.5.0, v2.0.0, v2.1.0, v2.2.0, v2.3.0, v3.0.0, v3.1.3, v3.2.0
electron/electron	C++	v0.10.4, v0.11.7, v0.15.5, v0.20.2, v0.32.0, v1.6.2, v4.0.5, v4.2.0, v6.0.0, v8.0.3
v2ray/v2ray-core	Go	v0.13, v0.14.1, v0.6.1, v1.14, v2.16.4, v2.19, v2.20, v3.30, v4.16.1, v4.20.0
elastic/elasticsearch	Java	v0.90.3, v0.90.6, v0.90.9, v1.4.4, v1.4.5, v5.4.0, v5.4.1, v5.4.2, v7.6.2, v7.9.1
vercel/next.js	JavaScript	1.1.1, 1.1.2, 2.1.1, 2.2.0, 2.4.4, 3.0.2, 3.2.3, 4.1.2, v8.0.4, v9.3.0
laravel/laravel	PHP	v3.0.4, v3.2.5, v4.0.5, v5.1.1, v5.2.27, v5.3.0, v5.4.23, v5.4.3, v5.6.21, v6.2.0
XX-net/XX-Net	Python	1.13.6, 1.3.6, 3.10.8, 3.11.9, 3.14.2, 3.6.3, 3.6.9, 3.8.0, 3.8.2, 4.3.0
CocoaPods/CocoaPods	Ruby	0.0.7, 0.16.0, 0.22.3, 0.3.0, 0.3.1, 0.35.0, 0.38.2, 0.5.0, 1.7.1, 1.7.5
microsoft/TypeScript	TypeScript	v1.8.7, v2.1.4, v2.1.5, v2.4.1, v2.5.3, v2.6.0, v2.7.1, v2.7.2, v3.2.1, v3.7.4

Table 4: The overall precision, recall, and F-measure of the assessed sample according to each strategy.

Strategy	Precision	Recall	F-measure
Time-based	99.92%	89.79%	92.10%
Range-based	99.95%	100.00%	99.97%

that the manually assigned list of commits was created by one researcher and verified by another. Thus, we double-checked the list of commits to avoid any construction bias. Both researchers achieved the same list of commits for all the assessed releases. The list of manually assigned commits is available in our replication package.

We created the list of commits assigned by the baseline algorithm using the *Releasy* tool (see Section 3.4). We sorted the lists by the commits' hashes. Next, for each release, we used the *diff* tool to compare the lists and identify miss-assigned commits from our baseline algorithm, considering the manually assigned list as a reference.

All the commits assigned by the baseline algorithm were correctly assigned, considering the manually and double-checked generated list of commits. Thus, we could conclude that the baseline algorithm is suitable for building the ground truth to compare the strategies for assigning commits to a release.

3.5 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 projects with many commits, 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 [29]. 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 [30] 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.

3.6 Research Questions

In this section, we present the four research questions that guided our study.

3.6.1 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. We interpreted the magnitude using the thresholds provided by Romano et al. [31], i.e. $|d| < 0.147$ "negligible", $|d| < 0.33$ "small", $|d| < 0.474$ "medium", otherwise "large".

3.6.2 How do the number of unique developers, the number of base releases, and the cycle time 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 or rapid release development. 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 scenarios.

We choose to investigate metrics related to parallel work. We used two factors: the *number of unique developers* and the *number of base releases*. A higher *number of unique 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 also choose to investigate metrics related to rapid release development. We used one factor: the *cycle time*, which is the time between the release's base release creation and the release creation [9]. When the

release has multiple base releases, we used the base release with the previous semantic version available when the release in the analysis was created, as done in Section 3.3.

We calculated the Spearman correlation of the *number of unique developers* with the number of commits in a release and we could observe that it is high ($\rho = 0.7410$), according to Hinkle et al. [32]. The high correlation helped us realize that analyzing the *number of unique developers* alone might be inappropriate, as large releases in terms of commits naturally have more developers. Hence, we normalized the *number of unique 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 we could observe that it is low ($\rho = 0.3396$), according to Hinkle et al. [32]. As the *number of base releases* does not increase in large releases, we did not normalize the *number of based releases*. Finally, we did not calculate the Spearman correlation of the *cycle time* and the number of commits because both metrics are likely to express the duration of the project. Hence, we did not normalize the *cycle time*.

Thus, we adopted the following metrics, extracted per release under analysis:

- *Number of unique 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.
- *Cycle time*: the days elapsed between the release's base release creation and the release creation.

We calculate all metrics for all releases. Then we calculated the mean of the *number of unique developers per 100 commits* ($\mu = 25.27$), the *number of base releases* ($\mu = 1.33$), and the *cycle time* ($\mu = 28.05$) considering all releases. Next, we used the mean of the *number of unique 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). Likewise, we used the mean of the *cycle time* to divide the releases into two groups (two treatments): the group with *rapid releases* (lesser than the mean) and the group with *traditional releases* (greater than or equal to the mean).

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:

$\mathbf{H_0^{2a}}$: *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.*

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

3.6.3 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 3.1.1, 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).

3.6.4 How does the use of a more accurate reference influence the precision and recall of the time-based strategy? (RQ4)

In the previous research questions, we assessed the strategies using the release's base release as a reference. Since the base release is available in the Git repository, a stakeholder may recover the release's base release tag and use its timestamp as a reference. However, Moreno et al. [3, 4] has suggested that the use of the base release as a reference is an approximation because the developers could start working on the release before finishing its base release. Hence, a skilled developer may choose the best time interval to analyze the release.

In this research question, we considered that the release development starts on the release's first commit. However, since the project may include parallel development, stakeholders who did not participate in the development may have difficulty identifying the first commit of the release. Hence, we used our baseline to identify the first commit of each release, which would be similar to contacting a skilled developer who participated in the release development.

This research question assesses the time-based strategy using both the release's first commit and base release as a reference. We would like to investigate whether changing the reference of the time-based strategy may achieve better results. We mined the releases using the time-based strategy considering two references, compared the results, and tested the following null hypothesis:

$\mathbf{H_0^4}$: *There is no significant difference between the mean of precision and recall on commit assignment to releases using the time-based strategy considering the release's base release and the release's first commit as a reference*

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).

4 Results

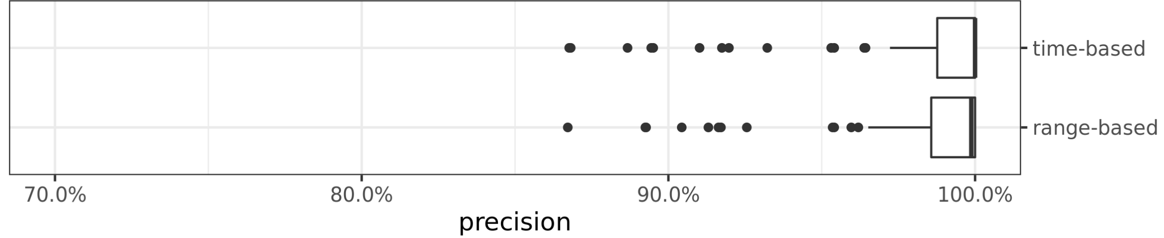
In this section, we present our experiment results and answer the research questions.

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

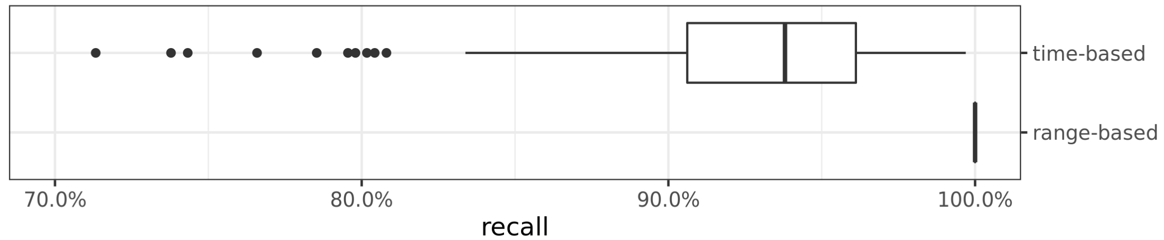
The Wilcoxon test returned p-value $= 0.0851$ for precision and p-value < 0.0001 for recall. Thus, we could not reject H_0^1 for precision but could reject for recall. The Cliff's Delta test returned $d = 1.0000$,

Table 5: 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%



(a)



(b)

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

which represents a large effect size. Thus, we could conclude that the strategies have similar precision but significantly different recall. Table 5 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.

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 (μ = 98.58% *vs.* μ = 98.62%, respectively), but the time-based strategy has lower recall (μ = 91.89%) than the range-based

Table 6: The overall precision, recall, and F-measure of the factors *number of unique developers*, *base releases* and *cycle time*.

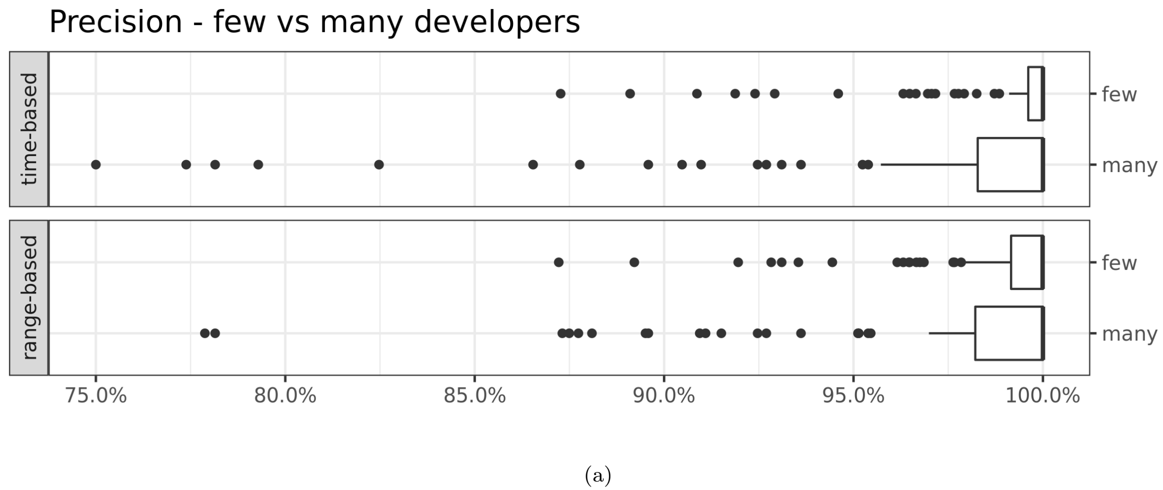
Factor	Strategy	Treatment	Precision	Recall	F-measure
Developers	Time-based	few	99.01%	89.19%	90.96%
		many	97.59%	96.82%	96.31%
	Range-based	few	98.92%	100.00%	99.20%
		many	97.82%	100.00%	98.19%
Base releases	Time-based	single	98.64%	95.13%	95.41%
		multiple	96.92%	71.39%	76.12%
	Range-based	single	98.83%	100.00%	99.02%
		multiple	97.45%	100.00%	98.32%
Cycle time	Time-based	rapid	99.98%	90.79%	92.13%
		traditional	97.96%	91.62%	92.47%
	Range-based	rapid	99.18%	100.00%	99.38%
		traditional	97.84%	100.00%	98.29%

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.

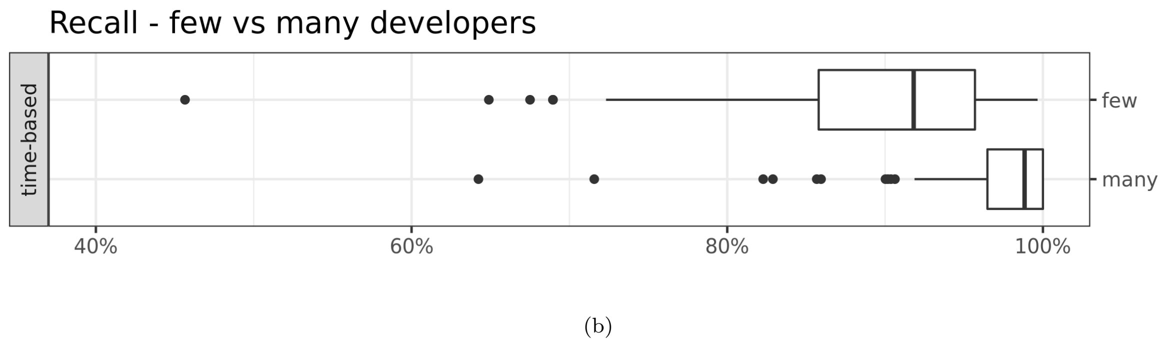
4.2 How do the number of unique developers, the number of base releases, and the cycle time influence the precision and recall of the time-based and range-based strategies? (RQ2)

In the analysis of the factor *number of unique 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, 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 unique developers* has negligible influence in the precision of both strategies but a large influence in the recall of time-based strategy. Table 6 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 unique 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*, the Wilcoxon test returned p-value $= 0.6529$ for precision of the time-based strategy, p-value $<$ 0.0001 for the recall of the time-based strategy, and p-value $= 0.0898$ for the precision of the range-



(a)



(b)

Fig. 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 unique developers*.

based strategy. Again, we did not test the recall of the range-based strategy because both treatments 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 6, 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*.

In the analysis of the factor *cycle time* and the treatments *rapid releases* and *traditional releases*, the Wilcoxon test returned p-value $= 0.9000$ for precision of the time-based strategy, p-value $= 0.3000$ for the recall of the time-based strategy, and p-value $= 0.6000$ for the precision of the range-based strategy. We could not reject H_0^{2c}. Again, we did not test the recall of the range-based strategy because both treatments have $\mu = 100\%$. In Table 6, we could observe that the precision of both strategies is higher with the

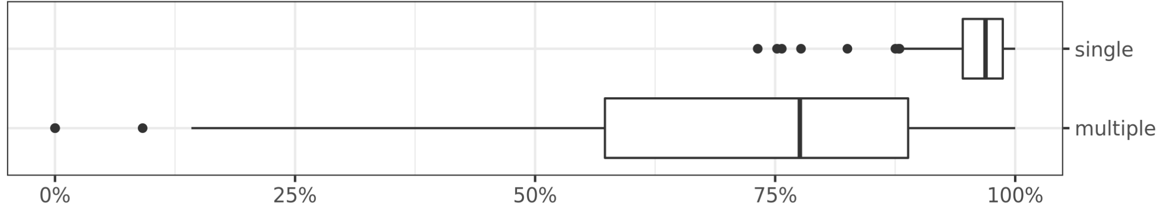


Fig. 4: The comparison of the macro averaged mean of recall for time-based strategy considering the factor *number of base releases.*

treatment of *traditional releases*. Also, the recall of time-based strategy is higher with the treatment of *traditional releases*. However, these differences are not statistically significant. Hence, we could conclude that the factor *cycle time* does not influence the precision and recall of both strategies.

RQ2. How do the number of unique developers, the number of base releases, and the cycle time 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 raise 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. Moreover, the cycle time does not influence the strategies.
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*.

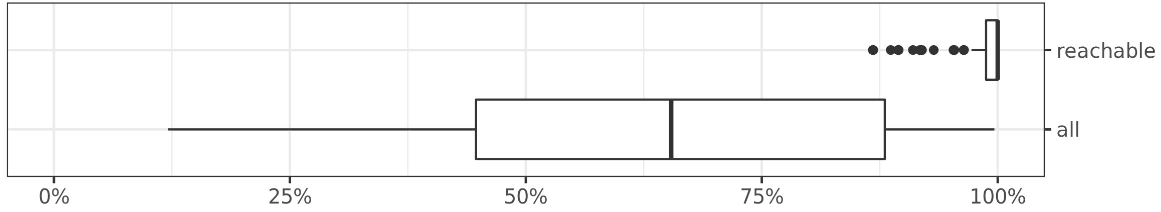
4.3 How does the inclusion of all the commits available in a specific time interval influence the precision and recall of the time-based strategy? (RQ3)

The Wilcoxon test returned p-value < 0.0001 for precision. Although only 13% of the projects presented differences in the recall, the Wilcoxon test returned p-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 7 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.

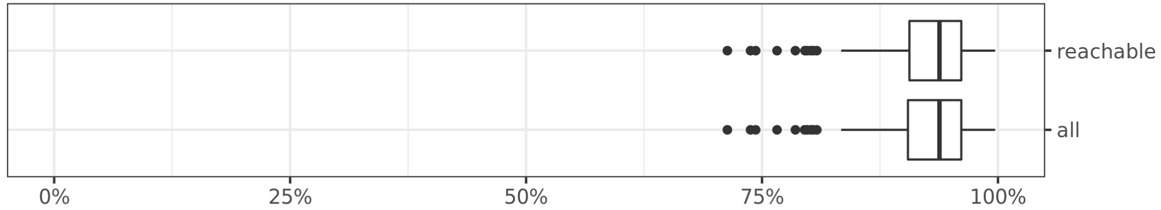
Fig. 5a shows the distribution of the projects' precision using the time-based strategy, including just the reachable commits and including all commits available in a specific time interval. We could observe that the precision is lower for the majority of projects using the time-based strategy, including 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.

Table 7: 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%



(a)



(b)

Fig. 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.

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

Table 8: The overall precision, recall, and F-measure of the time-based strategy, using the release's base release and the release's first commit as reference

Strategy	Precision	Recall	F-measure
Time-based (base release)	98.58%	91.89%	92.94%
Time-based (first commit)	91.32%	100.00%	93.77%

(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.

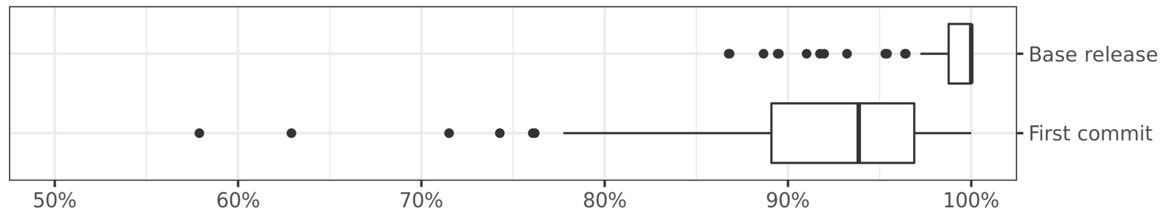
4.4 How does the use of a more accurate reference influence the precision and recall of the time-based strategy? (RQ4)

We calculated the time-based strategy macro averaged mean precision, recall, and F-measure using the release's base release and the release's first commit as reference. Table 8 shows the results. We could observe that when a skilled developer adjusts the reference to the release's first commit, the time-based strategy assigns all the commits to its actual release, achieving 100% recall. However, the new reference also assigns false-positives commits to the release, jeopardizing the strategy precision. This occurs when the development of the release under analysis started before the end of the previous release, thus having some commits in parallel.

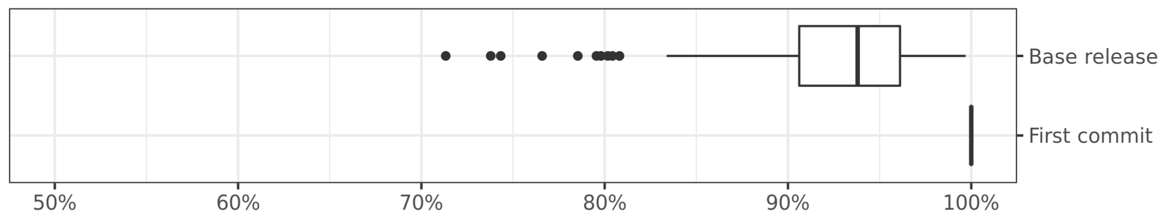
Figure 6 shows the difference between the results of the time-based strategy using each reference (base release and first commit). We could observe that there is an inversion between precision and recall. The time-based strategy using the release's base release as reference achieves a higher precision, with a significant difference (p-value < 0.0001) and large effect size ($d = 0.7750$). Nevertheless, the time-based using the release's first commit as reference achieves a higher recall, with a significant difference (p-value < 0.0001) and large effect size ($d = 1.0000$). Hence, we could reject H_0^4 for both precision and recall.

Finally, we tested the F-measure to check which time-based strategy's reference would achieve a higher overall result. Table 8 shows that the first commit reference achieve a slightly higher F-measure. However, the difference is not significant (p-value $= 0.1632$). Hence, developers using the time-based strategy should choose between the references, considering whether they need higher precision or recall in their analysis. On the one hand, using the base release as reference would let to missing some relevant commits, but not adding much irrelevant commits. On the other hand, using the first commit as reference would let to not missing any relevant commit, but adding some irrelevant commits in the analysis. Since it is easier to use the release's base release as a reference, developers will likely use the time-based strategy with the release's base release as a reference.

RQ4. How does the use of a more accurate reference influence the precision and recall of the time-based strategy?
Answer: We could observe that the time-base strategy using the release's first commit as a reference achieves a significant lower precision ($\mu = 91.32\%$ vs. $\mu = 98.59\%$, with a large effect size), but a significantly higher recall ($\mu = 91.89\%$ vs. $\mu = 100.00\%$, with a large effect size). There is no significant



(a)



(b)

Fig. 6: The comparison of the macro averaged mean of precision (a) and recall (b) for the time-based using the base release and the first commit as reference.

difference between the references F-measure.
Implications: Developers using the time-based strategy may use the first commit as a reference to achieve a higher recall at the expense of a lower precision.

5 *Releasy* tool execution time

We assessed the baseline algorithm execution time to evaluate whether it is feasible for developers to use it during their daily activities. The algorithm was implemented in tool named *Releasy*[3]. *Releasy* is a free and open-source Python library that inspects Git repositories to fetches release information, such as the commits that belong to each release. The *Releasy* documentation is available in its Git repository hosted on GitHub. Briefly, the user needs to create a Python program that imports and instantiates the library, points to the location where the Git repository to be analyzed is available, executes the main *Releasy* function to process the repository, and retrieve its release information. *Releasy* may enable novel works in the literature to use our baseline algorithm.

We assessed the *Releasy*'s execution time to evaluate whether it is feasible for developers to use the baseline algorithm during their daily activities. We intend to provide insight into whether *Releasy* is fast enough for interactive usage.

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

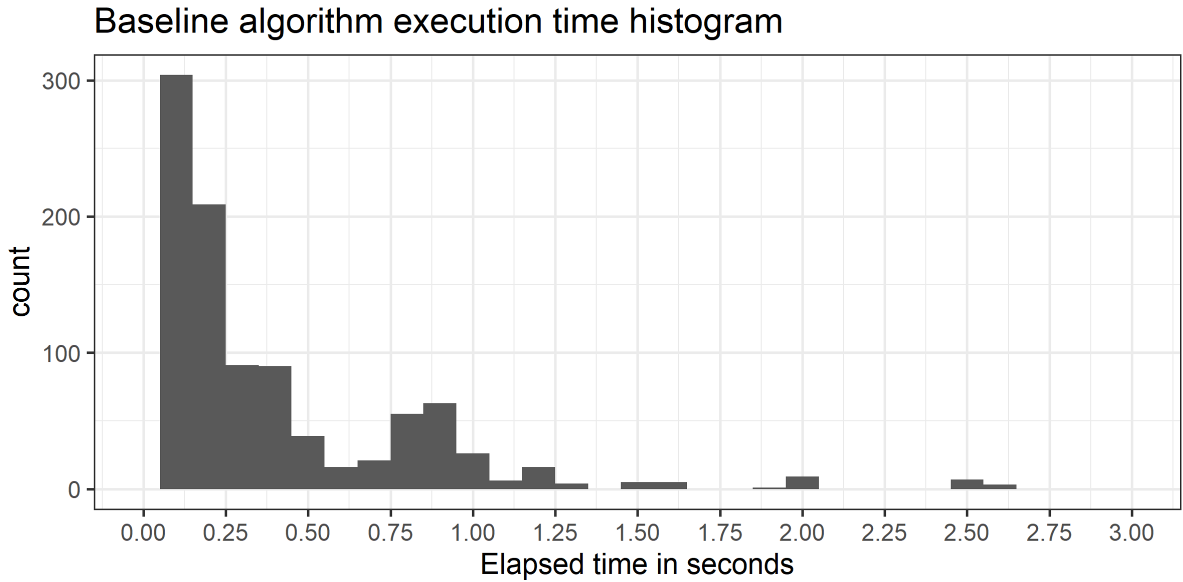


Fig. 7: The execution time histogram of *Releasy*

It is worth noting that the time-based and range-based strategies can be executed using Git, which was developed using the C programming language and has been highly optimized since its creation. *Releasy* was implemented using python in the context of our research and without any special performance optimization. Hence, we do not intend to directly contrast the strategies and *Releasy* in terms of execution time. We intend to put the execution time results in perspective with the other strategies that use Git.

The experiment consists in using *Releasy* to analyze the projects from our *corpus* and measure the time taken from the start to the end of the analysis. We ran the experiment in a dedicated workstation with the following configuration: Intel i7-8700K processor (6 cores, 12 threads, 3.7 GHz), 32 GB RAM (2,666 MHz DDR4), RAID 0 SSD disk (Samsung 960 PRO NVMe M.2), GeForce GTX 480 video card, and ASUS TUF Z370-PLUS GAMING motherboard.

We run the analyses without parallelism to avoid one project analysis influencing the execution time of the other projects. We also monitored the workstation to guarantee that no other relevant process would be in execution during the experiment. We repeated the experiment 10 times, totalizing 1,000 measurements (10 for each project in our *corpus*). This approach reduces the risk of an unpredictable process influencing our experiment.

Figure 7 shows the distribution of the execution time in a histogram. The execution time varied from 0.0323 seconds on the *vercel/hyper* project to 2.5763 seconds on the *elastic/elasticsearch* project. The *execution time* mean was 0.3963 seconds, the standard deviation was 0.4219 seconds, and the median was 0.2132. According to Nielsen [33], response times up to one second is the limit for the user's flow of thought to stay uninterrupted, and ten seconds is the limit for keeping the user's attention. Since we could observe that most of the executions (93.20%) run in less than one second, it is likely that our algorithm is appropriate for developers to use in their daily activities.

Moreover, aiming at understanding the scalability of *Releasy*, we calculated the Spearman correlation of the *execution time* with the following projects' characteristics: *number of releases*, *number of commits*, *number of merges*, and *number of unique developers*. Table 9 shows the results. According to Hinkle et al. [32], we

Table 9: The Spearman correlation of *Releasy*'s performance and the projects' characteristics

	Releases	**Commits**	**Merges**	**Developers**
ρ	0.4520	0.9797	0.5114	0.4766

Table 10: Execution time (in seconds) of the project *elastic/elasticsearch* using Git and *Releasy*

Tool	**Mean**	**Std**	**Median**
Git	1.6732	0.0243	1.6665
Releasy	2.5300	0.0248	2.5200

could observe that the correlation of the *execution time* and the *number of commits* is very high ($\rho = 0.9797$). We could also observe that the correlation of the *execution time* with the *number of merges* is moderated ($\rho = 0.5114$) and with the *number of releases* and *number of unique developers* are low ($\rho = 0.4520$ and $\rho = 0.4766$, respectively). This indicates that the scalability of *Releasy* is mainly influenced by the number of commits in the project. Nevertheless, it was able to process the *elastic/elasticsearch* project, the biggest project in our *corpus*, comprising 55,176 commits in the main branch and 88,299 commits altogether, in just 2.5763 seconds.

Finally, we measured the execution time using Git commands to put the results in perspective. We opted to use the range-base strategy because it achieved the highest values for F-measure. We ran the range-based strategy 10 times for the project *elastic/elasticsearch*, which was the project with the longer execution time using *Releasy*. We calculated the mean, standard deviation, and median of the execution time. Table 10 shows the execution time of the range-based strategy using Git and the baseline algorithm using Releasy.

As expected, Git is faster than *Releasy*. However, as we previously stated, *Releasy* execution time is still feasible for developers to use in their daily activities.

6 Discussion

6.1 The Strategies Limitations

This section discusses the limitations we have observed in the time-based and range-based strategies. Knowing their limitations help understand why there is a reduction in the precision and recall of the assignment of commits to releases. This discussion also helps to explain why the chosen baseline is a good fit.

First, the time-based and range-based strategies use only two references to compute the commits that belong to a release: the reference to the supposed beginning and the ending of the release. Both strategies use the release tag to assign the last release commit (i.e., the ending of the release) and a reference to the beginning of the release, to recognize which commits must belong to the release. Since a release may have more than one base release, the commits assigned may vary according to the choice. For instance, in Fig 1b, when analyzing the release "v0.12.7", if we use 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". Nevertheless, if we use the release "v0.10.40" as reference, the range-based strategy would assign the commits $\{a, b, c, f, g, h, i\}$. Hence, errors in the reference selection may impair the results of the time-based and range-based strategies. Since the baseline considers all the tags, it does not have problems with multiple base releases. It is worth noting that the time-based strategy can only use two references, but the range-based strategy can use more references and mitigate this limitation. However, users still need to know which additional references to

include, which may involve a complementary strategy that analyzes the commit graph and discovers the base releases that should be included as references. For instance, in 1b, the user must know that both "v0.12.6" and "v0.10.40" should be included as a reference to achieve similar results to the baseline.

Second, the time-based and range-based strategies also evaluate the releases individually. They ignore if a commit was already assigned to another release, which may lead to assigning a single commit to more than one release. For instance, in Fig 1b, the time-based strategy would assign the commit $\{d, e\}$ to releases "v0.12.7" and "v0.10.40", if we consider "v0.12.6" as reference. Since the baseline algorithm tracks the commits assigned to previous releases, it prevents assigning a single commit to more than one release.

Third, the time-based strategy relies on the commits and tags timestamps. In Git, the commit timestamp is obtained from the committers' computer. There is no validation on the repository regarding the correctness of this information. Moreover, developers can create an annotated 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. This problem partially affects the baseline because we use the tag timestamp to sort the releases. It does not impact the range-based strategy because it does not use timestamps to assign commits to releases.

Finally, the time-based strategy, the range-based strategy, and the baseline algorithm are impacted by unreachable code, i.e., the code developed and committed to the repository but not actually released. For instance, a project using feature toggles may commit and deliver a release to production with a disabled feature. The strategies would assign the commits related to the disabled feature to the release. Hence, the respective release notes would include a feature that was coded but is not available in the release for the users. The same problem happens on a repository that hosts multiple projects. The strategies may assign the commits of one project into a release of another project.

6.2 Choosing a strategy and knowing its implications

This section discusses when to use each strategy and aims at guiding developers and researchers to choose the best strategy. This work compared the time-based and range-based strategies regarding their precision and recall in relation to our baseline algorithm. The result suggests that stakeholders should prioritize using our baseline algorithm, then the range-based strategy, and then the time-based strategy.

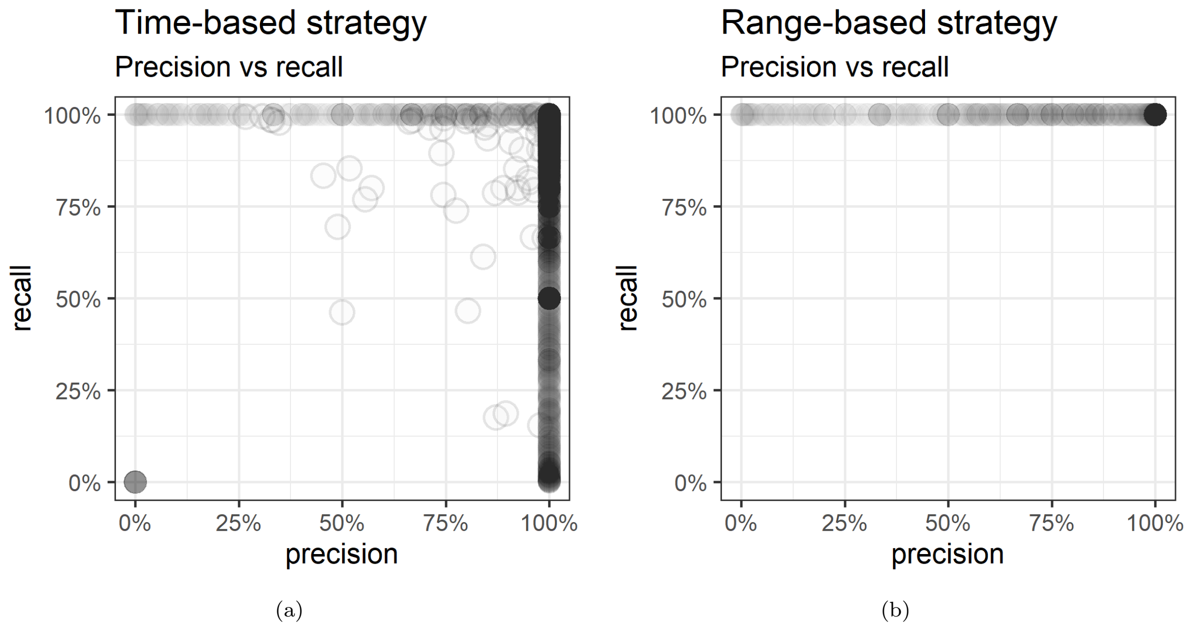
The baseline algorithm should be preferred because it overcomes some of the limitations of the time-based and range-based strategies. Moreover, the baseline algorithm does not include any drawbacks not already present in the other strategies. Also, the baseline algorithm is feasible for daily use in terms of execution time.

However, the baseline algorithm is a novel strategy that is not widespread. Hence, developers and researchers aiming at using the baseline algorithm would need to install the aforementioned Releasy tool or implement the algorithm themselves, which is not always possible or desired.

In contrast, Git, one of the most popular version control systems available nowadays, can run both the time-based and range-based strategies natively. Stakeholders using Git can use the commands presented in Section 3.1 to run the strategies. The availability and simplicity of these strategies end up favoring their use.

Still, some applications that need to assign commits to releases may use a strategy and may not enable users to switch to other strategies. For instance, Moreno et al. [3, 4] provides a release notes generator tool that uses the time-based strategy and do not allow users to choose another strategy.

Since it is not always possible to use the best strategy, one should understand the implications of the chosen strategy in the analysis. Although not perfect, the range-based strategy achieved a macro averaged mean precision of 98.62% and recall of 100%. Hence, stakeholders using the range-based strategy should know that it may include few irrelevant commits to the analysis. Also, the time-based strategy achieved a



(a) (b)

Fig. 8: All releases' precision (a) and recall (b). The darker the circle, the greater the number of releases.

macro averaged mean precision of 98.58% and and recall of 91.89%. Hence, stakeholders using the time-based strategy should be aware that it may include few irrelevant commits and miss some relevant commits to the analysis.

6.3 Releases with low precision and recall

Until now, we presented our results using the macro averaged mean to aggregate the precision and recall per project. In this section, we analyze individual releases that achieved low precision or recall. Figure 8 shows all releases from our *corpus* and their respective precision and recall using each strategy - the darker the circle, the greater the number of releases. We could observe releases that achieved:

1. Zero precision and recall;
2. Low precision and high recall; and
3. High precision and low recall.

We inspected these releases individually to understand why they achieved such results and discuss how they might introduce errors in the analysis.

6.3.1 Releases with zero precision and recall

We could observe that the releases represented in the bottom left of Figure 8a achieved zero precision and recall using the time-based strategy. We found this behavior in 47 releases from 10 projects. We analyzed the releases and discovered two patterns that led to this issue.

We found that 40% (19 out of 47) of these releases use annotated tags, which have their own timestamps. Hence, the tag timestamp can be different from the commit timestamp. We observed that this

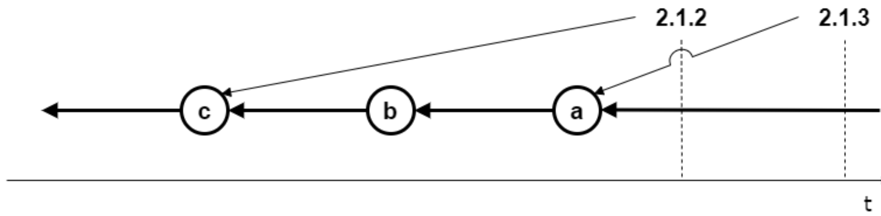


Fig. 9: The commit graph of the release 2.1.3 from the projetc dotnet/efcore.

issue happened when the base release tagging happened after all the release's commits were created. As the time-based strategy only assign commits with timestamp greater than the base release timestamp, no commit is assigned to the release, resulting in zero precision and recall. For instance, Figure 9 shows the release 2.1.3 from the project *dotnet/efcore*. The release 2.1.2 is the base release of release 2.1.3. The tag of release 2.1.3 references the commit a, and the tag of release 2.1.2 references the commit c. We would expect to assign the commits $\{a, b\}$ to release 2.1.3, but the time-based strategy fails because release 2.1.2 was tagged after all commits of release 2.1.3.

We found a similar pattern that affected the other 60% (28 out of 47) of the releases, all from the project *sebastianbergmann/phpunit*. All these releases and their base releases have the same timestamp. Hence, the time-based strategy was not able to assign the releases' commits. This pattern probably happened because the developers imported external releases all at once when created the Git repository.

As expected, this issue does not affect the range-based strategy because it traverses the commit graph instead of using the timestamps to assign commits to releases. All the 47 releases achieved 100% precision and recall using the range-based strategy.

Finally, this issue is rare and happened only in 47 releases, which represent less than 0.05% of the releases in our *corpus*. The issue is more likely to happen when developers take too long to create the release tag. Furthermore, stakeholders using the time-based strategy could use the commit timestamp instead of the tag timestamp to solve this issue.

6.3.2 Releases with low precision and high recall

We could observe that the releases represented in the top left of the figures 8a and 8b achieved low precision and high recall. We found 53 releases that achieved less than 10% precision, but 100 % recall: 90% (48 out of 53) of them achieved this result with both strategies, two releases achieved this result only with the time-based strategy, and three releases achieved this result only with the range-based strategy. We analyzed the releases and found some patterns that led to low precision.

We found that 83% (44 out of 53) of these releases did not respect the semantic order of the releases, i.e., releases named with a version lower than its base release. For instance, the release v5.3.30 from the project *laravel/framework* was created based on the release v5.4.4 instead of the release v5.3.29. Hence, stakeholders analyzing the release v5.3.30 considering the semantic order would use the release v5.3.29 as a reference and accidentally include the commits from release v5.4.4 to the analysis.

We also found in 15% (8 out of 53) of the releases that our base release selection criteria did not choose the best base release for the analysis. In Section 3.3, we explained that we used the base release as a reference for each strategy and that we would select the base release as the previous semantic version release available at the time the release in the analysis was created. We found that the actual base releases of these releases were created after the release. Hence, we selected an older release as a reference. For instance, the tag of the release v2.0.2 from project *etcd-io/etcd* was created before the tag of the release v2.0.1. Thus, our selection criteria mistakenly selected the release v2.0.0 as the base release of release v2.0.2. Although this

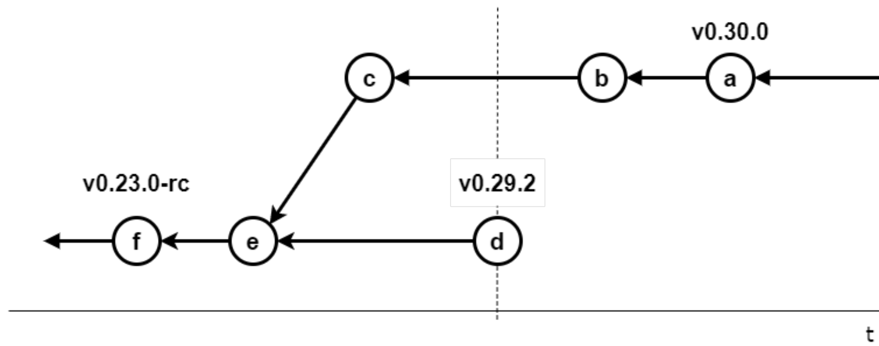


Fig. 10: The commit graph of the release v0.30.0 from the project *facebook/react-native*.

issue affected few releases and only for a short time, we consider this analysis relevant because it would also impact automatic processes triggered by events in the repository, such as those related to continuous integration.

Finally, we also found one release from the project *electron/electron* that the base release was created with the wrong timestamp. As described in Section 6.1, the time-based strategy can be jeopardized by commits and tags with wrong timestamps. For instance, the release v0.10.3 is based on the release v0.10.2. However, the release v0.10.2 was created with a wrong timestamp that was before the timestamps of some of the commits reachable by this release. Hence, the time-based strategy would include these commits from release v0.10.2 into release v0.10.3. This issue probably happened because the computer used to create the commit was with a problem in its internal clock.

All these patterns introduced false-positives to the analysis of the releases. Hence they contribute to the reduction of the precision of the strategies.

6.3.3 Releases with high precision and low recall

We could observe that the releases represented in the bottom right of the figure 8a achieved high precision and low recall. Since the range-based strategy achieved 100% recall to all releases, this issue only happens with the time-based strategy. We found 274 releases that achieved 100% precision but less than 10% recall. We sampled these releases and discovered a pattern that led to a low recall.

We found that the issue happens when part of the release history does not reach the base release tag commit. For instance, Figure 10 show the commit graph of release v0.30.0 from the project *facebook/react-native*. The time-based strategy would not include the commit c in the analysis because it happened before the base release. The range-based strategy would include it because it is not reachable from the base release "v0.29.2". We could observe the same pattern on Figure 1b when assigning commits to release "v0.12.7", where the base release "v0.12.6" does not reach the commit h.

7 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. [34].

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 strategies.

Additionally, 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 on our analysis.

Finally, we build a tool to conduct the experiments. The tool was tested and the results were verified by the researchers. However, eventual unnoticed bugs in our tool may impair the results.

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 [30]. Hence, releases with few commits will have the same weight in the analysis as releases with many commits.

In RQ2, we calculated the mean *cycle time* to separate the releases into two treatments: *rapid releases* and *traditional releases*. These names were chosen to allow a more natural explanation of the results. Despite the names, these groups were intended only to separate the releases from our *corpus* and should not be used to categorize them.

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*.

8 Related Work

In this section, we present studies that are related to our work. Although we found studies that assign commits to releases, they do not discuss the impact of misassignment of commits, do not calculate precision and recall, and do not compare different strategies. Hence, considering, specifically, the objective of our study, we did not find research efforts focused on the impact of assigning commits to releases, and, to the best of our knowledge, our work is the first attempt to investigate this subject.

However, we found studies that assign commits to related commits and to issues. We also found studies that assign issues to releases. Although these studies are not tightly related to ours, when combined they could reach a similar effect by transitivity. Thus, this section organizes the related work 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.

Commit assignment to issues. A more robust way of assigning commits to issues demands adding the issue id in the commit message. When such an id is not explicitly added during the commit, the assignment should be obtained using contextual information, which is error-prone, as detailed below.

The following studies aim at bridging the version control system and issue tracking system. Le et al. [35] 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 [36], which achieved overall 56.40% precision and 17.96% of recall.

Sun et al. [37] 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.

Both approaches have low precision and recall when compared to the approaches used to assign commits to releases. This is expected, considering that the problem of assigning commits to issues is much more unstructured than the problem of assigning commits to releases. Thus, using approaches that link commits to issues when there is no explicit issue id added in the commit message as a step of approaches that link commits to releases seems inappropriate.

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 to 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.

In both cases, the quality of the results is directly related to the quality of the manual indication of the issue id in the commit message. Nevertheless, even if 100% of the commits are correctly linked to the issues, the approaches are still subject to the imprecision of the time-based strategy when choosing which commits belong to each release.

Commit assignment to related commits. Dhaliwal et al. [38] 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 [39]. 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.

Although these works are relevant to their objectives, they are not appropriate to the problem of assigning commits to releases. Since the same feature may be released multiple times, with each release introducing new refinements over the previous release, grouping related commits and assigning them all at once to the same release would be a mistake. Nonetheless, these approaches can be seen as complementary to ours, as they can be used to filter the release history and allow detailed analyses regarding the evolution of specific features.

Commit assignment to releases Shobe et al. [20] 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. [19, 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, which is equivalent to considering 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. [40, 41] analyze patch backouts (i.e., reversing commits) on rapid releases using the time-based strategy.

Although these works suffer from the problems discussed in this paper, they at most recognize the threat but do not provide evidence on how the results were affected by such a threat. We believe that our work will help on catching the attention of researchers to this problem and on making better decisions regarding the assignment of commits to releases.

9 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. Also, stakeholders may use the release's first commit as a reference for the time-based strategy, which decreases its precision but improves its recall.

As future work, we intend to perform an in-depth study on the effect of the different strategies in release applications, such as release notes generation.

Data availability The data that support the findings of this study and the scripts we used to run our experiments are available in our replication package: https://github.com/gems-uff/release-mining-extended.

Competing Interests The authors declare that they have no competing interests. Furthermore, this publication is the sole initiative of the authors and does not represent the position of any company or organization.

Acknowledgements The authors would like to thank CNPq (grant 311955/2020-7) and FAPERJ (grant E26/201.038/2021) for their financial support.

References

1. B. Adams, S. Bellomo, C. Bird, B. Debic, F. Khomh, K. Moir, and J. ODuinn, "Release Engineering 3.0," vol. 35, no. 02, pp. 22–25.
2. S. L. Abebe, N. Ali, and A. E. Hassan, "An empirical study of software release notes," vol. 21, no. 3, pp. 1107–1142. [Online]. Available: https://doi.org/10.1007/s10664-015-9377-5
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, pp. 484–495. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635870
4. L. Moreno, G. Bavota, M. Penta, R. Oliveto, A. Marcus, and G. Canfora, "ARENA: An Approach for the Automated Generation of Release Notes," vol. 43, no. 02, pp. 106–127.
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*. [Online]. Available: http://arxiv.org/abs/1809.10265
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. ACM, pp. 1256–1266. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660320

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," vol. 20, no. 5, pp. 1384–1425. [Online]. Available: https://doi.org/10.1007/s10664-014-9338-4
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)*, pp. 179–188.
9. F. Khomh, B. Adams, T. Dhaliwal, and Y. Zou, "Understanding the impact of rapid releases on software quality," vol. 20, no. 2, pp. 336–373. [Online]. Available: https://doi.org/10.1007/s10664-014-9308-x
10. J. Tsay, H. K. Wright, and D. E. Perry, "Experiences Mining Open Source Release Histories," in *Proceedings of the 2011 International Conference on Software and Systems Process*, ser. ICSSP '11. ACM, pp. 208–212. [Online]. Available: http://doi.acm.org/10.1145/1987875.1987911
11. Martin. How to find out if a Git commit is included in a release? Stack Overflow. [Online]. Available: https://stackoverflow.com/q/32852374/1090745
12. Savrige. How to get a list of commits related to a specific release? Stack Overflow. [Online]. Available: https://stackoverflow.com/q/54787120/1090745
13. BenMorel. How to find out which release(s) contain a given GIT commit? Stack Overflow. [Online]. Available: https://stackoverflow.com/q/27886537/1090745
14. AnsFourtyTwo. How to find commit of first release in Git repository. Stack Overflow. [Online]. Available: https://stackoverflow.com/q/58766813/1090745
15. F. Pinto, "On the Impact of Rapid Release in Software Development."
16. C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, pp. 1–10.
17. F. C. d. R. Pinto, B. Costa, and L. Murta, "Assessing time-based and range-based strategies for commit assignment to releases," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 142–153.
18. T. Preston-Werner. Semantic Versioning 2.0.0. Semantic Versioning. [Online]. Available: https://semver.org/
19. 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*, pp. 20–29.
20. 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. Association for Computing Machinery, pp. 68–71. [Online]. Available: https://doi.org/10.1145/2597008.2597792
21. S. Chacon and B. Straub, *Pro Git*. Springer Nature.
22. GitHub. Comparing releases - GitHub Docs. [Online]. Available: https://docs.github.com/en/free-pro-team@latest/github/administering-a-repository/comparing-releases
23. H. Borges and M. Tulio Valente, "What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform," vol. 146, pp. 112–129. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121218301961
24. P. Mayer and A. Bauer, "An empirical analysis of the utilization of multiple programming languages in open source projects," in *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '15. Association for Computing Machinery, pp. 1–10. [Online]. Available: https://doi.org/10.1145/2745802.2745805
25. 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*, pp. 1–10.
26. 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, pp. 92–101. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597074
27. G. D. Israel, "Determining sample size."
28. J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "Understanding and improving the quality and reproducibility of Jupyter notebooks," vol. 26, no. 4, p. 65. [Online]. Available: https://doi.org/10.1007/s10664-021-09961-9
29. K. Beck, *Extreme Programming Explained: Embrace Change*. addison-wesley professional.
30. C. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press.
31. J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys," in *Annual Meeting of the Florida Association of Institutional Research*, vol. 177, p. 34.
32. D. E. Hinkle, W. Wiersma, and S. G. Jurs, *Applied Statistics for the Behavioral Sciences*. Houghton Mifflin College Division, vol. 663.
33. J. Nielsen, *Usability Engineering*. Morgan Kaufmann.
34. C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer Science & Business Media.
35. 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*,

pp. 36–47.
36. 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. Association for Computing Machinery, pp. 1–11. [Online]. Available: https://doi.org/10.1145/2393596.2393671
37. Y. Sun, Q. Wang, and Y. Yang, "Frlink: Improving the recovery of missing issue-commit links by revisiting file relevance," vol. 84, pp. 33–47.
38. 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)*, pp. 202–211.
39. M. Hammad, "Identifying related commits from software repositories," vol. 51, no. 3, pp. 212–218.
40. R. Souza, C. Chavez, and R. A. Bittencourt, "Do Rapid Releases Affect Bug Reopening? A Case Study of Firefox," in *2014 Brazilian Symposium on Software Engineering*, pp. 31–40.
41. ——, "Rapid Releases and Patch Backouts: A Software Analytics Approach," vol. 32, no. 2, pp. 89–96.