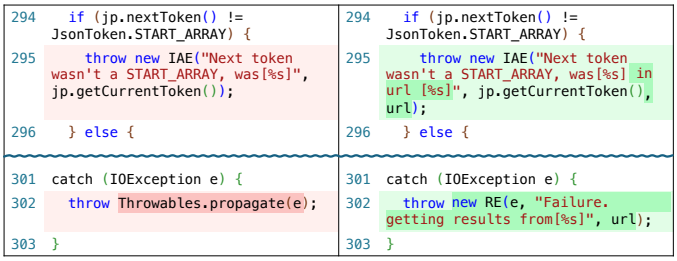
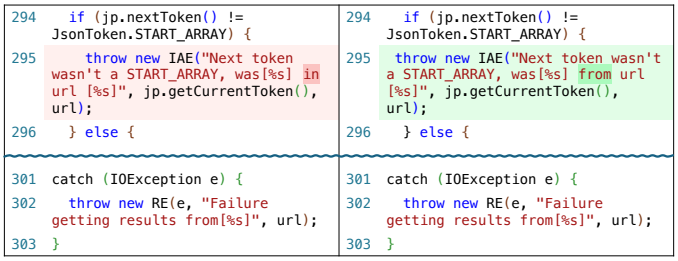
Published version: Daiki Muto, Takayoshi Ueno, Shinpei Hayashi, "D-Diff: An Interactive Environment for Adjusting Commit Boundaries Based on an Editable 3-way Diff,", in Proceedings of the 14th IEEE Working Conference on Software Visualization, pp. 198–208, 2026, https://doi.org/10.1109/VISSOFT71254.2026.00036

D-Diff: An Interactive Environment for Adjusting Commit Boundaries Based on an Editable 3-way Diff

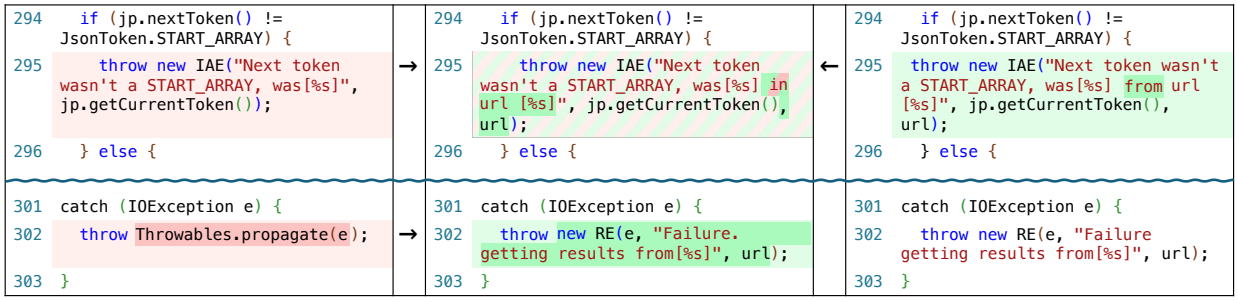
Daiki Muto, Takayoshi Ueno, Shinpei Hayashi
School of Computing, Institute of Science Tokyo, Tokyo 152–8550, Japan
{muto, ueno, hayashi}@se.comp.isct.ac.jp



(a) The first commit shown in a typical diff editor.



(b) The second commit shown in a typical diff editor.



(c) 3-way diff display of D-Diff.

Fig. 1. Example of two consecutive commits with a partially altered boundary: (a, b) their diffs displayed separately and (c) integrated by D-Diff.

***Abstract*—In version control, it is recommended that each commit include only changes related to one task. To follow this recommendation, developers may need to adjust commit boundaries, that is, to compare and modify the diffs between two consecutive commits. Existing tools either display only a single diff at a time, forcing developers to rely on their memory when comparing diffs, or display three files simultaneously without showing the two consecutive diffs, forcing developers to infer them; both increase their cognitive load and hamper the adjustment. As a first step toward supporting this process, we propose D-Diff, an interactive diff adjustment environment for two consecutive commits that each involve the same single file. Based on a 3-way diff display, D-Diff integrates the two diffs into a single compact view, which also provides a way to modify them. A within-subjects user study with 8 participants showed that D-Diff significantly outperforms a baseline tool in terms of efficiency, reducing the median adjustment time from 530 seconds to 230 seconds (approximately 57%). D-Diff also received a higher percentage of positive responses across all usability questionnaire items, while no statistically significant difference was found in accuracy.**

Video demonstration: https://youtu.be/ra47jbrP5ZA

***Index Terms*—version control, adjusting commit boundaries, 3-way diff, software visualization**

I. Introduction

In version control, it is recommended that each commit include only changes related to one task; such a commit is called a Task Level Commit (TLC) [1]. TLCs offer advantages such as facilitating the understanding of changes during code reviews [2] and reducing noise in repository mining [3]. However, developers often create commits that mix multiple intentions [3], [4]. Such commits, called Composite Commits (CC), hinder the understanding, reuse, and reverting of changes [5].

To follow this recommendation, developers may need to adjust commit boundaries. In this paper, *adjusting commit boundaries* refers to the process of comparing and modifying the diffs between two consecutive commits. This process is part of restructuring the entire commit history. A typical scenario is that a developer adjusts the boundaries of local commits before pushing them to a shared repository, since rewriting already shared history is discouraged. In this process, developers can leverage approaches that automatically split CCs into TLCs [6]–[8], increasingly driven by large language

models, to obtain initial commit boundaries. However, what counts as a single task varies across organizations and projects, so an automatic split rarely matches the developer's intent out of the box; a human must still inspect it and shift changes across commit boundaries to finalize it. Automatic splitting and manual adjustment are complementary, but the interactive support for this last-mile correction remains limited.

While several tools [9]–[11] and standard methods provided by version control systems (VCSs) can be used for adjusting commit boundaries, they are not well-suited for this process. Most of them display only a single diff at a time, as illustrated in Figs. 1a and 1b, requiring developers to memorize the not-displayed diff when comparing diffs. The others display three files simultaneously but show only where the files differ, requiring developers to infer the two consecutive diffs from the displayed differences.

In this paper, to address this challenge, we propose D-Diff, an interactive diff adjustment environment. Instead of manipulating commits indirectly through rebase commands and hunk-by-hunk staging, D-Diff lets developers directly edit the state that lies between the two commits: it shows the two diffs around this shared boundary version using a 3-way diff display, as shown in Fig. 1c, and the commits are updated as the developer moves changes across the boundary or edits it. This simultaneous view reduces the cognitive load of comparing diffs. As a first step toward supporting this process, D-Diff focuses on cases where the two commits each involve the same single file.

The primary contributions of this paper are as follows:

- We characterize adjusting commit boundaries as the comparison and modification of the diffs between two consecutive commits, and show that existing tools are ill-suited to it because they either display only a single diff at a time, forcing developers to rely on their memory when comparing diffs, or display three files simultaneously without showing the two consecutive diffs, forcing developers to infer them.
- We propose D-Diff, an interactive diff adjustment environment for two consecutive commits that each involve the same single file. D-Diff integrates, in one screen, (i) the simultaneous comparison of two diffs by adapting a 3-way diff display, with a coloring scheme that distinguishes the changes of each commit from those shared by both, and (ii) the modification of diffs in multiple ways: line-level movement, hunk-level movement, and direct editing. The alignment reuses established differencing techniques rather than a new algorithm; the contribution is the interaction design that turns editing the shared boundary state into a commit-boundary operation.
- We provide empirical evidence, through a within-subjects user study with 8 participants, that D-Diff significantly improves the efficiency of adjusting commit boundaries over a baseline tool (median time reduced from 530 to 230 seconds, about 57%), and is rated higher in usability, while no statistically significant difference in accuracy is observed.

TABLE I
COMPARISON OF EXISTING TOOLS USABLE FOR ADJUSTING COMMIT BOUNDARIES

Tool	Comparison Method	Modification Method: Editing	Modification Method: Moving
git command	Single diff + memory	✓	
GUI clients and IDEs	Single diff + memory	✓	✓
Jujutsu [12] + Meld [13] / Diffedit3 [14]	Single diff + memory	✓	✓
ChTree [9]	Single diff + memory		✓
ChangeBeadsThreader [10]	Single diff + memory		✓
SmartCommit [11]	Single diff + memory		✓
diff3 [15]	Three files simultaneously		
vimdiff (Vim [16])	Three files simultaneously	✓	✓

The structure of this paper is as follows. Section II clarifies the requirements for the proposed approach. Section III reviews related work. Section IV presents the design and implementation of D-Diff. Section V evaluates it through a user study. Section VI concludes.

II. REQUIREMENTS FOR THE PROPOSED APPROACH

Adjusting commit boundaries consists of comparing the contents of diffs included in two commits and modifying the diffs so that changes are included in the correct commit. Developers need to examine both commits simultaneously to determine which commit each change should belong to, and to check how the boundary changes as they modify the diffs. Given three versions of a file, s, s', and s'', the first commit is represented by the diff between s and s', and the second commit is represented by the diff between s' and s''. In this view, the intermediate version s' is the boundary shared by the two commits, and modifying the boundary can be modeled as editing this intermediate version.

As a concrete example, Fig. 2 shows the workflow of a developer adjusting the boundary between two consecutive commits with Visual Studio Code (VS Code) [17] and the GitLens [18] extension, the setup we adopt as the baseline in our evaluation (Section V). Modifying the commit to which a change belongs requires an interactive rebase. More fundamentally, to decide which changes belong to each commit, the developer must compare the diffs of the two commits, yet only one of them can be displayed at a time. The developer therefore opens the first commit's diff, memorizes the relevant lines, switches to the second commit's diff, and switches back, relying on memory because the two diffs are never visible together. This comparison recurs throughout the task.

As Table I shows, several tools can be used to adjust commit boundaries, but none of them is well-suited to it. These tools other than diff3 [15] and vimdiff modify diffs by editing changes, moving them, or both; notably, Jujutsu [12] combined with the Meld [13] or diffedit3 [14] diff editors provides an editable pane placed between the two versions of a diff. However, they display only the diff of one commit at a time. Developers using them therefore cannot see the two diffs simultaneously, nor observe how the commit boundary changes as they modify the diffs, and must instead rely on their memory. Conversely, diff3 and vimdiff display three

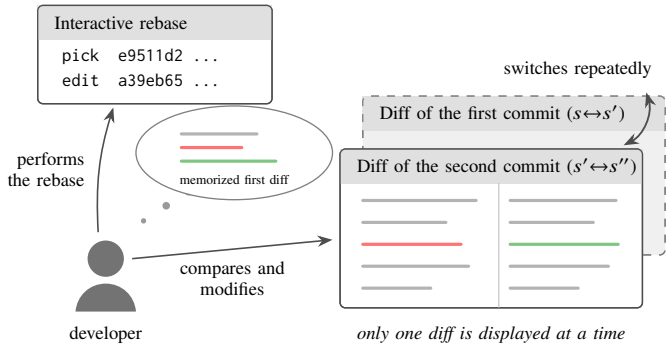


Fig. 2. Workflow of adjusting commit boundaries with VS Code and GitLens.

files simultaneously. However, both of them only show where the files differ from each other: diff3 outputs the differing ranges of lines, and vimdiff highlights them. Developers must therefore still infer the two consecutive diffs against the middle version. We argue that these limitations impair work efficiency and may lead to incorrect judgments.

To overcome these limitations, the proposed approach must satisfy two requirements.

- **Requirement 1:** The approach must allow developers to compare the diffs without relying on their memory.
- **Requirement 2:** The approach must provide a method for modifying diffs.

III. Related Work

Tools for Modifying Commit History. Git and the tools built around it provide general-purpose means of modifying commit history. Using interactive rebase with `git rebase -i`, developers can reorder, combine, and split commits. Furthermore, `git add -p` allows developers to interactively stage changes on a hunk-by-hunk basis. GUI clients such as GitKraken [19], GitHub Desktop [20], and Sourcetree [21] are widely used to improve the usability of these git commands. Some tools further support moving changes across existing commit boundaries. git absorb [22] automatically distributes staged changes as fixup commits to the earlier commits that last modified the corresponding lines. Jujutsu [12], a Git-compatible VCS, provides commands such as `jj squash` that move changes from one commit into another. However, all of these general-purpose tools display only one commit at a time, so developers cannot check how the boundary between two consecutive commits changes as they operate.

Apart from these general-purpose tools, several TLC creation support tools have been proposed. Sothornprapakorn et al. proposed a method to visualize changes in a tree structure and implemented ChTree [9]. Yamashita et al. proposed a method to untangle changes by splitting and merging automatically generated change clusters and implemented ChangeBeadsThreader [10]. Shen et al. proposed SmartCommit [11], which performs graph-based clustering and allows developers to edit the results through two types of operations: adjusting thresholds and moving changes between clusters. These tools are designed to split a single set of changes into groups for creating TLCs. They display only the single diff being decomposed, and their modification is limited to regrouping changes.

Change Untangling. Many approaches have been proposed to automatically untangle a CC, i.e., to split it into groups of changes, each of which is related to a single task. Some approaches group changes based on static analysis or heuristic rules [4], [23]–[25]. Dias et al. applied machine learning to fine-grained change histories recorded in the IDE [26]. More recent approaches construct graph representations of changes and apply clustering [6], [11], [27]–[29], or leverage large language models [7], [8]. Guo and Song proposed ChgCutter, which interactively decomposes a CC so that a subset of the changes can be reviewed and tested in isolation [30]. While these approaches untangle a CC after it has been created, Kirinuki et al. proposed a method that warns developers of tangled changes at commit time to prevent CCs [31]. These approaches split a single CC into TLCs, whereas D-Diff supports adjusting the boundary between two consecutive commits that have already been separated. They are therefore complementary: automatically untangled results that do not match a project's TLC standards can be adjusted with D-Diff.

Diff Visualization. A line of research enriches the conventional text-based diff to help developers comprehend changes. Some approaches augment the diff with the semantics of the change. RefactorInsight [32] and RAID [33] annotate the diff with detected refactorings, in the IDE and on GitHub, respectively. ChangePrism [34] overlays refactorings and micro-changes on the diff. ChangeViz [35] adds method call information to the GitHub pull request diff. Others refine the diff itself. Spike [36] highlights fine-grained intra-line changes in the editor. DiffViz [37] visualizes edit scripts independently of the underlying differencing algorithm, supporting granularities up to the abstract syntax tree level. These approaches enrich a single diff, whereas D-Diff displays two consecutive diffs simultaneously to support adjusting commit boundaries.

3-way Diff Display. 3-way diff display is a method that presents three versions of source code simultaneously.

Some general-purpose tools can display three files simultaneously. The diff3 command [15] compares three files and outputs the ranges of lines that differ among them. However, it provides no means of editing and thus cannot be used to adjust commit boundaries. The vimdiff command, a diff display mode of the Vim [16] editor, accepts three files and displays them in a Side-by-Side Diff format, allowing the developer to edit the displayed files and to copy a differing hunk from one file to another. However, its highlighting only shows where each file differs from the other two, not the two consecutive diffs against the middle version. Developers must therefore infer which commit each difference belongs to. Moreover, it lacks support for preserving the two outer versions and for integrating the edits into the commit history.

The most widely used form of 3-way diff display is the merge editor. For example, GUI clients and IDEs such as VS Code provide such editors. They typically place the two

conflicting source codes side by side in the upper panes and display the merge result in the lower pane. These editors, however, assume two changes diverging from a common base, not two consecutive diffs in a linear history.

More recently, another type of 3-way diff display has emerged for modifying commit history: the two versions of a diff are shown in the left and right panes, and an editable pane is placed in the middle. Jujutsu experimentally supports this style of editing with two external diff editors. One is the Meld [13] merge tool, whose merge view is repurposed for diff editing. The other is diffedit3 [14], a browser-based editor built for this purpose. These editors are invoked from commands such as `jj squash -i`, which moves selected changes between two commits. The developer selects the changes to be moved by editing the middle pane directly or by moving hunks into it from the side panes. These editors share with D-Diff the idea of directly editing an intermediate version. However, the three panes present only the diff of the single commit from which the changes are moved. The two consecutive diffs that result from the operation are never displayed together with their shared boundary version. The developer must inspect the resulting diffs one by one after the operation, e.g., with `jj diff`.

IV. D-Diff: A Diff Adjustment Environment

In this section, we propose D-Diff, an interactive diff adjustment environment that satisfies the requirements mentioned above. In this paper, as a first step toward solving the problem of adjusting commit boundaries, we limit the scope to cases where the two commits each involve the same single file. This single-file, two-commit case is the building block of the general setting: as long as adjusting the boundary does not require comparing changes across different files, a multi-file commit decomposes into independent per-file adjustments, and a longer commit sequence into consecutive pairs.

A. Overview

Figure 1c, introduced in Section I, shows an example of applying D-Diff to two TLCs[1] with partially altered commit boundaries. As shown in the figure, D-Diff simultaneously visualizes two consecutive diffs of the same file. This enables developers to compare the diffs without relying on their memory, satisfying Requirement 1. In addition, D-Diff provides two operations to support modifying diffs, described later: direct editing and moving changes. This satisfies Requirement 2.

D-Diff employs a 3-way diff display based on the intermediate version s' shared by the two consecutive diffs, introduced in Section II. A straightforward alternative is to display the two diffs independently, as in Figs. 1a and 1b. This removes the need to switch views, but the boundary version s' is then represented twice, and each copy is aligned differently against its own counterpart. Relating a change across the two diffs therefore still requires re-locating the corresponding line in the other view and joining the two copies mentally, and after each modification, its dual effect, leaving one diff and entering the other, appears in two separate places. The 3-way diff display eliminates this residual reliance on memory by materializing the correspondence: the single shared copy of s' serves as the common coordinate, each line shows its role in both commits at once, and the dual effect of an edit appears on the same line. This display also matches the mental model of adjusting boundaries: developers can reason about whether each change belongs before or after the boundary.

[1]https://github.com/apache/druid/compare/dd20950^...a39eb65

B. Comparison

D-Diff arranges three versions of source codes side by side in a 3-way diff layout. The difference between the left and center source code corresponds to the first commit, and the difference between the center and right source code corresponds to the second commit.

D-Diff colors the background using a combination of line-level and character-level highlighting, consistent with existing diff displays. The left source code corresponds to the deletion in the first diff, and the right source code corresponds to the addition in the second diff. Accordingly, as seen in line 295 of Fig. 1c, the left and right source code are assigned the same background colors as in a typical diff editor. In contrast, the center source code corresponds to both the addition in the first diff and the deletion in the second diff. Lines or characters that represent only an addition or a deletion are displayed with background colors similar to those of a typical diff editor, as seen in line 302 of Fig. 1c. Lines or characters representing both an addition and a deletion are shown with a green and red diagonal stripe pattern, as seen in line 295. Green denotes an addition and red a deletion, as in a typical diff editor; unmodified content is left uncolored, and the cited line numbers are those shown in the editor gutter in Fig. 1c.

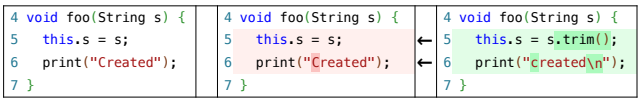
C. Modification

D-Diff provides two operations for modifying diffs: direct editing of commit boundaries and moving changes at different granularities. Direct editing allows boundaries to be placed at any position, but it requires significant effort. In contrast, moving changes is limited by available granularity, but it requires less effort. Therefore, moving changes serves as the primary operation, while direct editing is intended for finer modifications such as separating changes within the same line.

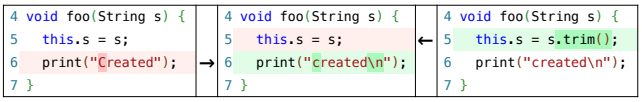
Direct Editing. Developers perform direct edits by modifying the center source code. Because the center source code is the boundary between the two commits, editing it lets developers place the commit boundary at any position.

Moving Changes. Moving changes is available at two levels of granularity: line level and hunk level (groups of consecutive lines). This operation is inspired by the buttons in IDE diff editors that allow developers to stage changes by hunk. Line-level movement is also provided because a hunk can span dozens of lines, making hunk-level movement alone too coarse-grained.

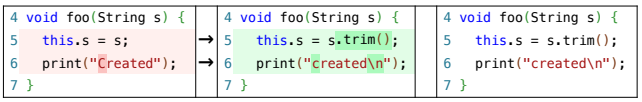
Developers can move changes by clicking the buttons located between the source codes. These buttons appear for lines



(a) State before line-level or hunk-level movement.



(b) Moving the change on Line 6 of (a) to the first commit.



(c) Moving the hunk on Lines 5–6 of (a) to the first commit.

Fig. 3. Modification by line-level or hunk-level movement.

or hunks where the content differs when comparing the outer (left or right) source code with the inner (center) source code. When a button is clicked, the corresponding part of the center source code is replaced with the content from the outer source code. Because the center version is the boundary, replacing part of it with the left version removes that change from the first diff and adds it to the second, and replacing it with the right version does the reverse. Moving is thus a structured special case of directly editing the boundary. Developers can toggle granularity by holding a specific modifier key.

Figure 3 applies the two granularities to the state in Fig. 3a. A line-level move sends the change on Line 6 to the first commit, as shown in Fig. 3b, and a hunk-level move sends the hunk on Lines 5–6 to the first commit, as shown in Fig. 3c.

D. Implementation

We implemented D-Diff as a web application using TypeScript and Nuxt.js. As shown in Fig. 4, this tool consists of two parts: (A) a header and (B) a diff adjustment editor. The header is an area that displays information about the two target commits, including the SHA and message of each commit, as well as a button for toggling the visibility of the messages. The diff adjustment editor provides the diff adjustment environment based on the 3-way diff display proposed in Section IV.

In the following, we describe how D-Diff constructs the 3-way diff display from its input. The input consists of three versions of the same file, s, s', and s'', where (s, s') corresponds to the first commit and (s', s'') to the second; the center version s' is shared by the two commits as their boundary. The construction proceeds in two sequential stages, one at the line level and one at the character level, whose results are then rendered on the editor.

At the line level, we obtain the three versions of the source code aligned line by line, together with a per-line classification. First, we compute the line-level diff between s and s' and that between s' and s'' using the Myers algorithm [38] provided by `jsdiff` [39]. Second, using the center version s' as the shared anchor, we align s and s'' to it: the first diff maps each line of s' to its corresponding line in s, and the second diff maps it to its corresponding line in s''. Third, we equalize the three columns row by row by inserting blank lines: within each changed region the shorter side is padded so that all three versions span the same number of rows, and the padding of the shared center is taken as the larger of the amounts the two diffs require. This construction follows the same principle as diff3 [15]: it anchors on the center version and composes two pairwise diffs rather than introducing a new alignment algorithm. We reuse this well-understood alignment principle intentionally, concentrating our contribution on the interaction layer. Finally, we classify each line as *unmodified*, *added*, *removed*, *both*, or *empty*. A center line is classified as *both* when it is added by the first commit and deleted by the second. Padding lines are classified as *empty*.

At the character level, we obtain a per-character classification within each hunk. A hunk is a group of consecutive lines in which either of the two versions is classified as *added*, *removed*, or *both*. For each hunk, we join its lines into one string per version and compute two character-level diffs with `diff-match-patch` [40]: one between s and s' and one between s' and s''. This library applies the Myers algorithm and then a semantic cleanup that improves the readability of the resulting diff. Each differing character is classified as *added*, *removed*, or *both*, as in the line-level classification. Because the two character-level diffs are computed independently, they can occasionally disagree on the classification of a shared span; resolving such cases more precisely is left for future work.

Finally, we render the three aligned versions of the source code on three editors placed side by side, implemented with the Monaco Editor [41]. Each editor is decorated with the line-level background colors from the line-level classification and the character-level highlighting from the character-level classification. The three editors are scroll-synchronized so that corresponding lines remain aligned. Only the center editor is editable to support direct editing. Movement buttons are placed on the lines or hunks whose content differs between the inner source code and an outer source code.

V. Evaluation

We set the following Research Questions (RQs) to evaluate the usefulness of the proposed approach D-Diff from the perspective of a developer adjusting commit boundaries.

- RQ_1 (Accuracy): Does D-Diff improve the accuracy of adjusting commit boundaries compared to the baseline?
- RQ_2 (Efficiency): Does D-Diff improve the efficiency of adjusting commit boundaries compared to the baseline?
- RQ_3 (Usability): Does D-Diff improve developer usability in adjusting commit boundaries compared to the baseline?

To address these RQs, we conducted a within-subjects user study. In this experiment, participants adjusted commit boundaries using both D-Diff and a baseline tool, and then completed questionnaires.

A. Experimental Setup

1) Tools: We used VS Code with the GitLens [18] extension installed as a baseline tool. No purpose-built tool adjusts

Fig. 4. Screenshot of D-Diff.

the boundary between two given consecutive commits, as the tools in Section III operate on a single composite commit, so the realistic choice for our baseline is a general VCS workflow, which moves changes across an existing boundary through interactive rebasing. VS Code was ranked as the most widely used IDE in the Stack Overflow Developer Survey 2025[2], so the comparison reflects practical use. While VS Code provides Git GUI operations as a standard feature, it does not support operations necessary for adjusting commit boundaries, such as interactive rebasing. GitLens is an extension that complements these missing features with a GUI.

We implemented the approach D-Diff proposed in Section IV and used it in the experiment. For this implementation, we enabled detailed logging. The logs include the start and end times of work on specific tasks, as well as modifications made through moving or editing changes.

2) Dataset: We built our own dataset of commit-boundary adjustment tasks, as none is available. Such a task needs both a misplaced boundary as its starting point and the correct boundary as ground truth for scoring, and cases that offer both are rare in real repositories. A boundary adjustment is usually made on local commits before pushing, so the misplaced starting state seldom reaches the shared repository, and a misplaced boundary that does remain there rarely carries a recorded correct target to score against. Finding cases suitable for the experiment in real repositories is therefore challenging, so we instead took clean TLC pairs as ground truth and altered their boundaries to obtain controlled misplaced starting states. As the source of clean TLC pairs, we used an artificially generated TLC dataset proposed in UTANGO [28]. CC datasets are either manually verified from real repositories or artificially generated; the manually verified ones [3], [42] are not publicly available, whereas artificially generated ones extract real TLC sequences and recombine them with `git cherry-pick`. Of the several available [11], [27], [28], only UTANGO met the criteria of the extraction step described later.

From this dataset, we extracted tasks usable in the experiment based on the limitations of the proposed approach. We first limited the 1,410 TLC sequences in the UTANGO dataset to those of length 2, obtaining 536 commit pairs. Next, we restricted the pairs to those where the two commits each involve the same single file, obtaining 30 commit pairs.

Finally, we performed a visual inspection to select only TLC pairs suitable for the experiment, obtaining 8 commit pairs. The first criterion was that the commit message accurately described the changes contained in the TLC. This is because the commit message is provided to participants as a guideline for the adjusting task in our experiment. An inaccurate message would prevent participants from identifying the correct state to be achieved. The second criterion was that the changes were non-trivial, ensuring a meaningful task.

We set task labels for each of the 8 selected pairs. We divided the tasks into two groups (A and B) of 4 tasks each to allow each participant to use both tools. We imposed two constraints on this division. First, both groups contained the same number of tasks requiring in-line separation (2 out of 4 each). Second, the per-repository difference in task count between the groups was at most one. Subject to these constraints, we minimized the inter-group gap in the sum of Levenshtein distances. Each distance was measured between the original boundary from the TLC pair in the dataset and the commit boundary input to the participant (described later). Table II shows the selected TLC pairs and the group division.

For the selected TLC pairs, we altered the commit boundaries line by line, the simplest choice for this first step, and used the resulting pairs as input for the participants in the experiment. First, we mapped the corresponding lines between the two commits and listed the lines where changes occurred in either commit. From that list, we randomly selected 20% of the lines and moved those lines to the opposite commit. Note that for lines where changes were made in both commits, we moved them together to one of the two commits chosen at random. Restoring such a line requires character-level separation rather than a line move, which is why some tasks require the in-line separation marked in Table II. These listing and movement operations were repeated until the resulting commit boundaries became syntactically correct.

[2]https://survey.stackoverflow.co/2025/technology#1-dev-id-es

TABLE II
TLC PAIRS SELECTED AS TASKS

Group	Task	Repository	Commit Pair	Filename	In-line Separation	Distance
A	A1	ReactiveX/RxJava	`b611684`, `ffbfb67`	BlockingObservable.java		56
	A2	square/retrofit	`f78f2fd`, `5cadd33`	RestAdapter.java		127
	A3	elastic/elasticsearch	`35d947b`, `742607e`	RestIndicesAction.java	✓	28
	A4	elastic/elasticsearch	`83a61ca`, `c3ccfe0`	BoolFilterParser.java	✓	3
	Total					214
B	B1	elastic/elasticsearch	`0d2675e`, `92a6968`	ShardIndexingService.java		215
	B2	elastic/elasticsearch	`112935f`, `b143400`	IpFieldMapper.java		25
	B3	NationalSecurityAgency/ghidra	`3813583`, `f15798c`	VarnodeAST.java	✓	26
	B4	square/retrofit	`478e1dd`, `0f59dc7`	SimpleXMLConverter.java	✓	64
	Total					330

B. Experimental Process

We recruited 8 participants from the authors' affiliated institution, requiring students majoring in computer science with at least two years of Java programming experience. Each participant received a 3,000 JPY (approx. $20) Amazon gift card as compensation for their participation. In accordance with the ethics screening criteria of the authors' affiliated institution, this study was determined not to require a detailed ethical review. Before starting the task, we provided the participants with preliminary explanations and tutorials through pre-recorded videos, covering the background of the experiment, the usage of the two tools, and the experimental procedure, followed by one practice task with each tool. To mitigate the difference in required operations between the tools, the video for the baseline tool was given more than twice the length of that for D-Diff (15 vs. 7 minutes).

We applied counterbalancing to the assignment of experimental conditions to minimize order and learning effects. Each participant used different tools across two sessions and addressed different task groups. We adopted a 4×4 Latin square based on Williams' design [43] to eliminate the carryover effect from preceding tasks. This design ensures that each task appears equally in every position, and that the task preceding any given task varies among participants.

We asked the participants to adjust commit boundaries according to their assignment. The original TLC commit messages served as the guideline for the adjustment task. Since the baseline tool does not support detailed logging, we instructed the participants to record their screens and measure the time taken for each task. The screen recording of one participant was lost when the recording tool crashed.

After the participants finished the adjusting commit boundaries tasks, we asked them to complete a questionnaire.

C. RQ_1: Accuracy

1) Study Design: We use the improvement rate of the center source code as an evaluation metric for accuracy. Let s^{in} be the center source code presented as input to the participant, and s^{out} be the center source code obtained by the participant upon completion of the task. Let s^* be the correct center source code, i.e., the original center source code of the TLC pair in the dataset. Let $d(a, b)$ be the Levenshtein distance between source codes a and b. The distance is calculated in characters, counting whitespace characters such as newlines, tabs, and spaces as ordinary characters, so that differences in formatting are also reflected as errors. The improvement rate is then defined as follows:

$$\text{Imp} = \begin{cases} 1 - \dfrac{d(s^{\text{out}}, s^*)}{d(s^{\text{in}}, s^*)} & \text{if } d(s^{\text{out}}, s^*) \leq d(s^{\text{in}}, s^*), \\ \dfrac{d(s^{\text{in}}, s^*)}{d(s^{\text{out}}, s^*)} - 1 & \text{if } d(s^{\text{out}}, s^*) > d(s^{\text{in}}, s^*). \end{cases}$$

Imp measures how much the Levenshtein distance from the correct answer shrinks relative to the input: Imp = 1 indicates a perfect match, Imp = 0 no improvement, and a negative value a deterioration. We treat Imp as a proxy for the correctness of the adjustment rather than a validated accuracy measure; it scores the character-exact distance to a single reference s^*, so a functionally correct but differently formatted result still counts as imperfect. Because both tools are scored by this same metric, it penalizes neither tool over the other and does not bias their comparison.

We evaluate the difference between the tools based on the improvement rate. We treat the participant as the primary unit of analysis: for each participant, we take the median of the improvement rates over the 4 tasks performed with each tool, obtaining one pair of values per participant, and perform a Wilcoxon signed-rank test on the resulting 8 within-participant pairs. Because of the counterbalanced design, the two values in a pair come from different task groups; counterbalancing (a Williams square) balances task difficulty across participants rather than within each pair. As a complementary robustness view, we also report a per-task analysis that pairs the median improvement rate of each tool on each of the 8 tasks. If significant, we evaluate the effect size using Cliff's delta [44].

2) Results and Discussion: Figure 5 shows the distribution of the improvement rate for each tool. Blue and red boxes represent the baseline tool and D-Diff, respectively. The left plot shows the overall distribution: the median improvement rate is higher for D-Diff than for the baseline tool. The right plot shows the per-task distribution, where no clear difference between the tools is observed.

This higher median for D-Diff, however, is not statistically significant. The participant-level Wilcoxon signed-rank test yielded a p-value of 0.5469, failing to show a significant difference at the 5% significance level; the complementary per-task analysis agrees ($p = 0.9165$).

Figure 5 shows that the improvement rate is low for tasks requiring in-line separation. Specifically, the improvement rate is low for tasks A3, B3, and B4, all of which require this type of separation. We examined the center source code for these tasks. Direct editing makes in-line separation possible, as described in Section IV-C; some participants used it to separate the changes correctly, while others hardly used it or applied it without achieving a correct separation. The capability alone may therefore not be enough; the difficulty lies less in editing than in noticing that a single line mixes changes from both commits, so support that makes such lines easier to spot could be useful. In addition, for task A3, which had the lowest improvement rate, the participants failed to notice the need for separation; recognizing it required domain knowledge of the code being changed, such as a non-obvious default the Joda-Time library applies for an omitted constructor argument.

> No statistically significant difference was found in the accuracy of the two tools. Additionally, across all tasks, the improvement rate tended to be lower for tasks requiring in-line separation compared to those that did not.

D. RQ_2: Efficiency

1) Study Design: We use the time required to adjust commit boundaries as an evaluation metric for efficiency. D-Diff is timed in the logs from the start button, which immediately displays the three-pane diff, to the end button pressed upon completing the adjustment. The baseline is timed by one of the authors from the first diff display to rebase or stash completion on the screen recording, or from self-report for the participant whose recording was lost. Both intervals thus run from the first display of the diff to the completion of the adjustment in each tool. They are not perfectly symmetric: the baseline includes executing the rebase or stash, whereas D-Diff does not produce actual commits; we believe this discrepancy is small relative to the observed difference in adjustment time. We report no inter-rater reliability; the baseline interval excludes pre-display reading and post-task verification, so imprecision in its endpoints underestimates rather than inflates the baseline times.

In addition, we counted the number of screen switches with the baseline tool from the same screen recordings to investigate the cause of the difference in adjustment time. We defined a switch as a transition of the view displayed in the editor area, e.g., between diff editors and the text editor. This analysis covers the seven participants with an intact recording. We also draw on the participants' free-text questionnaire responses for qualitative insight into the difference in adjustment time.

We evaluate the difference between tools in terms of the adjustment time. As in RQ_1, we treat the participant as the primary unit: for each participant, we take the median of the adjustment times over the 4 tasks performed with each tool and apply the Wilcoxon signed-rank test and Cliff's delta [44] to the 8 within-participant pairs. As a complementary view, we also report the per-task analysis that pairs the median adjustment time of each tool on each task.

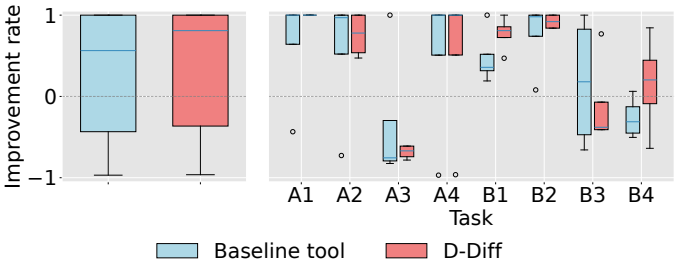


Fig. 5. Improvement rate for each tool (Left: overall, Right: per task).

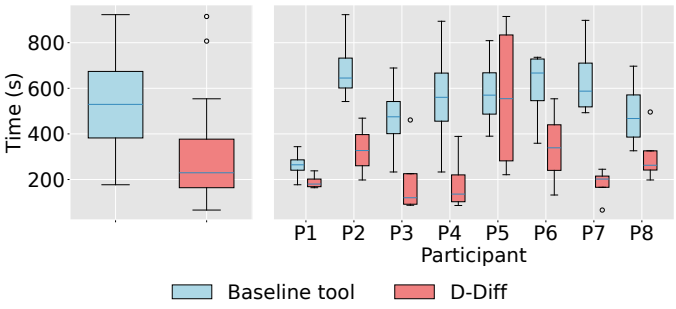


Fig. 6. Time taken by tool (Left: overall, Right: per participant).

2) Results and Discussion: Figure 6 shows the distribution of adjustment time for each tool. Blue and red boxes represent the baseline tool and D-Diff, respectively. The left plot shows the overall distribution of adjustment time. The median adjustment time is shorter for D-Diff than for the baseline tool. The median times for the baseline tool and D-Diff were 530 seconds and 230 seconds, respectively, with D-Diff reducing the time by 300 seconds (57%). The right plot groups the adjustment time by participant, in contrast to the per-task grouping used for accuracy in Fig. 5, because the efficiency advantage is consistent across participants. Every participant was faster overall with D-Diff than with the baseline tool.

D-Diff significantly reduces the adjustment time compared to the baseline. The participant-level Wilcoxon signed-rank test yielded a p-value of 0.0078, indicating a statistically significant difference at the 5% significance level. The complementary per-task analysis agrees ($p = 0.0156$); although the per-task median for task B4 favors the baseline tool, this reversal appears only in the per-task view, since every participant who used D-Diff on B4 was still faster overall. In addition, Cliff's delta is 0.8438, indicating a *large* effect size.

The questionnaire responses suggest that part of the difference in adjustment time is attributable to a difference in cognitive load. D-Diff may have reduced cognitive load by consolidating two diffs into a single screen. Figure 7 shows the distribution of the number of screen switches with the baseline tool for each task. The participants switched screens 18.6 times per task on average, whereas D-Diff has a single screen by design, so switching does not arise. The median number of screen switches ranged from 13 to 25 across the 8 tasks, indicating that frequent screen switching was required in every task rather than only in a few difficult ones. We treat these switch counts

as descriptive corroboration of the cognitive-load hypothesis rather than an independent test, since they come from the same recordings as the baseline times and from the only condition in which switching occurs. One participant noted that *"D-Diff had a simple UI that consolidated necessary information into one screen, making it easy to operate,"* while another said the baseline tool *"required operations across multiple screens, which really trained my working memory [required effort to remember]."* These responses suggest that the baseline tool imposed a high memory load from referencing multiple screens, which D-Diff mitigates.

The two tools also appear to differ in the intuitiveness of their operations: one participant reported that *"In the baseline tool, there were several points where I didn't know how to operate without looking at the slides, which took time,"* whereas another found that *"With D-Diff, I was able to grasp the task to be done and perform corrections intuitively, even for the first time."* These responses suggest that while the baseline tool required trial and error before reaching the desired operation, D-Diff made it easy to identify the next step. These differences in cognitive load and operability likely contributed to the shorter adjustment time with D-Diff. At the same time, because the boundary alteration moves whole lines, the injected errors are largely what D-Diff's move operations address, so part of this efficiency advantage may reflect the task design rather than the tool in general.

> The adjustment time was significantly shorter with D-Diff than with the baseline tool ($p = 0.0078$, large). Furthermore, questionnaire responses suggest that the reduced cognitive load and improved operability of D-Diff contributed to the reduction in adjustment time.

E. RQ_3: Usability

1) Study Design: Usability is evaluated based on Likert-scale questionnaire items covering three aspects. The first aspect assesses the ease of use of each tool with five items: comparing the changes included in the commits, keeping track of the state of the source code being modified, modifying by moving, modifying by editing, and overall ease of use. The second aspect assesses the ease of use of five individual features of D-Diff: the parallel display of the two diffs, the background coloring of the center source code, line-level movement, hunk-level movement, and direct editing. The third aspect directly compares the two tools in terms of perceived task completion time, mental burden, and willingness to use. Free-text fields asked for the reasons behind these ratings. These items are author-designed, and RQ_3 reports the response distributions rather than a confirmatory test, surfacing perceived usability.

2) Results and Discussion: Figures 8a and 8b show the questionnaire results regarding usability for the baseline tool and D-Diff, respectively. The vertical axis represents each question regarding usability, and the horizontal axis represents the number of Likert scale responses. D-Diff has a higher percentage of positive responses for all questions. D-Diff is therefore easier to use than the baseline tool in terms of comparing changes, understanding the source code being modified, and modifying through moving or editing.

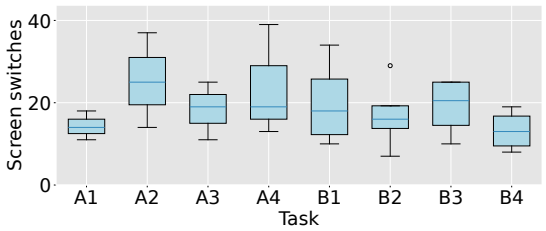
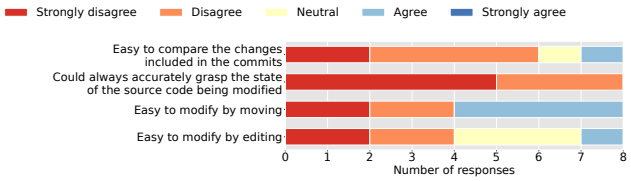
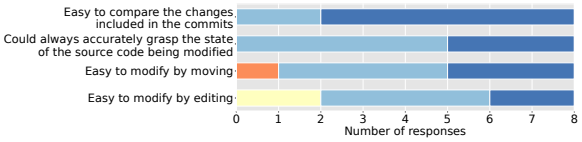


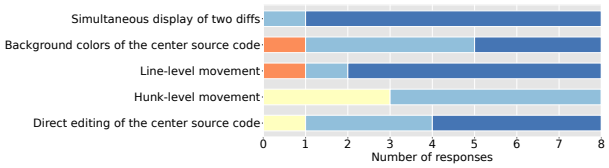
Fig. 7. Number of screen switches per task with the baseline tool.



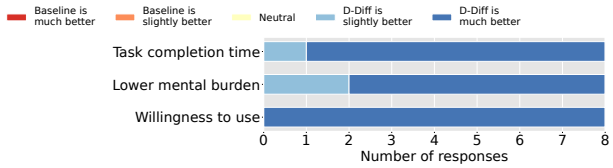
(a) Usability of the baseline tool.



(b) Usability of D-Diff.



(c) Individual features of D-Diff.



(d) Comparison between tools.

Fig. 8. Questionnaire results regarding usability.

Figure 8c shows the questionnaire results regarding the ease of use of each function of D-Diff. The vertical axis represents each function. The percentage of positive responses exceeds half for all features. All participants "agreed" on the simultaneous display of two diffs and line-level movement, while positive responses for hunk-level movement were comparatively fewer. Each feature of D-Diff is therefore generally considered easy to use. The free-text responses indicate that the small scale of the experimental tasks was a reason why hunk-level movement received fewer positive responses, as one participant explained: *"I didn't use it because the diffs in the experiment were small-scale."* They also requested clearer

background colors for deletions and additions.

Figure 8d shows the questionnaire results regarding the comparison of ease of use between tools. The vertical axis represents each question. D-Diff received a high percentage of positive responses across all questions. Participants felt that D-Diff reduced both the perceived task completion time and the mental burden in adjusting commit boundaries, and expressed willingness to use it.

> D-Diff was consistently rated higher in usability than the baseline tool. Participants found it easier to compare and modify changes, and would use it in practice.

F. Threats to Validity

1) Internal Validity: The effect of the 3-way diff display alone may not have been isolated due to the functional differences between D-Diff and the baseline tool. For example, the baseline tool supports resetting commits and resolving conflicts, which D-Diff does not provide, and the two tools require different operations to achieve an adjustment. The observed differences may therefore reflect the integrated environment as a whole rather than the 3-way display alone; RQ_2 thus evaluates that environment against a realistic workflow, not the display in isolation. Comparing D-Diff with a variant that shows the two diffs in separate views would isolate the effect of the display and is left for future work. In addition, the participants may have guessed which tool was proposed, partly because the two tutorials differed in length, and answered the questionnaire in its favor; this demand characteristic most affects RQ_3 and least affects the RQ_2 timing. Moreover, most participants self-reported low proficiency in interactive rebase, so their limited familiarity with the baseline workflow may partly explain the large time difference observed in RQ_2, although the tutorial and practice task for the baseline tool were designed to mitigate this gap.

2) Construct Validity: The improvement rate based on character-level Levenshtein distance may not fully capture the actual accuracy of the adjustment results, as it cannot capture semantic differences. In addition, because the injected errors are line-level, the tasks favor move-based correction, so the efficiency advantage may not extend to adjustments that moving alone cannot perform.

3) Conclusion Validity: The primary statistical tests pair the per-participant median of each tool over the 4 tasks, yielding 8 within-participant pairs, and we report the per-task analysis only as a complementary view. Because each participant uses each tool on a different task group, the two members of a pair are not matched on the same tasks; counterbalancing balances task difficulty across participants rather than within each pair, and this confound cannot be removed analytically. A crossed mixed-effects model that separates participant and task effects would be preferable, but its variance components cannot be reliably estimated with only 8 participants and 8 tasks. The power to detect small effects is therefore limited, and the absence of a significant difference in accuracy does not imply that the two tools are equivalent.

4) External Validity: The number of participants is limited to 8, which may not be representative of all developers. While students majoring in computer science served as participants, prior studies have shown that the performance of students and professional developers is comparable [45], [46]. In addition, further verification is needed to generalize the findings to other languages, since only four Java repositories were used. Moreover, the tasks were created by randomly altering the boundaries of artificially generated TLC pairs; real tangled commits may require more semantic judgment, and the findings may not directly generalize to them. Finally, we argue for the relevance of this adjustment scenario but do not empirically measure how often it arises in practice.

VI. Conclusion

We proposed D-Diff, an interactive diff adjustment environment that integrates the comparison and modification of diffs. D-Diff enables developers to compare the diffs without relying on their memory by visualizing two diffs simultaneously based on a 3-way diff display.

We evaluated its usefulness through a user study with 8 participants. The results showed that D-Diff was significantly superior to the baseline tool in terms of efficiency. The median adjustment time was 230 seconds for D-Diff compared to 530 seconds for the baseline tool, representing a reduction of approximately 57%. D-Diff also received a higher percentage of positive responses across all usability questionnaire items, with participants expressing willingness to use it in practice. No statistically significant difference in accuracy was found between the tools ($p = 0.5469$).

Future work on the tool itself includes comparing related changes across different files, extending to longer commit sequences, integrating D-Diff into IDEs, and supporting fine-grained modifications such as in-line separation. On the evaluation side, the user study could be extended to practitioners. Beyond commit boundary adjustment, the core interaction of directly editing a shared boundary state could also support splitting a single commit, redistributing changes between any two versions, and curating the outputs of automatic untangling.

The tool and evaluation data are available [47].

Acknowledgments

This work was partly supported by JSPS KAKENHI (JP26K02889, JP23K24823, JP26K02888, JP25K03102, JP25H01125, and JP24H00692).

Declaration of AI Usage. During the preparation of this work, the authors used Claude and Nani to improve presentation, including illustration, text readability, and language; they reviewed, edited, and take full responsibility for all content.

References

[1] S. Berczuk and B. Appleton, *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Addison-Wesley Professional, 2002.

[2] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes? An exploratory study in industry," in *Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2012)*, no. 51, 2012, pp. 1–11.

[3] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR 2013)*, 2013, pp. 121–130.

[4] Y. Tao and S. Kim, "Partitioning composite code changes to facilitate code review," in *Proceedings of the 12th IEEE/ACM Working Conference on Mining Software Repositories (MSR 2015)*, 2015, pp. 180–190.

[5] S. Hayashi and M. Saeki, "Recording finer-grained software evolution with IDE: An annotation-based approach," in *Proceedings of the 4th International Joint ERCIM/IWPSE Symposium on Software Evolution (IWPSE-EVOL 2010)*, 2010, pp. 8–12.

[6] M. Fan, W. Zhang, H. Zhao, G. Liang, and Z. Jin, "Detect hidden dependency to untangle commits," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE 2024)*, 2024, pp. 179–190.

[7] B. Hou, X. Tan, K. Zheng, F. Liu, Y. Zhu, and L. Zhang, "LLM-driven collaborative model for untangling commits via explicit and implicit dependency reasoning," *ACM Transactions on Software Engineering and Methodology*, 2026. [Online]. Available: https://doi.org/10.1145/3822177

[8] K. Zhu, Z. Tian, S. Wang, M. Leng, and X. Mao, "Atomizer: An LLM-based collaborative multi-agent framework for intent-driven commit untangling," in *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE 2026)*, 2026, pp. 1431–1443.

[9] S. Sothornprapakorn, S. Hayashi, and M. Saeki, "Visualizing a tangled change for supporting its decomposition and commit construction," in *Proceedings of the 42nd IEEE Annual Computer Software and Applications Conference (COMPSAC 2018)*, 2018, pp. 74–79.

[10] S. Yamashita, S. Hayashi, and M. Saeki, "ChangeBeadsThreader: An interactive environment for tailoring automatically untangled changes," in *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2020)*, 2020, pp. 657–661.

[11] B. Shen, W. Zhang, C. Kästner, H. Zhao, Z. Wei, G. Liang, and Z. Jin, "SmartCommit: A graph-based interactive assistant for activity-oriented commits," in *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*, 2021, pp. 379–390.

[12] "Jujutsu," https://jj-vcs.dev/.

[13] "Meld," https://meldmerge.org/.

[14] "Diffedit3," https://github.com/ilyagr/diffedit3.

[15] D. MacKenzie, P. Eggert, and R. Stallman, "Comparing and merging files," https://www.gnu.org/software/diffutils/manual/diffutils.pdf, 2025.

[16] "Vim," https://www.vim.org/.

[17] "Visual Studio Code," https://code.visualstudio.com/.

[18] "GitLens," https://marketplace.visualstudio.com/items?itemName=eamodio.gitlens.

[19] "GitKraken," https://www.gitkraken.com/.

[20] "GitHub Desktop," https://github.com/apps/desktop.

[21] "Sourcetree," https://www.atlassian.com/software/sourcetree.

[22] "git absorb," https://crates.io/crates/git-absorb.

[23] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri, "Helping developers help themselves: Automatic decomposition of code review changesets," in *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE 2015)*, 2015, pp. 134–144.

[24] W. Muylaert and C. De Roover, "Untangling composite commits using program slicing," in *Proceedings of the 18th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2018)*, 2018, pp. 193–202.

[25] M. Wang, Z. Lin, Y. Zou, and B. Xie, "CoRA: Decomposing and describing tangled code changes for reviewer," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*, 2019, pp. 1050–1061.

[26] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," in *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2015)*, 2015, pp. 341–350.

[27] P.-P. Pârțachi, S. K. Dash, M. Allamanis, and E. T. Barr, "Flexeme: Untangling commits using lexical flows," in *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, 2020, pp. 63–74.

[28] Y. Li, S. Wang, and T. N. Nguyen, "UTANGO: Untangling commits with context-aware, graph-based, code change clustering learning model," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*, 2022, pp. 221–232.

[29] S. Chen, S. Xu, Y. Yao, and F. Xu, "Untangling composite commits by attributed graph clustering," in *Proceedings of the 13th Asia-Pacific Symposium on Internetware (Internetware 2022)*, 2022, pp. 117–126.

[30] B. Guo and M. Song, "Interactively decomposing composite changes to support code review and regression testing," in *Proceedings of the 41st IEEE Annual Computer Software and Applications Conference (COMPSAC 2017)*, vol. 1, 2017, pp. 118–127.

[31] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto, "Hey! Are you committing tangled changes?" in *Proceedings of the 22nd International Conference on Program Comprehension (ICPC 2014)*, 2014, pp. 262–265.

[32] Z. Kurbatova, V. Kovalenko, I. Savu, B. Brockbernd, D. Andreescu, M. Anton, R. Venediktov, E. Tikhomirova, and T. Bryksin, "RefactorInsight: Enhancing IDE representation of changes in Git with refactorings information," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE 2021)*, 2021, pp. 1276–1280.

[33] R. Brito and M. T. Valente, "RAID: Tool support for refactoring-aware code reviews," in *Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension (ICPC 2021)*, 2021, pp. 265–275.

[34] L. Chen, M. Lanza, and S. Hayashi, "ChangePrism: Visualizing the essence of code changes," in *Proceedings of the 13th IEEE Working Conference on Software Visualization (VISSOFT 2025)*, 2025, pp. 69–73.

[35] L. Gasparini, E. Fregnan, L. Braz, T. Baum, and A. Bacchelli, "ChangeViz: Enhancing the GitHub pull request interface with method call information," in *Proceedings of the 9th Working Conference on Software Visualization (VISSOFT 2021)*, 2021, pp. 115–119.

[36] R. Escobar, J. P. S. Alcocer, H. Tarner, F. Beck, and A. Bergel, "Spike – a code editor plugin highlighting fine-grained changes," in *Proceedings of the 10th Working Conference on Software Visualization (VISSOFT 2022)*, 2022, pp. 167–171.

[37] V. Frick, C. Wedenig, and M. Pinzger, "DiffViz: A diff algorithm independent visualization tool for edit scripts," in *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution (ICSME 2018)*, 2018, pp. 705–709.

[38] E. W. Myers, "An O(ND) difference algorithm and its variations," *Algorithmica*, vol. 1, pp. 251–266, 1986.

[39] "jsdiff," https://github.com/kpdecker/jsdiff.

[40] "diff-match-patch," https://github.com/google/diff-match-patch.

[41] "Monaco Editor," https://microsoft.github.io/monaco-editor/.

[42] K. Herzig, S. Just, and A. Zeller, "The impact of tangled code changes on defect prediction models," *Empirical Software Engineering*, vol. 21, no. 2, pp. 303–336, 2016.

[43] E. J. Williams, "Experimental designs balanced for the estimation of residual effects of treatments," *Australian Journal of Scientific Research Series A: Physical Sciences*, vol. 2, no. 2, pp. 149–168, 1949.

[44] 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's d for evaluating group differences on the NSSE and other surveys," in *Annual meeting of the Florida Association of Institutional Research*, vol. 177, no. 34, 2006, pp. 1–3.

[45] I. Salman, A. T. Misirli, and N. Juristo, "Are students representatives of professionals in software engineering experiments?" in *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE 2015)*, 2015, pp. 666–676.

[46] D. Falessi, N. Juristo, C. Wohlin, B. Turhan, J. Münch, A. Jedlitschka, and M. Oivo, "Empirical software engineering experts on the use of students and professionals in experiments," *Empirical Software Engineering*, vol. 23, no. 1, pp. 452–489, 2018.

[47] D. Muto, T. Ueno, and S. Hayashi, "Artifact of "D-Diff: An interactive environment for adjusting commit boundaries based on editable 3-way diff"," figshare, 2026. [Online]. Available: https://doi.org/10.6084/m9.figshare.32660529