# Recovering Revisions of Pull Requests with Altered History

Gengyi Sun
University of Waterloo
Waterloo, Canada
gengyi.sun@uwaterloo.ca

Georges Aaron Randrianaina
University of Waterloo
Waterloo, Canada
georgesaaron.randrianaina@uwaterloo.ca

Tao Xiao
Kyushu University
Fukuoka, Japan
xiao@ait.kyushu-u.ac.jp

Yasutaka Kamei
Kyushu University
Fukuoka, Japan
kamei@ait.kyushu-u.ac.jp

Shane McIntosh
University of Waterloo
Waterloo, Canada
shane.mcintosh@uwaterloo.ca

## Abstract

In pull-based development, Pull Requests (PRs) undergo peer review. To address review feedback, authors may "force-push" after amending, squashing, or rebasing commits. Such revisions can conflate PR-specific changes with updates on the base branch, hindering further review and introducing noise into PR-based datasets.

In this paper, we empirically study 935,011 PRs from 1,445 well-maintained GitHub projects. We observe 568,218 force-push events in 242,624 PRs that span 1,366 projects. We denote the HEAD commits of the PR before and after force-pushing as a force-push pair, and observe that 210,837 pairs (37.1%) have a relationship that conflates PR changes with updates to the base branch.

To systematically isolate PR-specific changes in such revisions, we propose AntiDiffamine—an ancestry-based approach that aligns commits on a common reference. AntiDiffamine can automatically deconflate 98.1% of the studied revisions, decreasing the diff size by one to three orders of magnitude.

We also evaluate AntiDiffamine on fail-fix pairs, i.e., revisions that fix failing builds. A case study of Kubernetes reveals that 130 of 523 pairs have conflated diffs. Moreover, the fixing files identified when AntiDiffamine is applied overlap more with a human-labelled ground truth than when conflation is untreated or excluded.

AntiDiffamine is a first attempt at addressing force-push-based diff conflation, which is having a real-world impact on practical and research workflows. To ease practical adoption, we provide a template GitHub Action, which scans PR updates for force-push events and applies AntiDiffamine to deconflate the diff.

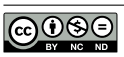
## CCS Concepts

• **Software and its engineering → Software evolution**.

## Keywords

Pull-based development, code review, software analytics

**ACM Reference Format:**
Gengyi Sun, Georges Aaron Randrianaina, Tao Xiao, Yasutaka Kamei, and Shane McIntosh. 2026. Recovering Revisions of Pull Requests with Altered History.




*ASE '26, Munich, Germany*

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837417

In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3837417

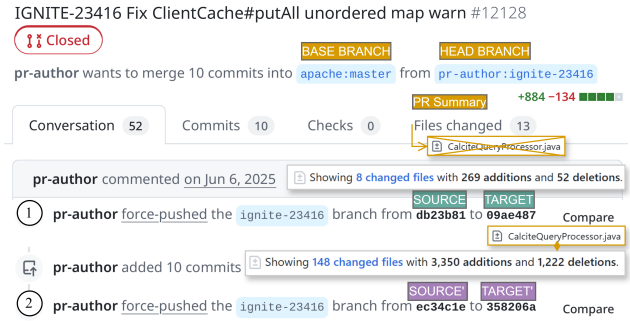
## 1 Introduction

Pull-based development is a collaborative workflow where developers propose changes through Pull Requests (PRs) that are reviewed, discussed, and integrated into a shared codebase [14]. To create a PR, developers fork the repository and record changes on a head branch corresponding to a base branch in the original repository. The PR lifecycle begins when the developer opens a PR comparing the head and base branches, followed by an iterative code review cycle, where reviewers and bots suggest improvements until maintainers grant merge approval. To incorporate review feedback, authors may append commits or rework existing ones through amending, squashing, and rebasing [14, 16, 22, 29], which requires force-pushing since rewriting commit history is inherently dangerous. While this keeps commit history linear, diffs summarizing changes before and after force-pushing may conflate PR-specific changes with base-branch updates.

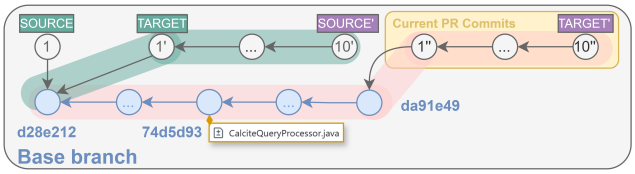
While rewriting commit history keeps history linear, diffs that compare changes before and after force-pushing may conflate PR-specific changes with base-branch updates. We refer to the `HEAD` commits of the PR before and after such a revision as a *"force-push pair"*. Figure 1 illustrates a scenario where *git diff* compares two commits (*e.g.,* `SOURCE′` and `TARGET′`) before and after a force-push (shaded regions in Figure 1b), yielding a diff conflated by base-branch commits (*e.g.,* `74d5d93`) and their changes (*e.g.,* `CalciteQueryProcessor.java`) that are not part of the PR. This conflation complicates peer review. For example, reviewers verifying whether feedback was addressed in a force-pushed revision must traverse PR revisions to filter out base-branch changes.

This conflation problem also introduces noise in datasets that rely on PRs, such as those that (1) identify changes that fix failing builds [4, 17, 18, 39], (2) compare bug-inducing and -fixing commits [5, 36], and (3) identify duplicate or competing pull requests [24, 25, 43]. For example, BugSwarm [39] includes a fail-fix pair[1] with an unusually large diff (394 changed files, with 9,603 added and 1,679 removed lines). A closer inspection reveals that the two commit contents remain the same across failing and fixing versions. The failure was fixed by an update that was received from the base branch by rebasing the PR, which is entangled with feature

[1] SonarSource-sonar-java-341335847




**(a) Pull request timeline with force-push events.**




**(b) Commit graph of the source and target commits. Arrows indicate parent-child commit relations.**

**Figure 1: PR 12128 from the apache/ignite project.**

code introduced by other developers and PRs. While such a large diff may indeed repair the build, the diff is no longer semantically aligned with the PR description.

To better understand the conflation problem, we conduct an empirical study of real-world PRs. More specifically, we address the following research questions (RQs):

**RQ1: How prevalent are force-push pairs on GitHub?**
*Motivation:* Conflation arising from force-pushing can impact development and research workflows. To estimate the potential impact magnitude, we set out to study the prevalence of force-push events.
*Results:* Section 3 shows that 1,366 of 1,445 studied projects include at least one PR with a force-push event. A total of 568,218 force-push events span 242,624 of the 935,011 studied PRs (26%).

**RQ2: What characterizes conflation-prone force-push pairs?**
*Motivation:* It is not clear whether all force-push pairs are equally prone to conflation. Therefore, we set out to characterize the force-push pairs that are conflation-prone.
*Results:* Section 4 shows that frequently syncing with the base branch to avoid merge conflicts by force-pushing produces "hetero-ancestral" relationships, conflating PR-scope changes with base branch updates. We find that 210,837 and 235,296 of 568,218 studied force-push pairs are hetero-ancestral and co-ancestral, respectively.

**RQ3: To what extent can we deconflate force-pushed revisions?**
*Motivation:* Given the prevalence of hetero-ancestral force-push pairs, there is a need for an automated solution. Hence, we set out to propose and evaluate such a solution.
*Results:* Section 5 introduces AntiDiffamine—an ancestry-based approach that systematically isolates PR-specific changes. AntiDiffamine can deconflate 98.1% of hetero-ancestral force-push pairs. Moreover, those deconflated diffs are orders of magnitude smaller than their conflated counterparts.

**RQ4: To what extent does conflation impact downstream analyses?**
*Motivation:* The identification of revisions of PRs is a key step in other analyses. Since force-push-based conflation is prevalent, we set out to evaluate its impact on downstream analytical tasks.
*Results:* Section 6 shows that diff conflation also impacts fail-fix pair analyses, and treating these pairs with AntiDiffamine substantially outperforms leaving them untreated or excluding them.

We conclude that diff conflation impacts both practitioners and researchers. In code review, diff conflation complicates PR revisions by obscuring the true scope of changes. For researchers, the complex nature of PR revisions discourages the use of this rich, iterative data, limiting the community's ability to leverage PR-based discussions and code evolution. AntiDiffamine addresses both challenges by narrowing conflated diffs to the true PR scope.

To integrate AntiDiffamine into existing developer workflows, we implement a GitHub Action that triggers automatically when a PR receives new commits. If hetero-ancestral pairs are detected in the commits of the PR, the action posts a comment on that PR that contains a URL, which includes an AntiDiffamine-treated diff along with other contextual information.

## 2 Revision History in Pull-Based Development

In this section, we define PRs (Section 2.1) and force-push events (Section 2.2). We also discuss the impact of altered revision history on practitioners (Section 2.3) and researchers (Section 2.4).

### 2.1 Pull-Based Development

Pull-based development is a common workflow supported by modern social coding platforms (*e.g.,* GitHub and GitLab), where developers often contribute to repositories by submitting a PR. To prepare a PR, a developer forks the original (*a.k.a.* upstream) repository, develops their changes in that forked repository, and submits their set of changes for integration by submitting a PR to the upstream repository. In GitHub terminology, the branch within the fork repository from which the PR is submitted is called the head branch (PR branch), and the branch within the upstream repository on which the changes should land is called the base branch. The base branch may be any branch in the upstream repository, whereas the head branch may either be a branch local to the forked repository or another branch within the same upstream repository [14].

Figure 1a shows an example of a PR in the apache/ignite repository. The contributor forked the repository apache/ignite and created the local branch ignite-23416. When the PR was submitted, GitHub compared the head branch pr-author:ignite-23416 with the base branch apache/master in the upstream repository. The PR includes all commits reachable (*i.e.,* the recorded ten commits) from the head branch, but not from the base branch [14].

### 2.2 Force-Push Events

After a PR is submitted, the upstream maintainers review it by inspecting the changes and leaving comments. The author continues to push commits to the head branch in their fork (*e.g.,* to address reviewers' feedback), and the PR will update accordingly. Reviewers

then examine the new version, and this cycle continues until the PR is either merged or closed. In practice, the base branch also evolves in the meantime, and it is well-recognized that long-lived code on isolated branches can be problematic when merging [6–8, 13, 40]. Therefore, the author must periodically incorporate those upstream changes, typically by rebasing the PR onto the latest HEAD commit of the base branch. Since this action alters the commit history on the PR branch, developers must force-push such updates to take responsibility for the risks.

Figure 1a shows how force-push events (①, ②) are recorded in the PR activity log on GitHub. The two commits form a pair, which we denote as the source and target commits, *i.e.,* the HEAD commit of the PR before and after the force-push event, respectively.

Figure 1b shows the commit graph associated with the PR, with two force-push events (①, ②), which we refer to as SOURCE–TARGET and SOURCE′–TARGET′, respectively. SOURCE and TARGET are the only commits on the PR before and after the force-push event. Their ancestor on the base branch is commit d28e212, which was the HEAD of the base branch when the fork was created.

If there are multiple commits on the PR before the force-push event, the log message only refers to the HEAD commits. In Figure 1b, the SOURCE′ and TARGET′ point to the HEAD of commits $1' - 10'$ and $1'' - 10''$, respectively. The two sets of ten commits both form a linear parent-child commit chain, where their ancestors on the base branch are commits d28e212 and da91e49, respectively.

After the force-push event, the previously recorded commits become detached. For example, Figure 1b shows the current commits on the PR are the commits with TARGET′ as their HEAD.

## 2.3 Impact on Practitioners

While the cognitive load of reviewing such conflated PR revisions remains difficult to quantify, developer complaints about conflated diffs caused by force-pushing date back almost a decade on StackOverflow. According to GitHub Community discussion, reviewers cannot directly isolate and inspect the changes to the PR that were responses to their code review comments. As a result, existing tools show code reviewers as many as thousands of modified lines that do not correspond to meaningful edits within the context of the PR. Other social coding platforms that support pull-based development also suffer from this limitation (*e.g.,* GitLab and Codeberg). For example, on HackerNews, nine developers complain that they must *"review the whole branch all over again"* because there is *"no diff between two versions of the branch"*.

This conflated diff is also a challenge for PR authors and CI maintainers. According to GitHub Community discussion, a failing test case may be correctly fixed by a force-pushed revision to a PR, yet the code snippet that resolves the failure can be buried within a conflated diff. As a result, PR authors and CI maintainers may find it difficult to identify the true cause of the change to build status, limiting their ability to learn from past mistakes and/or improve the robustness of their pipelines.

***Limitation of Existing Solution.*** A potential solution is git *range-diff*—a Git subcommand that relies on heuristic, non-deterministic commit-matching. This solution is not yet integrated into any social coding platform. Moreover, it produces a "diff of diffs", which introduces challenges when interpreting its output.

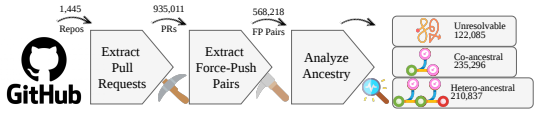



**Figure 2: Data collection and analysis overview.**

## 2.4 Impact on Researchers

Force-push events also challenge researchers. Rapaport *et al.* [31] report that altered histories are prevalent in version control repositories, with pull requests being the most frequently affected, challenging assumptions that repository-based analyses rely on authentic, unaltered histories. Indeed, Silva *et al.* [35] were forced to discard part of their dataset for fault localization tasks because force-pushes masked commits that failed earlier in the lifetime of a PR, obscuring the true fault-inducing changes.

More broadly, an inaccurate history of PR revisions may introduce contamination risks into datasets. TravisTorrent [4], a large-scale CI log dataset, notes that identifying a build-breaking commit is *"undecidable"* when a PR is rebased onto the HEAD of the base branch. Following Beller *et al.*'s [4] procedure, Tomassi *et al.* [39] and Madeiral *et al.* [26] mine Travis CI to build BugSwarm and BEARS by identifying consecutive fail-fix build pairs, using the diff between commits as an estimate of the fix. Such fail-fix pairs can suffer from diff conflation, which may produce misleading results.

To mitigate these risks, researchers may exclude altered commit histories from their analyses. For example, PR-based benchmarks [30, 42] (*e.g.,* SWE-bench [17] and DevOps-Gym [38]) focus on the final state of merged PRs rather than intermediate revisions.

Fields like fault localization and program repair rely on accurately identified fix-inducing and fixing commits [2, 9, 12, 18, 21, 34, 41], often extracting such pairs from linearized main-branch commits. Chen *et al.* [10] emphasize this linearized history as essential for mitigating noise, since the code has already undergone review and refinement. While this suits their design, as a community, an exclusive focus on the main branch overlooks the rich context of discussions and revisions inherent to pull-based development. Capturing this process could yield more fine-grained, representative data for the broader community.

# 3 How Prevalent Are Force-Push Pairs on GitHub?

In this section, we present our approach to studying the prevalence of force-push pairs on GitHub (Section 3.1), followed by the results that we observe (Section 3.2).

## 3.1 Approach

Figure 2 describes our process for querying GitHub, collecting PRs, and extracting force-push pairs. At a high level, our approach consists of: (1) selecting the studied projects, (2) collecting PRs, and (3) identifying force-push pairs. Below, we describe each step.

***Selecting the Studied Projects.*** We leverage the search engine of GitHub to identify the studied projects. We construct a query to search for GitHub repositories, filtering out irrelevant repositories from the entire GitHub corpus, which contains more than 420

**Table 1: Number of repositories and pull requests with and without force-push pairs.**

| | Repositories | PRs | Revised PRs | FPs |
|---|---|---|---|---|
| With FP | 1,366 | 242,624 | 242,624 | – |
| Without FP | 79 | 692,387 | 215,003 | – |
| Total | 1,445 | 935,011 | 457,627 | 568,218 |

million repositories. To exclude small projects, we remove repositories: (i) with fewer than 1,000 forks, which reduces all projects on GitHub to a sample on the order of tens of thousands, and (ii) that are smaller than 10,000 KB, which further reduces the sample to the order of thousands of repositories. To ensure that our analysis reflects actively developed projects, we exclude repositories that have been marked as archived or mirrored, which reduces the sample by roughly one thousand repositories. To filter out inactive projects, we select repositories with at least one recently merged PR. Since weekend activity varies substantially across repositories, we select repositories that have at least one commit landing on the main branch on at least one of the weekdays of the week when our data collection process was initiated. These conditions are expressed using one query and reduce all repositories reachable on GitHub to a sample of 1,673 repositories. Finally, to focus on software development, we select repositories whose primary programming language ranks among the ten most popular languages on GitHub. GitHub detects the programming languages in a repository using the Linguist tool, which computes the proportion of bytes attributed to each recognized language. The primary programming language is the language that accounts for the largest proportion of recognized source-code bytes.

After filtering, we obtain a dataset of 1,445 repositories. The GitHub query and its results are available in our replication package.

***Collecting PRs.*** For each repository, we use the GitHub REST API to search the list of PR numbers submitted within a six-month window, between February and July 2025. For each PR of each sampled repository, we collect its corresponding raw `HTML` page. We collect raw `HTML` because it contains contextual data that we may require, and that is not present in the REST API response.

Overall, due to systematic pauses introduced to prevent abusive web scraping behaviour, the data collection process took 24 days. The dataset of 935,011 PRs consumes 110 GB of storage space.

***Identifying Force-Push Pairs.*** For each PR, we use BeautifulSoup to process the structured `HTML` pages. We first identify force-push events by matching the keyword "force-pushed" in the `HTML` content. We then extract the associated commit identifiers from the embedded hyperlinks using regular expressions [1]. Finally, following the convention illustrated in Figure 1a, we label commits according to their roles, where `SOURCE` and `TARGET` denote the commits before and after the force-push, respectively.

## 3.2 Results

Table 1 provides an overview of our studied repositories and PRs with and without force-push pairs, along with the total number of force-push pairs and PRs that are revised.

***Observation 1: Force-push activity is common in pull-based development.*** Table 1 shows the number of repositories and PRs with at least one force-push pair. Among 935,011 PRs, 457,627 (48.9%) undergo at least one revision, and 242,624 (26.0%) are revised using force-push. Overall, we identify 568,218 force-push pairs. Among 1,445 repositories, 1,366 (94.5%) contain at least one force-push pair. These repositories account for 932,212 PRs (99.7%), suggesting that force-push events are a common occurrence.

**Conclusion.** Force-push events are not isolated or non-standard coding practices, as they occur in 94.5% of our studied projects.

# 4 What Characterizes Conflation-Prone Force-Push Pairs?

Not all force-push pairs are equally prone to diff conflation. Figure 1b shows that comparing commit pairs that diverge within the PR scope produces PR-specific commits (*e.g.,* `SOURCE–TARGET`). In contrast, commit pairs that diverge on the base branch can yield conflated diffs (*e.g.,* `SOURCE′–TARGET′`), since their ancestors on the base branch (*e.g.,* branch ancestors `d28e212` and `da91e49`) are separated by unrelated updates (*e.g.,* commit `74d5d93`).

In this section, we present our approach to classify force-push pairs based on their ancestral relationships (Section 4.1), and our observations about their prevalence and characteristics (Section 4.2).

## 4.1 Approach

Based on the positions of their respective ancestors on the base branch, we classify force-push pairs as ***co-ancestral*** pairs if they are not prone to conflation, or as ***hetero-ancestral*** pairs if they are prone to conflation. Algorithm 1 presents the procedure for identifying the ancestral relationships of force-push pairs. Figure 3 illustrates this procedure using two examples, the co-ancestral `SOURCE–TARGET` and the hetero-ancestral `SOURCE′–TARGET′`.

The `classifyRelationships` function takes force-push pairs and their base branch (Line 18) as input. If either the source or target commit, or their containing branch, are unavailable on the remote (Line 10), we return `NULL` for branch ancestors (Line 11). For each source and target commit, we first compute their common ancestor using `git merge-base` (Line 12). If no common ancestor can be computed (Line 13), we also return `NULL` for branch ancestors (Line 14). Figure 3 shows that the common ancestor for both `SOURCE–TARGET` and `SOURCE′–TARGET′` is commit `d28e212`.

Next, we identify the closest ancestor on the base branch, referred to as the *branch ancestor*. In cases where a PR is merged, commits from the PR may also appear on the base branch. Therefore, we define the branch ancestor as the first ancestor that lies on the base branch (Line 5) and is not part of the commit history of the PR (Line 6). The `computeBA` function (Line 1) locates the branch ancestor of a given reference with respect to the base branch, which in this case coincides with the common ancestor in Figure 3. First, we enumerate the ancestor commits in chronological order from the common ancestor to the reference using `git rev-list` (Line 3). Figure 3 illustrates the ancestor lists: [1] for `SOURCE`, [1′] for `TARGET`, [1′, …, 10′] for `SOURCE′`, and [1″, …, 10″] for `TARGET′`. Our goal is to determine whether they share the same branch ancestor by first examining the position of the branch ancestor relative to the common ancestor. If the branch ancestor appears within the ancestor lists, we then check if the identified branch ancestors of the source and target commits are the same.

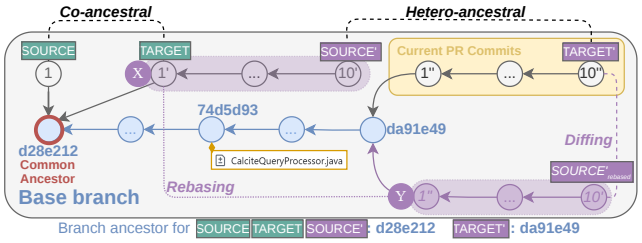



**Figure 3: Identifying the ancestral relationship of force-push pairs and rebasing the SOURCE′ to the branch ancestor of TARGET′. Arrows indicate parent-child commit relations.**

If the branch ancestor is not in the list of candidates (Line 8), it implies that the common ancestor is within the PR scope. Thus, the algorithm returns the last commit in the ancestor list, which corresponds to the common ancestor. This indicates that both the source and target commits are children of the same commit on the base branch and therefore share the same branch ancestor.

After determining the branch ancestors of the source and target commits (Line 21), we proceed to compare them. If the branch ancestors are `NULL`, the pair is classified as *unresolvable* (Lines 22–23). If the pair shares the same branch ancestor, it is classified as *co-ancestral*; otherwise, as *hetero-ancestral* (Lines 24–27). In Figure 3, `SOURCE` and `TARGET` share the same branch ancestor, and the pair is classified as *co-ancestral*. In contrast, `SOURCE′` and `TARGET′` have different branch ancestors, and are classified as *hetero-ancestral*.

Finally, the function returns three sets, one with pairs that are unresolvable, one with co-ancestral pairs, and one with the hetero-ancestral pairs (Line 28).

## 4.2 Results

Figure 4 provides an overview of the prevalence of ancestral relationships among force-push pairs.

***Observation 2: Force-push pairs with a hetero-ancestral relationship, which requires additional treatment to avoid conflation, occur at rates comparable to those with a co-ancestral relationship.*** Figure 4 shows that, among 568,218 force-push pairs, 210,837 (37.1%) exhibit a hetero-ancestral relationship that is prone to diff conflation, whereas 235,296 (41.4%) exhibit a co-ancestral relationship that can be directly compared. The cumulative distribution functions further indicate that the overall distributions of co- (dashed) and hetero-ancestral (dash-dotted) pairs are similar, suggesting that developers frequently rewrite revision histories in ways that result in conflation of changes.

Finally, 122,085 pairs (21.5%) yield unresolvable relationships, 122,015 of which (99.94%) are due to deleted commits or branches; these cases are artifacts of data availability in our analysis rather than limitations of our proposed approach (Section 5). If force-push pairs are analyzed at their creation time rather than in the batch process that we conduct in this section, the missing remotes and branches would still exist, and these pairs can be analyzed. A negligible fraction, 70 of 568,218 pairs (0.01%), do not share a common ancestor (*e.g.,* due to merges involving commits from external repositories). Although this case may be viewed as a third

**Algorithm 1:** Classify Ancestral Relationships

```
1  function computeBA(common, ref, branch):
2      // locate branch ancestor
3      ancestors ← git.rev-list(common, ref, branch)
4      foreach commit ∈ ancestors do
5          if commit ∈ getBaseBranchCommits(ref, branch)
6          and commit ∉ getPRCommits(ref) then
7              return commit
8      return last(ancestors)
9  function getBothBAs(source, target, branch):
10     if either source, target, branch not on remote then
11         return NULL, NULL
12     common ← git.mergebase(source, target)
13     if no common then
14         return NULL, NULL
15     sourceBA ← computeBA(common, source, branch)
16     targetBA ← computeBA(common, target, branch)
17     return sourceBA, targetBA
18 function classifyRelationships(forcePushes):
19     unresolvable, coAncestral, heteroAncestral ←{ }, { }, { }
20     foreach (source, target, branch) ∈ forcePushes do
21         sourceBA, targetBA ← getBothBAs(source, target, branch)
22         if sourceBA == NULL or targetBA == NULL then
23             unresolvable ← unresolvable ∪ {(source, target)}
24         else if sourceBA == targetBA then
25             coAncestral ← coAncestral ∪ {(source, target)}
26         else
27             heteroAncestral ← heteroAncestral ∪ {(source, target)}
28     return unresolvable, coAncestral, heteroAncestral
```

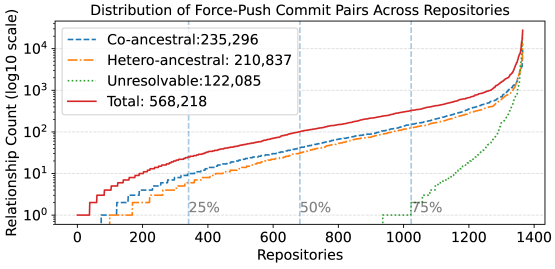



**Figure 4: Sorted distribution of ancestral relationships.**

relationship alongside co- and hetero-ancestral, its extremely low prevalence leads us to exclude it from further analysis in this paper.

**Conclusion.** Hetero-ancestral force-push pairs occur at rates comparable to co-ancestral ones. This is likely due to the good habit of keeping the PR branch up-to-date with the base branch.

# 5 To What Extent Can We Deconflate Force-Pushed Revisions?

In this section, we investigate the severity of the conflation problem inherent to the hetero-ancestral force-push pairs, present our approach to mitigating conflation (Section 5.1), and evaluate the effectiveness of that approach (Section 5.2).

## 5.1 Approach

Figure 1 shows that the co-ancestral `SOURCE–TARGET` pair differs within the PR scope and thus do not require additional treatment to compare. In contrast, the hetero-ancestral `SOURCE′–TARGET′` pair

**Table 2: Comparison of `git diff` statistics before and after applying *AntiDiffamine* on force-push pairs.**

| | | | Default Rebase (76.2%) | Strategic Rebase (21.9%) | Manual Rebase (1.9%) |
|---|---|---|---|---|---|
| Rate (%) | | 25th | 59 | 10 | 0 |
| | | 50th | 74 | 23 | 0 |
| | | 75th | 88 | 36 | 3 |
| Changed Files | Original | 25th | 6 | 18 | 34 |
| | | 50th | 31 | 96 | 191 |
| | | 75th | 152 | 394 | 839 |
| | Rebased | 25th | 0 | 1 | – |
| | | 50th | 0 | 1 | – |
| | | 75th | 1 | 3 | – |
| Added Lines | Original | 25th | 64 | 299 | 519 |
| | | 50th | 575 | 2,283 | 4,652 |
| | | 75th | 3,785 | 12,099 | 25,974 |
| | Rebased | 25th | 0 | 1 | – |
| | | 50th | 0 | 5 | – |
| | | 75th | 4 | 30 | – |
| Removed Lines | Original | 25th | 25 | 141 | 282 |
| | | 50th | 242 | 1,018 | 2,916 |
| | | 75th | 1,763 | 5,854 | 16,672 |
| | Rebased | 25th | 0 | 1 | – |
| | | 50th | 0 | 3 | – |
| | | 75th | 3 | 27 | – |
| Merge Conflicts | Files | 25th | – | 1 | 1 |
| | | 50th | – | 1 | 2 |
| | | 75th | – | 2 | 4 |
| | Lines | 25th | – | 1 | 0* |
| | | 50th | – | 1 | 1 |
| | | 75th | – | 3 | 4 |

*Merge conflicts involving structural changes, file deletions, or non-textual files (*e.g.*, *.png*, *.zip*, *.tar*) are not parsed.

**Algorithm 2:** Rebasing and Diff Force-Push Pairs

```
1  function AntiDiffamine(forcePushes):
2      foreach (source, target, branch) ∈ forcePushes do
3          originalDiff ← git.diff(source, target)
4          sourceBA, targetBA ← getBothBAs(source, target, branch)
5          git.checkout(source)
6          mergeConflicts, source_rebased ← git.rebase(--onto, targetBA,
               sourceBA, source) // default rebase
7          if mergeConflicts is not empty then
8              extractAndStore(mergeConflicts)
9              mergeConflicts, source_rebased ← git.rebase(--onto, targetBA,
                   sourceBA, source, option=target) // strategic rebase
10             if mergeConflicts is not empty then
11                 rebasedDiff ← NULL // require manual rebase
12         rebasedDiff ← git.diff(source_rebased, target)
13         return originalDiff, rebasedDiff
```

differs by numerous commits on the base branch, which will include non-PR-specific changes when applying *git diff*.

**AntiDiffamine.** To mitigate this conflation caused by hetero-ancestral relationships, we propose *AntiDiffamine*, which (1) identifies a suitable common branch ancestor as the comparison base, (2) rebases the commits onto this ancestor, (3) isolates changes specific to the two commits, and (4) produces a more meaningful diff for the hetero-ancestral pairs than a direct comparison.

Figure 3 illustrates a scenario of rebasing the SOURCE′–TARGET′ pair using *AntiDiffamine*, which copies the highlighted commits ⓧ from SOURCE′ until the child of the branch ancestor of SOURCE′, and reapplies them at the branch ancestor of TARGET′. This action produces the highlighted commits ⓨ.

Algorithm 2 implements the process. For each source and target commit and their base branch in a force-push pair (Line 2), we first compute the diff between the original source-target pair (Line 3, *e.g.*, SOURCE′–TARGET′ in Figure 3) and their branch ancestors (Line 4). Then, the source commit (*e.g.*, SOURCE′ in Figure 3) is checked out (Line 5). A default rebase operation is attempted (Line 6), which fails if merge conflicts occur (Line 7). In such cases, the rebase process is paused to extract and record the conflicting lines (Line 8). To resolve the merge conflicts during rebasing, Git provides strategy options. When merge conflicts occur, a strategic rebase is attempted, which prioritizes the changes in the target commit, *i.e.*, the PR-updated version of the codebase (Line 9). If the attempted strategic rebase also fails, then we skip the force-push pair because it requires manual intervention (Lines 10–11).

If either the default or strategic rebase succeeds, the HEAD points to the rebased version of source, which has a new SHA identifier (*e.g.*, SOURCE′ to $\text{SOURCE}'_{rebased}$ in Figure 3). We then diff the new pair (Line 12, *e.g.*, $\text{SOURCE}'_{rebased}$–TARGET′ in Figure 3).

***Comparing with Range-Diff.*** *Range-diff* is a built-in Git function that compares two ranges of commits using heuristic commit matching. Although *range-diff* takes similar inputs as *AntiDiffamine* (*i.e.*, two sequences of commits), the two approaches are designed for different purposes. *Range-diff* determines the extent to which two commit ranges diverge, whereas *AntiDiffamine* provides an overview of the changes between two commit ranges. Therefore, rather than treating *range-diff* as a baseline, we characterize its differences with *AntiDiffamine* quantitatively.

We make three adaptations to compare the output of *range-diff* with *AntiDiffamine*. First, since *range-diff* focuses on how each commit diverges, it matches commits individually and displays the line-level differences only when the differences fall within its fuzzy matching criteria. Otherwise, only the commit hashes of the divergent commits are displayed, requiring users to look up their diffs. Thus, we append the content of the divergent commits to the range-diff output for a fair comparison with *AntiDiffamine*.

Second, *range-diff* is not intended to be machine-readable according to its official documentation. In addition to the standard prefixes of the unified diff format (+ and -), *range-diff* introduces composite prefixes: -- (removal of deletion), -+ (removal of addition), ++ (target-side-only addition), and +- (target-side-only deletion). Since *AntiDiffamine* focuses on the diff of the source-to-target revision, we map only the prefixes to changes comparing the source and target revisions: ++ and -- as addition and +- and -+ as deletion, treating - and + as unchanged context lines.

Finally, *range-diff* takes in a creation factor that ranges from 0 to 100, which controls the strictness of its fuzzy matching. The larger the factor, the less likely that commits will match. We explore the impact of the creation factor by evaluating eleven settings (*i.e.*, 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100).

## 5.2 Results

We apply *AntiDiffamine* to all hetero-ancestral force-push pairs identified in Section 4 and observe a failure rate of 0.2% (448 pairs from 99 repositories) due to commits on the remote branch being

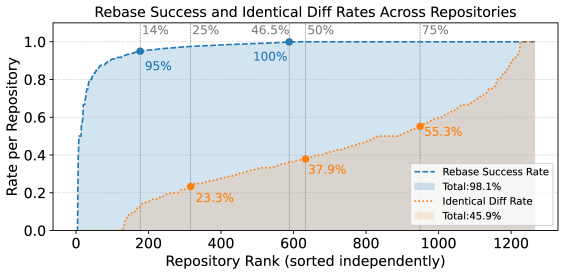



**Figure 5: Per-repo rebase success and identical diff rates.**

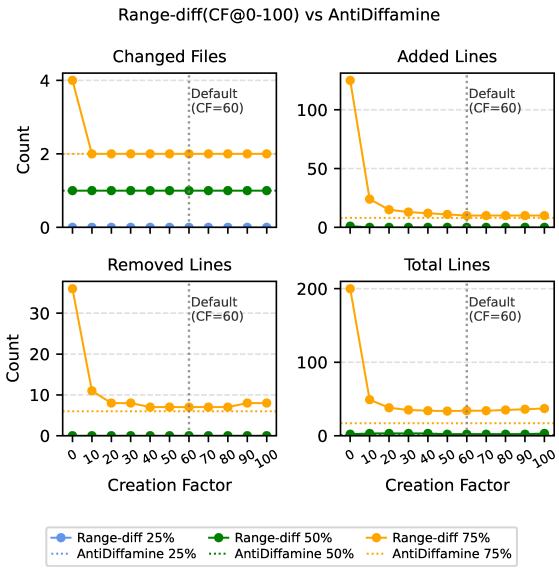



**Figure 6: Comparing *range-diff* to *AntiDiffamine*.**

deleted. Similar to Section 4.2, these failures are an artifact of our evaluation timing rather than a flaw in our approach.

Table 2 shows: (1) the 25th, 50th, 75th percentiles, and the total counts of the per-repository rebase success rate, (2) the original and rebased diff sizes, and (3) the number of files and lines involved in merge conflicts. Figure 5 shows the distributions of: (1) force-push pairs that are automatically rebased and analyzed per-repository, and (2) force-push pairs with identical commits per-repository. Figure 6 shows the number of changed files, added lines, removed lines, and total lines produced by *range-diff* and *AntiDiffamine* at the 25th, 50th, and 75th percentiles.

***Observation 3: Diffs of hetero-ancestral force-push pairs are heavily conflated, while AntiDiffamine reduces them by at least an order of magnitude after rebasing.*** Prior studies show review diffs are typically small [3, 14, 32, 33], with median (mean) sizes under 100 lines and few exceeding several hundred. The Original columns in Table 2 show that the untreated hetero-ancestral diffs are orders of magnitude larger.

The fourth column in Table 2 shows that, for default-rebased pairs, the 50th percentile numbers of changed files, added lines, and removed lines decrease from 31, 575, and 242 to zero, with no merge conflicts. For strategically-rebased pairs (the fifth column), the 50th percentile numbers of changed files, added lines, and removed lines decrease from 96, 2,283, and 1,018 to one, five, and three, respectively, with merge conflict files and lines of one. Especially for force-push pairs that require manual rebasing (the sixth column), the original diffs at the 50th percentile numbers of changed files, added lines, and removed lines are 191, 4,652, and 2,916, respectively.

***Observation 4: AntiDiffamine can automatically deconflate almost all hetero-ancestral force-push pairs.*** Indeed, 98.1% of force-push pairs can be rebased automatically using either the default (76.2%) or the fallback strategic (21.9%) approach. The dashed line in Figure 5 shows that at least 95% of force-push pairs can be automatically rebased and analyzed in 86% of repositories. Moreover, in 53.5% (100%–46.5%) of repositories, all force-push pairs can be automatically rebased and analyzed. These results suggest that *AntiDiffamine* is applicable to the majority of force-push pairs and repositories in the studied sample.

The 1.9% of force-push pairs that require manual intervention during the rebase action have merge conflicts that involve few files and lines. Indeed, the sixth column of Table 2 shows 25th (50th) percentile merge-conflict files at one (two), and merge-conflict lines at zero (one), respectively. Since our parser detects merge-conflict markers in textual files, merge conflicts involving file structure, removed files, or binary files (*e.g.,* `.png`) are excluded from the conflict-line count.

***Observation 5: AntiDiffamine reveals a substantial portion of hetero-ancestral force-push pairs with identical commits.*** The dotted line in Figure 5 shows that when applying *AntiDiffamine*, 45.9% of the studied hetero-ancestral pairs (96,848) yield no changes within the PR scope, suggesting their original changes are merely synchronizations with the most recent `HEAD` of the base branch. This phenomenon can be observed across 90.5% of the studied repositories. Within each repository, the 25th, 50th, and 75th percentile values of identical diffs are 23.3%, 37.9%, and 55.3%, respectively. These results suggest that large diffs may create the perception of an onerous review, whereas they often reflect simple branch synchronization, and should be explicitly labelled to avoid unnecessary review effort.

***Observation 6: Range-diff outputs are larger in size than AntiDiffamine.*** Figure 6 shows that *range-diff* outputs vary with the setting of the creation factor. The number of lines of output of *range-diff* spans from two to three lines at the 50th percentile and 37 and 200 at the 75th percentile, respectively. In contrast, the number of lines of output of *AntiDiffamine* is one line at the 50th percentile and 17 lines at the 75th percentile.

In terms of the number of changed files, added lines, and removed lines, *range-diff*'s output size similarly varies with the creation factor, spanning from two to four changed files, ten to 125 added lines, and eight to 36 removed lines; whereas *AntiDiffamine* produces eight added lines and six removed lines at the 75th percentile. At the median, *range-diff* performs comparably to *AntiDiffamine*. Both *range-diff* and *AntiDiffamine* detect no changed files, added lines, and removed lines at the 25th percentile, confirming Observation 5, *i.e.,* that almost half (45.9%) of the hetero-ancestral force-push pairs are branch synchronization.

**Table 3: Comparison of `git diff` statistics before and after applying *AntiDiffamine* on Kubernetes build fail-fix pairs.**

| | | | Default Rebase (63.8%) | Strategic Rebase (30%) | Manual Rebase (6.2%) |
|---|---|---|---|---|---|
| Changed Files | Original | 25th | 51 | 194 | 160 |
| | | 50th | 210 | 559 | 500 |
| | | 75th | 994 | 1,093 | 1,162 |
| | Rebased | 25th | 1 | 1 | – |
| | | 50th | 2 | 2 | – |
| | | 75th | 3 | 8 | – |
| Added Lines | Original | 25th | 160 | 6,694 | 16,604 |
| | | 50th | 6,103 | 23,079 | 51,932 |
| | | 75th | 50,540 | 53,642 | 82,941 |
| | Rebased | 25th | 1 | 3 | – |
| | | 50th | 6 | 23 | – |
| | | 75th | 30 | 124 | – |
| Removed Lines | Original | 25th | 475 | 2,409 | 1,580 |
| | | 50th | 1,928 | 10,783 | 11,407 |
| | | 75th | 21,055 | 32,496 | 40,981 |
| | Rebased | 25th | 0 | 3 | – |
| | | 50th | 4 | 26 | – |
| | | 75th | 35 | 104 | – |
| Merge Conflicts | Files | 25th | – | 1 | 1 |
| | | 50th | – | 1 | 2 |
| | | 75th | – | 2 | 3 |
| | Lines | 25th | – | 1 | 0* |
| | | 50th | – | 2 | 1 |
| | | 75th | – | 4 | 18 |

*Merge conflicts involving structural changes, file deletions, or non-textual files (*e.g.*, *.png*, *.zip*, *.tar*) are not parsed.

To provide additional context for practitioners, our *AntiDiffamine* GitHub Action also computes *range-diff* output at the default creation factor of 60 alongside the *AntiDiffamine* output, rendering both in a common interface.

**Conclusion.** Force-pushed revisions are substantially conflated. *AntiDiffamine* reduces the size of diff to better align with the needs of developers. *AntiDiffamine* also makes diffs that correspond to branch synchronization trivial to filter out, thereby reducing unnecessary review effort.

## 6 To What Extent Does Conflation Impact Downstream Analyses?

In Section 1, the conflated fail-fix pair in BugSwarm [39] illustrates that diff conflation can interfere with downstream analyses. To systematically assess this impact, we study fail-fix pairs of build executions that rely on clean PR revision data.

We study fail-fix pairs in the context of Kubernetes—one of the largest and most visible projects in our dataset. More specifically, we investigate the extent to which co- and hetero-ancestral relationships occur when analyzing this other pull-based development activity. When hetero-ancestral relationships emerge, we evaluate whether *AntiDiffamine* can deconflate the diffs as well.

Below, we describe the context (Section 6.1), the approach (Section 6.2) and the results (Section 6.3) of our Kubernetes case study.

### 6.1 Context

A fail-fix pair can be identified by examining build execution history, where a failing build is followed by a passing build for the same PR.

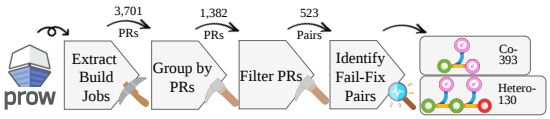



**Figure 7: Overview of the case study.**

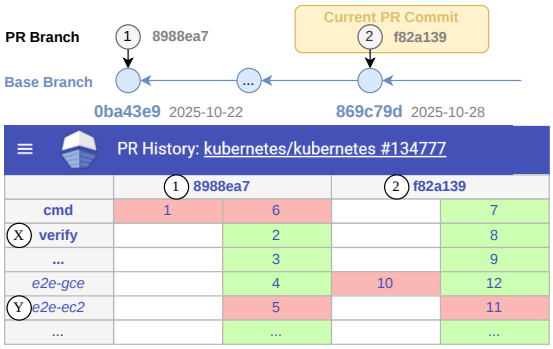



**Figure 8: Prow build dashboard for Kubernetes-PR 134777**

Fail-fix pairs are frequently mined to construct datasets for fault localization and program repair tasks [18, 35, 39].

In a linear parent-child sequence of commits with associated builds, a fail-fix transition is between a parent commit with a failing build and a child commit with a passing build. When a force-push event occurs, the linear sequence may not be preserved. In such cases, a fail-fix pair may coincide with a force-push pair, since PR updates are commonly submitted via force-pushes (Section 2.2).

**Key Insight 1.** Therefore, fail-fix pairs that are mined from build execution history are prone to the same diff conflation issues caused by force-push pairs.

To study the impact of force-push-induced diff conflation on fail-fix pairs, we extract build execution status and job information. Doing so requires understanding the build context of each studied repository rather than simply issuing API calls. Since projects may execute their builds using different CI providers and a project may even use multiple providers simultaneously, processing the logs of all 1,366 studied projects presents practical challenges. Solving those challenges would likely introduce construct and internal threats to validity by producing a build outcome signal that would be unreliable. Hence, we perform a case study to allow us to dedicate resources to producing a reliable build outcome signal.

We select Kubernetes—a popular container orchestration system—as our case study subject. The Kubernetes team follows a pull-based development model and uses Prow as its CI platform.

Figure 8 illustrates an example of the Prow build dashboard, showing PR 134777 from the Kubernetes project. Each build consists of multiple jobs, which may be *required*, *i.e.*, it must pass for a PR to be permitted to merge (*e.g.*, Ⓧverify), or *optional*, *i.e.*, its outcome does not affect the merge permission for a PR (*e.g.*, Ⓨe2e-ec2). We focus on the build outcomes of the required jobs, since they impact merge decisions.

**Table 4: Top 10 files in the *git-diff* of build fail-fix pairs under three treatments on the hetero-ancestral pairs.**

| Untreated | Count | Excluded | Count | Treated with AntiDiffamine | Count |
|---|---|---|---|---|---|
| **pkg/.../zz_generated.openapi.go** | 102 | *test/.../versioned_feature_list.yaml* | 17 | *test/.../versioned_feature_list.yaml* | 28 |
| **api/openapi-spec/swagger.json** | 100 | **api/openapi-spec/swagger.json** | 15 | **api/openapi-spec/swagger.json** | 26 |
| **pkg/features/kube_features.go** | 98 | **pkg/.../zz_generated.openapi.go** | 11 | **pkg/kubelet/kubelet.go** | 24 |
| **pkg/kubelet/kubelet.go** | 96 | **pkg/features/kube_features.go** | 11 | **pkg/.../zz_generated.openapi.go** | 22 |
| go.sum | 85 | **pkg/kubelet/kubelet.go** | 11 | *staging/.../cacher_whitebox_test.go* | 21 |
| pkg/kubelet/kubelet_test.go | 85 | *hack/golangci.yaml* | 10 | **pkg/features/kube_features.go** | 20 |
| go.mod | 84 | hack/golangci-hints.yaml | 9 | staging/.../allocator.go | 19 |
| staging/.../go.mod | 84 | *staging/.../cacher_whitebox_test.go* | 9 | *pkg/kubelet/kubelet_test.go* | 17 |
| vendor/modules.txt | 84 | test/.../test_data/versioned_feature_list.yaml | 8 | staging/.../allocator_test.go | 17 |
| test/e2e/feature/feature.go | 82 | *test/e2e/feature/feature.go* | 7 | *hack/golangci.yaml* | 15 |

**Table 5: Precision and recall of the three treatments**

| | Mean | | | Median | | |
|---|---|---|---|---|---|---|
| Treatment | Precision | Recall | F1 | Precision | Recall | F1 |
| Untreated | .02 | .60 | .04 | 0 | 1 | 0 |
| Excluded | .29 | .29 | .29 | 0 | 0 | 0 |
| AntiDiffamine | .54 | .73 | .62 | .50 | 1 | .67 |

## 6.2 Approach

***Identifying Fail-Fix Pairs.*** A *failing build* is a build where at least one required job fails, whereas a *passing build* is one where all required jobs succeed. Figure 8 shows two build attempts (columns `8988ea7` and `f82a139`), each being composed of jobs (rows). The required job *cmd* fails on ①commit `8988ea7`, while the subsequently force-pushed ②commit `f82a139` resolves the failure. Due to build flakiness, the job initially fails on commit `f82a139` (10), but passes when re-executed (12). To account for this, we consider that a job has passed on a commit if its latest invocation succeeds. Similar to Beller *et al.* [4], we identify fail-fix commit pairs as two consecutive builds where the earlier build fails and the subsequent build passes.

Figure 7 shows an overview of our data collection and analysis pipeline. We first group the Prow build jobs by PR, and filter out the PRs that invoke partial builds that do not execute all required jobs. Next, we identify fail-fix pairs from the Prow dashboard and map each pair to its corresponding commits. Following the same approach described in Sections 4 and 5, we identify the ancestral relationships of the commits that correspond to fail-fix pairs and apply AntiDiffamine to reduce the diff of hetero-ancestral pairs.

***Identifying the Most Frequently Implicated Files in Build-Fixing Diffs.*** We compute the frequency with which files appear in diffs of hetero-ancestral pairs, which are (1) not treated (*i.e.,* retain all pairs and directly compare them), (2) excluded (*i.e.,* retain only the co-ancestral pairs and directly compare them), and (3) treated (*i.e.,* retain all pairs and treat the hetero-ancestral ones with AntiDiffamine). We rank and report the top ten files most frequently implicated in the three lists (Table 4).

***Labelling Actual Build-Fixing Files.*** To assess which treatment generates diffs that are closest to the changes within the PR scope that actually fix the build failure, two annotators independently label the files implicated in the fixing commits for the hetero-ancestral build fail-fix pairs in Kubernetes. A build failure is fixed by changes within the PR scope (*i.e.,* a failure triggered by PR commits) or by synchronizing to the base branch (*i.e.,* a failure triggered by not updating to the base branch). In the context of AntiDiffamine, we focus our attention on changes within the PR scope because AntiDiffamine focuses on PR evolution.

To identify fix locations, annotators are provided with the diffs between the source and target commits and their branch ancestor (*i.e.,* the source diff and target diff), along with the logs of the failing jobs as an additional reference. To account for subjectivity, we estimate inter-rater reliability using Krippendorff's alpha [20, 27], where the population for each pair is the union of the sets of files in the source and target diffs. For each case where annotators disagree, we conduct a follow-up discussion. If an agreement cannot be reached, a third annotator casts the deciding vote.

To assess which treatment performs best in locating fix-inducing files on hetero-ancestral pairs (*i.e.,* untreated, excluded, and treated with AntiDiffamine), we calculate precision, recall, and F1-score with respect to the ground truth. To further evaluate the downstream impact, we report and compare the rankings of the ten most frequently implicated files across hetero-ancestral fail-fix pairs under different treatments with the ground truth.

## 6.3 Results

We could not label six of the 130 fail-fix pairs. Two pairs lack failure logs, since Kubernetes periodically purges CI logs. The four other pairs lack PR revisions, since the author abandoned the PR by rebasing the branch onto the base `HEAD`, leaving zero commits. A build is still triggered by the rebase action, but the PR has no changes to label. An example PR and commit graph are included in our replication package.

For the remaining 124 hetero-ancestral fail-fix pairs, two annotators label the fix locations at the file level, achieving a Krippendorff's alpha of 0.807, which indicates a high level of agreement [20, 27].

Table 3 shows the sizes of diffs of fail-fix pairs with hetero-ancestral relationships before and after applying AntiDiffamine. Table 4 shows the ten most frequently implicated files in the original and rebased diffs. Table 5 shows the precision, recall, and F1-score for each treatment using the human-labelled fix locations as the ground truth. Table 6 presents the ten most frequently implicated files in human-labelled fix locations, as well as when hetero-ancestral pairs are not treated, or are treated with AntiDiffamine.

***Observation 7: hetero-ancestral relationships occur at a considerable rate in build fail-fix pairs.*** There are 3,701 Kubernetes PRs between February and October in 2025. These PRs may only invoke a subset of build jobs for quick testing instead of the full build execution. Out of the 1,382 PRs that contain all required build

**Table 6: Top 10 files in hetero-ancestral pairs, untreated and treated with AntiDiffamine, and the human-labelled fix locations**

| Untreated | Count | Treated with AntiDiffamine | Count | Ground-Truth | Count |
|---|---|---|---|---|---|
| pkg/.../zz_generated.openapi.go | 91 | staging/../allocator.go | 14 | pkg/../versioned_kube_features.go | 3 |
| pkg/features/kube_features.go | 87 | **pkg/kubelet/kubelet.go** | 13 | **test/../versioned_feature_list.yaml** | 3 |
| ../openapispec/swagger.json | 85 | **staging/.../cacher_whitebox_test.go** | 12 | **pkg/kubelet/kubelet_test.go** | 3 |
| **pkg/kubelet/kubelet.go** | 85 | api/openapi-spec/swagger.json | 11 | test/e2e_node/pod_resize_test.go | 2 |
| go.sum | 82 | pkg/../zz_generated.openapi.go | 11 | test/../dra.go | 2 |
| go.mod | 81 | **pkg/kubelet/kubelet_test.go** | 11 | **pkg/kubelet/kubelet.go** | 2 |
| staging/../go.mod | 81 | **test/../versioned_feature_list.yaml** | 11 | **staging/../cacher_whitebox_test.go** | 2 |
| vendor/modules.txt | 81 | ../..v1alpha3_openapi.json | 10 | ../api__v1_openapi.json | 2 |
| **pkg/kubelet/kubelet_test.go** | 79 | ../..v1beta1_openapi.json | 10 | staging/../reflector_test.go | 2 |
| staging/../go.sum | 77 | pkg/../types.go | 10 | staging/../watch_test.go | 2 |

jobs, we identify 523 fail-fix pairs at the PR level. While 393 of the pairs are co-ancestral, 130 are hetero-ancestral, accounting for 24.9% of the studied fail-fix pairs.

***Observation 8: Hetero-ancestral fail-fix pairs yield large diffs, suggesting that they are also prone to conflation.*** In total, 93.8% of fail-fix pairs are automatically rebased. The fourth column of Table 3 shows that, for default-rebased pairs, the corresponding values decrease from 210, 6,103, and 1,928 to two, six, and four. For strategically-rebased pairs (the fifth column), the 50th percentile diff sizes are reduced from 559, 23,079, and 10,783 to two, 23, and 26, respectively, with merge-conflict files and lines of one and two, respectively. Across categories, AntiDiffamine effectively removes changes to the base branch from diffs.

The sixth column of Table 3 shows that 6.2% of hetero-ancestral fail-fix pairs cannot be automatically rebased. They have merge-conflict files and lines of one and zero at the 25th percentile, respectively, and two and one at the 50th percentile, respectively. Even when AntiDiffamine encounters conflicts that require manual intervention, the size of the conflict regions remains manageable. The frequency of large and broad diffs suggests that the base branch activity is also dominating the content of revisions in fail-fix pairs.

> **Key Insight 2.** Downstream analyses on fail-fix pairs, such as fault localization, program repair, or patch characterization, may draw incorrect conclusions of the location and nature of the fix.

The conflated diffs may attribute the fix to files on the base branch rather than within the PR itself.

***Observation 9: Fail-fix pairs are often identical before and after the rebase.*** Similar to Observation 5, we find that the diffs of 11.5% of hetero-ancestral fail-fix pairs contain no changes after factoring out changes on the base branch by rebasing.

> **Key Insight 3.** Fail-fix pairs often correspond to simple synchronization with the base branch, which alone is sufficient to resolve build failures.

This finding aligns with examples in Section 1. If care is not taken, one may incorrectly attribute the conflated diff to the PR author when the build was actually fixed by branch updates outside the scope of the PR. With the AntiDiffamine Action installed, practitioners can easily recognize such branch synchronization activity.

***Observation 10: The most frequently implicated files almost always change ranking across different treatments on hetero-ancestral pairs.*** Table 4 shows that even among overlapping files, implication frequencies differ substantially, ranging from 82–102 down to 7–17. Only one file (a build-generated `JSON`) retains the same ranking across all three treatments. We suspect that these discrepancies would become more pronounced if the analysis were conducted at finer granularities (*e.g.,* line level). When hetero-ancestral pairs are untreated or excluded, only five of the top ten files are source files (*e.g.,* `.go`), compared to seven for AntiDiffamine.

> **Key Insight 4.** Failing to address hetero-ancestral pairs underestimates source-file involvement, biasing downstream analyses.

Researchers unaware of hetero-ancestral relationships may generate data via direct comparisons over all fail-fix pairs [39] (untreated), or recognize the noise and exclude such pairs [4] (excluded). AntiDiffamine instead preserves the full dataset while narrowing diffs to PR-specific changes.

***Observation 11: The edits of hetero-ancestral pairs identified by AntiDiffamine are the closest to the actual fix of the build failure.*** Table 5 shows that AntiDiffamine outperforms both alternatives, achieving mean (median) precision, recall, and F1-scores of 0.54 (0.50), 0.73 (1.0), and 0.62 (0.67), respectively, compared to the untreated and excluded approaches, whose mean (median) precision are 0.02 and 0.29 (0 and 0), recall are 0.60 and 0.29 (1.0 and 0.0), and F1-scores are 0.04 and 0.29 (0 and 0), respectively.

When the ground truth is an empty set, a treatment returning no files has a precision and recall of one, while one returning any file has values of zero. The excluded group always returns no files, receiving precision and recall of either one or zero. AntiDiffamine filters out non-PR-related edits and thus returns no files more often than the untreated group, explaining its higher recall.

Table 6 shows that the excluded treatment returns no files for hetero-ancestral pairs, so it is omitted from the table. The untreated and AntiDiffamine-treated groups overlap with the actual fix locations in two and four files, respectively. By detecting what has changed between two revisions (a superset of the actual fix locations), AntiDiffamine narrows down the set of files of potential fix locations compared to the untreated group.

> **Conclusion.** The diff conflation problem is also present in build fail-fix pairs. AntiDiffamine effectively reduces the diff size and removes changes made outside the PR context.

## 7 Related Work

Gousios *et al.* [14] found that merges and cherry-picking via rebasing often fail to preserve revision history, hindering reviewers' ability to trace earlier revisions. Later, GitHub added a force-push link to the PR timeline, enabling direct inspection of force-push differences, which has allowed researchers to study PR revisions otherwise unretrievable [11, 15].

Despite official support to trace code revisions in PRs, their intrinsic characteristics may negatively impact the code review process. Li *et al.* [23] investigated repeated revisions in PRs by detecting force-push events. They analyzed the reasons behind PR non-acceptance in the context of such repeated revisions and highlighted the need for improved tooling support to facilitate the review of force-pushed updates [22]. They further identified rebasing performed to update PRs as a contributing factor to PR abandonment. Paixao and Maia [29] found that frequent rebasing on Gerrit can be harmful, as some of the source code changes under review may originate from commits on the base branch rather than from modifications within the code review. Such out-of-context changes increase the cognitive load on reviewers. Similarly, Nachuma and Zibran [28] studied agent-authored PRs and identified force-push indicators as a strong factor associated with non-acceptance. They attribute this effect to force-push activities "disrupting the shared context". In contrast, we find that force-push is an inherent and often necessary practice in PR workflows, particularly for synchronizing with the base branch.

Prior work has advocated for improved diffing support in next-generation code review tools to better compare different versions of code under review [19]. To that end, this paper proposes AntiDiffamine to mitigate the impact of diff conflation.

## 8 Threats to Validity

In this section, we discuss the potential threats to construct, internal, and external validity.

***Construct Validity.*** While classifying ancestral relationships, we encountered exceptions affecting classification and the consistency of rebase reconstruction. To mitigate this, we document, inspect, and report reasons for unresolvable pairs, introducing targeted strategies and re-running the process for consistency. We find 21.5% of pairs are unresolvable, 99.94% due to deleted commits or branches, suggesting maintainers may adopt varying artifact retention or branch management policies. In practice, AntiDiffamine would be applied immediately after a force-push, while relevant artifacts still exist.

***Internal Validity.*** The distribution of ancestral relationships among force-push pairs may shift if currently unresolvable cases become resolvable, and additional relationships may exist among these unavailable artifacts. For example, 70 unresolvable cases (0.01% of all pairs) lack a common ancestor, typically involving commits without a valid parent (*e.g.,* an orphan branch). Nonetheless, such potential distributional changes do not invalidate our findings on the identified hetero-ancestral pairs and diff conflation.

***External Validity.*** Our studied dataset is collected from GitHub. However, conflation exists in other pull-based platforms, such as GitLab and Codeberg. Thus, the prevalence and impact of conflated diffs may vary across platforms. Nonetheless, our study is large-scale (1,445 projects, 935,011 PRs) and GitHub is the most broadly adopted of such platforms. Moreover, our implementation is built on Git primitives, making it extensible to other Git-based platforms.

The case study on build fail-fix pairs is conducted on one project, and the prevalence of hetero-ancestral pairs may not generalize to other projects. Nonetheless, Kubernetes is a large, active, and well-maintained project. Moreover, an example from BugSwarm confirms the existence of such hetero-ancestral pairs in build fail-fix pairs.

We adopt the default Myers diffing algorithm in Git, which may produce slightly different diffs compared to using minimal, patience, or histogram algorithms. These differences do not invalidate our findings on diff conflation, which is introduced at the commit level, although the reported diff size may vary. Furthermore, AntiDiffamine is compatible with other diffing algorithms and can adopt the preferred algorithm when presenting diffs to developers.

## 9 Conclusion and Implications

In this paper, we describe the force-push-based diff conflation problem in pull-based development. Analyzing 935,011 PRs across 1,445 projects, we observe 568,218 force-push pairs spanning 242,624 PRs in 1,366 projects, of which 210,837 hetero-ancestral pairs are affected by conflation. We propose AntiDiffamine, which focuses diffs on PR-specific changes, and can reduce the size of conflated diffs by orders of magnitude. Building on this, we conduct a Kubernetes fail-fix pair case study and find similar conflation-prone hetero-ancestral relationships. Finally, comparing three treatments of hetero-ancestral pairs, we show that ignoring or excluding conflation-prone pairs introduces bias in downstream analyses.

***Facilitating Code Review.*** Developers have long expressed concerns about conflated diffs. AntiDiffamine addresses this issue by providing a PR-specific diff that isolates changes within the PR scope. To integrate AntiDiffamine into existing developer workflows, we implement a GitHub Action that triggers automatically when a PR receives new commits. When that action detects a hetero-ancestral pair among the commits of the PR, it invokes AntiDiffamine and posts a comment that provides a URL where a deconflated diff can be viewed.

With the AntiDiffamine Action installed, wasted code review effort can be avoided in 45.9% of cases, as the AntiDiffamine Action would notify reviewers that there are no actual changes to review (Observation 5). When there are actual changes, the misleading and exhaustive diff can be shortened from hundreds or thousands of files and lines to a few, and merge conflicts are highlighted so reviewers can see which changes the PR author kept (Observations 3 and 4). This benefit extends beyond force-pushed revisions—Observations 8 and 9 that show 11.5% of hetero-ancestral build fail-fix pairs are pure branch synchronization and can likewise be marked as requiring no review.

***Encouraging Fine-Grained Benchmarks.*** Diff conflation can discourage researchers from leveraging the rich yet inaccurate PR revisions, and improperly treating conflation can bias downstream analyses. Existing benchmarks typically rely on the merged state of PRs on the main branch, where discussion and code refinement have already occurred, thereby abandoning the iterative code evolution process. AntiDiffamine enables the construction of datasets at a finer granularity across settings such as fail-fix pairs, and potentially bug-inducing and -fixing pairs and competing PRs. Future work should explore additional use cases of AntiDiffamine.

***Future Work.*** AntiDiffamine is motivated by discussions of developers and lack of PR revision datasets; however, the perceived benefits of adopting AntiDiffamine remain unclear. Future work should investigate the perception of both groups.

## Acknowledgments

We gratefully acknowledge the financial support of: (1) JSPS for the KAKENHI grants (JP25K22845, JP26H02500, JP26K21198) and Bilateral Program grant JPJSBP120239929; (2) Japan Science and Technology Agency (JST) as part of Adopting Sustainable Partnerships for Innovative Research Ecosystem (ASPIRE), Grant No. JPMJAP2415, and (3) the Inamori Research Institute for Science for supporting Yasutaka Kamei via the InaRIS Fellowship.

## Data Availability Statement

All data and scripts that produced the results in this paper are available in our replication package [37].

## References

[1] Alfred V. Aho. 1991. *Algorithms for finding patterns in strings.* MIT Press, 255–300. https://doi.org/10.1016/B978-0-444-88071-0.50010-2

[2] Gabin An, Jingun Hong, Naryeong Kim, and Shin Yoo. 2023. Fonte: Finding Bug Inducing Commits from Failures. In *Proc. of the 45th Int'l Conf. on Software Engineering*. 589–601. https://doi.org/10.1109/ICSE48619.2023.00059

[3] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. 2013. The influence of non-technical factors on code review. In *Proc. of the 20th Working Conf. on Reverse Engineering*. 122–131. https://doi.org/10.1109/WCRE.2013.6671287

[4] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *Proc. of the 14th Int'l Conf. on Mining Software Repositories*. 447–450. https://doi.org/10.1109/MSR.2017.24

[5] Markus Borg, Oscar Svensson, Kristian Berg, and Daniel Hansson. 2019. SZZ unleashed: an open implementation of the SZZ algorithm - featuring example usage in a study of just-in-time bug prediction for the Jenkins project. In *Proc. of the 3rd Int'l Workshop on Machine Learning Techniques for Software Quality Evaluation*. 7–12. https://doi.org/10.1145/3340482.3342742

[6] Caius Brindescu, Iftekhar Ahmed, Carlos Jensen, and Anita Sarma. 2020. An empirical investigation into merge conflicts and their effect on software quality. *Empirical Software Engineering* 25, 1 (2020), 562–590.

[7] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2011. Proactive detection of collaboration conflicts. In *Proc. of the 19th ACM SIGSOFT Symp. and the 13th European Conf. on Foundations of software engineering*. 168–178. https://doi.org/10.1145/2025113.2025139

[8] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2013. Early Detection of Collaboration Conflicts and Risks. *IEEE Trans. on Software Engineering* 39, 10 (2013), 1358–1375. https://doi.org/10.1109/TSE.2013.28

[9] An Ran Chen, Tse-Hsun (Peter) Chen, and Junjie Chen. 2022. How Useful is Code Change Information for Fault Localization in Continuous Integration?. In *Proc. of the 37th Int'l Conf. on Automated Software Engineering*. 1–12. https://doi.org/10.1145/3551349.3556931

[10] Jialong Chen, Xander Xu, Hu Wei, Chuan Chen, and Bing Zhao. 2026. *SWE-CI: Evaluating Agent Capabilities in Maintaining Codebases via Continuous Integration.* Technical Report 2603.03823. arXiv. https://doi.org/10.48550/arXiv.2603.03823

[11] Flávia Coelho, Nikolaos Tsantalis, Tiago Massoni, and Everton L. G. Alves. 2021. An Empirical Study on Refactoring-Inducing Pull Requests. In *Proc. of the 15th Int'l Symp. on Empirical Software Engineering and Measurement*. 1–12. https://doi.org/10.1145/3475716.3475785

[12] Valentin Dallmeier and Thomas Zimmermann. 2007. Extraction of bug localization benchmarks from history. In *Proc. of the 22nd Int'l Conf. on Automated Software Engineering*. 433–436. https://doi.org/10.1145/1321631.1321702

[13] Cleidson R. B. de Souza, David Redmiles, and Paul Dourish. 2003. "Breaking the code", moving between private and public work in collaborative software development. In *Proc. of the Int'l Conf. on Supporting Group Work*. 105–114. https://doi.org/10.1145/958160.958177

[14] Georgios Gousios, Martin Pinzger, and Arie Van Deursen. 2014. An exploratory study of the pull-based software development model. In *Proc. of the 36th Int'l Conf. on Software Engineering*. 345–355. https://doi.org/10.1145/2568225.2568260

[15] Tao Ji, Liqian Chen, Xin Yi, and Xiaoguang Mao. 2020. Understanding Merge Conflicts and Resolutions in Git Rebases. In *Proc. of the 31st Int'l Symp. on Software Reliability Engineering*. 70–80. https://doi.org/10.1109/ISSRE5003.2020.00016

[16] Jing Jiang, Jiangfeng Lv, Jiateng Zheng, and Li Zhang. 2022. How Developers Modify Pull Requests in Code Review. *IEEE Trans. on Reliability* 71, 3 (2022), 1325–1339. https://doi.org/10.1109/TR.2021.3093159

[17] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *Proc. of the 12th Int'l Conf. on Learning Representations*. https://doi.org/10.48550/arXiv.2310.06770

[18] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proc. of the Int'l Symp. on Software Testing and Analysis*. 437–440. https://doi.org/10.1145/2610384.2628055

[19] Oleksii Kononenko, Olga Baysal, and Michael W. Godfrey. 2016. Code review quality: how developers see it. In *Proc. of the 38th Int'l Conf. on Software Engineering*. 1028–1038. https://doi.org/10.1145/2884781.2884840

[20] Klaus Krippendorff. 2004. *Content Analysis: An Introduction to Its Methodology* (2nd ed.). Sage Publications. https://doi.org/10.4135/9781071878781

[21] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Trans. on Software Engineering* 41, 12 (2015), 1236–1256. https://doi.org/10.1109/TSE.2015.2454513

[22] Zhixing Li, Yue Yu, Tao Wang, Shanshan Li, and Huaimin Wang. 2022. Opportunities and Challenges in Repeated Revisions to Pull-Requests: An Empirical Study. *ACM Human-Computer Interaction* 6, CSCW2 (2022). https://doi.org/10.1145/3555208

[23] Zhixing Li, Yue Yu, Tao Wang, Gang Yin, ShanShan Li, and Huaimin Wang. 2022. Are You Still Working on This? An Empirical Study on Pull Request Abandonment. *IEEE Trans. on Software Engineering* 48, 6 (2022), 2173–2188. https://doi.org/10.1109/TSE.2021.3053403

[24] Zhixing Li, Yue Yu, Minghui Zhou, Tao Wang, Gang Yin, Long Lan, and Huaimin Wang. 2022. Redundancy, Context, and Preference: An Empirical Study of Duplicate Pull Requests in OSS Projects. *IEEE Trans. on Software Engineering* 48, 4 (2022), 1309–1335. https://doi.org/10.1109/TSE.2020.3018726

[25] Cleyciane Lima and Daricelio Soares. 2022. On the Nature of Duplicate Pull Requests: An Empirical Study Using Association Rules. In *Proc. of the 16th Brazilian Symp. on Software Components, Architectures, and Reuse*. 68–75. https://doi.org/10.1145/3559712.3559722

[26] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *Proc. of the 26th Int'l Conf. on Software Analysis, Evolution and Reengineering*. 468–478. https://doi.org/10.1109/SANER.2019.8667991

[27] Giacomo Marzi, Marco Balzano, and Davide Marchiori. 2024. K-Alpha Calculator–Krippendorff's Alpha Calculator: A user-friendly tool for computing Krippendorff's Alpha inter-rater reliability coefficient. *MethodsX* 12 (2024), 102545. https://doi.org/10.1016/j.mex.2023.102545

[28] Costain Nachuma and Minhaz Zibran. 2026. *When AI Teammates Meet Code Review: Collaboration Signals Shaping the Integration of Agent-Authored Pull Requests.* Technical Report 2602.19441. arXiv. https://doi.org/10.48550/arXiv.2602.19441

[29] Matheus Paixao and Paulo Henrique Maia. 2019. Rebasing in Code Review Considered Harmful: A Large-Scale Empirical Investigation. In *2019 19th Int'l Working Conf. on Source Code Analysis and Manipulation*. 45–55. https://doi.org/10.1109/SCAM.2019.00014

[30] Daniel Ramos, Hailie Mitchell, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. 2023. MELT: Mining Effective Lightweight Transformations from Pull Requests. In *Proc. of the 38th Int'l Conf. on Automated Software Engineering*. 1516–1528. https://doi.org/10.1109/ASE56229.2023.00117

[31] Solal Rapaport, Laurent Pautet, Samuel Tardieu, and Stefano Zacchiroli. 2025. Altered Histories in Version Control System Repositories: Evidence from the Trenches. In *Proc. of the 40th Int'l Conf. on Automated Software Engineering*. 2184–2195. https://doi.org/10.1109/ASE63991.2025.00181

[32] Peter Rigby, Brendan Cleary, Frederic Painchaud, Margaret-Anne Storey, and Daniel German. 2012. Contemporary Peer Review in Action: Lessons from Open Source Development. *IEEE Software* 29, 6 (2012), 56–61. https://doi.org/10.1109/MS.2012.24

[33] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: a case study at google. In *Proc. of the 40th Int'l Conf. on Software Engineering: Software Engineering in Practice*. 181–190. https://doi.org/10.1145/3183519.3183525

[34] Ripon K. Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R. Prasad. 2018. Bugs.jar: a large-scale, diverse dataset of real-world Java bugs. In *Proc. of the 15th Int'l Conf. on Mining Software Repositories*. 10–13. https://doi.org/10.1145/3196398.3196473

[35] André Silva, Matias Martinez, Benjamin Danglot, Davide Ginelli, and Martin Monperrus. 2021. *FLACOCO: Fault Localization for Java based on Industry-grade Coverage.* Technical Report 2111.12513. arXiv. https://doi.org/10.48550/arXiv.2111.12513

[36] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes?. In *Proc. of the Int'l Workshop on Mining Software Repositories*. 1–5. https://doi.org/10.1145/1082983.1083147

[37] Gengyi Sun, Georges Aaron Randrianaina, Tao Xiao, Yasutaka Kamei, and Shane McIntosh. 2026. *Recovering Revisions of Pull Requests with Altered History.* https://doi.org/10.5281/zenodo.21410986

[38] Yuheng Tang, Kaijie Zhu, Bonan Ruan, Chuqi Zhang, Michael Yang, Hongwei Li, Suyue Guo, Tianneng Shi, Zekun Li, Christopher Kruegel, Giovanni Vigna,

Dawn Song, William Yang Wang, Lun Wang, Yangruibo Ding, Zhenkai Liang, and Wenbo Guo. 2026. DevOps-Gym: Benchmarking AI Agents in Software DevOps Cycle. In *The 14th Int'l Conf. on Learning Representations*. https://doi.org/10.48550/arXiv.2601.20882

[39] David A. Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T. Devanbu, Bogdan Vasilescu, and Cindy Rubio-Gonzalez. 2019. BugSwarm: Mining and Continuously Growing a Dataset of Reproducible Failures and Fixes. In *Proc. of the 41st Int'l Conf. on Software Engineering*. 339–349. https://doi.org/10.1109/ICSE.2019.00048

[40] Chuck Walrad and Darrel Strom. 2002. The Importance of Branching Models in SCM. *Computer* 35, 9 (2002), 31–38. https://doi.org/10.1109/MC.2002.1033025

[41] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. 2020. BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies. In *Proc. of the 28th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering*. 1556–1560. https://doi.org/10.1145/3368089.3417943

[42] Boyang Yang, Jiadong Ren, Shunfu Jin, Yang Liu, Feng Liu, Bach Le, and Haoye Tian. 2025. *Enhancing repository-level software repair via repository-aware knowledge graphs*. Technical Report 2503.21710. arXiv. https://doi.org/10.48550/arXiv.2503.21710

[43] Yue Yu, Zhixing Li, Gang Yin, Tao Wang, and Huaimin Wang. 2018. A dataset of duplicate pull-requests in github. In *Proc. of the 15th Int'l Conf. on Mining Software Repositories*. 22–25. https://doi.org/10.1145/3196398.3196455