"What Led to this Change?" Organizing Code Diffs Using Semantic Information

Noam Har-Tuv
Technion, Israel
bitroix1@gmail.com

Cynthia Richey
University of Pennsylvania, US
lapwing@engineering.upenn.edu

Chandrakana Nandi
Certora Inc., US
chandra@certora.com

Hila Peleg
Technion, Israel
hilap@technion.ac.il

***bstract*—Inspecting code changes in source-control commits is an important task for developers. Professional best practice is to maintain a "compiling-commit" policy, where each commit is buildable. In practice, a single change can introduce cascading changes that must be committed together to keep the project compiling. Such commits are unwieldy to navigate and labor-intensive to understand, especially in the commonly used *line diff* tools that present them as deletions from the old version and additions to the new version. We present Diffmagic, a new algorithm and interface that leverages the compiling-commit policy to organize diffs according to causality. Since commits must remain buildable, changes to a *definition* lead to changes to its *uses*, forming a causal structure that Diffmagic uses to find *change batches*, i.e., changes with a shared root cause. Diffmagic uses semantic information from running just the compiler's front-end, on only the changed files (i.e., it requires neither full compilation nor whole-project analysis) to recover these *use-definition* connections that imply the causality. Within each change batch, Diffmagic assigns a *causality score* to prioritize changes that are likely to trigger other downstream changes.**
user study with 14 participants performing two tasks on real-world commits suggests that Diffmagic makes diffs easier to understand and helps users complete diff inspection tasks faster.

I. Introduction

Inspecting code changes is an important and time-consuming task for developers. This can be in the course of a structured code review activity [47], for finding the cause of a fault, or for understanding the contents of a feature [52]. Code changes are often viewed in a line difference (or a *diff*) format [11], where changes are displayed as deletions from the old version of the file and additions to the new version. djacent changed lines form a *hunk*, and a handful of surrounding unchanged lines are typically shown as context. Some graphical diff viewers like the desktop application Beyond Compare [48] and GitHub's diff viewers [12, 13] allow users to explore the full file and search through changes.

However, despite many small usability changes that have become standard over the years (e.g., syntax highlighting of code in the diff, highlighting the changed characters within a changed line) and better ways to compute the changes from the raw files [19, 21, 34, 45], diffs are still essentially presented as pairs of changed lines with some context, ordered lexicographically by the path of the file they appear in.

While this can work for small changes, line diffs become cumbersome for large changes. This is further exacerbated by the common industry practice of making only "compiling commits" [35], that is, commits in which the whole project remains in a compiling state to help maintainers deal with problems down the line. Changes to definitions must be propagated throughout the codebase to their usages before the code can compile and a commit can be created. Renaming a variable or method causes all of its uses to change, changing the signature (i.e., number or type of arguments) of a function affects all its calls, and deleting a class necessitates removing all its uses. Changes like these can have far-reaching effects that must all be included in a single (often large) commit.

Moreover, in such commits, the hunk containing the *cause* of a change may be located far from the hunks of its effects. Depending on file names in the project, it may even appear *after* those effects, or be interleaved among them. Even within the same file, the effects of a change can come before the cause: in languages like Java, class members can be declared in any order in the class scope, so changes to uses of methods or fields may appear before the declaration change that necessitated them. The *causal structure* of the commit is therefore hidden by the order in which hunks in the diff are displayed, making diffs hard to understand.

Our insight is that the "compiling-commit" policy contains within it the key to the remedy: in such commits, *definitions* and their *uses* form a relationship of *causality*, which can serve as an organizing principle for the commit, grouping changes with the same root cause. These relationships can be discovered using lightweight semantic information about the program, namely *name resolution*, which identifies the declaration of each identifier in a program. Using name resolution, we can connect a function call to the declaration of the called function, function arguments to their corresponding parameters, and a method to the interface method it overrides. In the context of a change, such connections can be used to relate changes that would otherwise be separated by many lines or even be in different files. We refer to these groupings as *change batches* to be displayed separately to the user. We perform name resolution by running a compiler front-end on the code before and after a change.

Since name resolution is directional, i.e., from uses to their definitions, we can use it to identify which change is the cause and which is the effect. To identify the cause of many changes that are grouped together by this technique, and to generate display names for the change batches according to their cause, we score each change in a batch using reachability via name-resolution paths.

Fig. 1: Diffmagic displaying a diff between two versions of a Java program. ◯ Diffmagic breaks the changes in the diff down into *change batches* categorized by type, e.g., ① renames of methods and variables. ② Individual change batches can be selected and be viewed separately. Ⓑ Diffmagic displays the selected batch in its center panel display. Change ③ is a result of change ④, but comes before it in the original file. Ⓒ The list of changes in a batch lets the user navigate between them. The list is sorted by causality of the changes, with ⑤ the likely source of the batch pointing to ④.

Importantly, this approach does not require the entire project to be compiled: if only a handful of files out of a very large project change, performing name resolution only on the changed files is sufficient. Relying on the assumption that both the version before and after the change compile, the change will contain both its cause and its effects, so program elements used in effect hunks of the diff will have their definitions to resolve to. Definitions declared outside of the changed files may not successfully resolve, but since they were not changed, we do not need to find a change batch for them.

To help users better understand large diffs, we implemented Diffmagic[1], a rich diff viewer that batches together related changes and their causes in Java programs. Diffmagic uses structural diffing [34] and name resolution to batch together related diff hunks, then computes a *causality score* for each hunk in the diff, so that hunks can be sorted by their distance from the source of the change. Diffmagic's interface is based on the GitHub diff viewer [12], retaining its familiar elements. Figure 1 shows a diff in Diffmagic with all its components.

To evaluate Diffmagic, we conducted a between-subjects user study with 14 participants and two diff comprehension tasks based on real-world commits from open-source pache Java projects. We compared Diffmagic against the popular GitHub Desktop tool as a baseline. Our results show that Diffmagic improves users' ability to understand the goal of a commit and helps users accomplish their tasks faster with fewer users timing out. Users also said that Diffmagic's organization of changes helped them understand the root cause, and the order in which changes were displayed was logical and made it easier to connect related changes.

In all, this paper makes the following key contributions:

- We present our key insight that the causal structure of a diff can be reconstructed using semantic information (name resolution) from the compiler front-end.
- Based on this insight, we present a new algorithm for organizing diffs using structural diffing and name resolution to batch changes according to a causal relationship.
- We present an implementation of this algorithm in a tool called Diffmagic that shows organized diffs in a new interface built on top of the GitHub Desktop tool.
- We provide evidence from a user study that Diffmagic's new way of showing diffs makes them easier to understand, and faster to inspect.

[1]Our implementation is available at https://github.com/Diffmagic/Diffmagic. demo video is available at https://youtu.be/bro4TqO88xY.

II. B CKGROUND ND REL TED WORK

This paper presents a new way of showing code diffs. To that end, we first introduce key terminology on program diffing that will be used in the rest of the paper. We then position our work at the intersection of (i) knowledge about how developers understand and review code changes, (ii) diffs that carry additional information, and (iii) usability improvements in diff visualization.

. *Terminology*

We begin by introducing terminology essential to the inner working of Diffmagic.

Diffs and diff hunks. The diff between two files is computed by first computing what has *not* changed, or *aligning* the files according to their unchanged elements. The parts that cannot be aligned are then displayed as the diff's *hunks*: contiguous regions of the file that have changed. In a compressed display of a diff [11, 13], hunks are displayed with a small context of unchanged code around them. When two hunks are separated by fewer unchanged lines than the size of the context window, they are merged and displayed together. Some visual diff viewers [48] show the entire file with the hunks highlighted.

Traditional *line diff* algorithms compute hunks by aligning unchanged *lines* in the file, marking the remaining lines as changed. Since these algorithms are designed for any textual file, they have several shortcomings when used for code. They have no understanding of program structure, so they can get caught up on uninteresting formatting changes. dditionally, large changes are often misinterpreted because the alignment is so fragile.

On the other hand, *structural diff* algorithms operate on two bstract Syntax Trees (STs) generated from parsing the code files. Instead of aligning pairs of unchanged lines, they align pairs of ST nodes, one from each ST. Since the resulting hunks are trees themselves, structural diffs provide a finer-grained view of changes, are not sensitive to changes in indentation and formatting, and often better reflect the programmer's intent than purely line-based matching. GumTree [18, 19, 43] and ChangeDistiller [21, 22] are considered the canonical ST diffing algorithms, supplemented by different algorithms exploring optimizations for specific types of code changes [2, 3, 8, 14, 39, 40, 56].

more recent structural diff tool, Difftastic [34], models the diff space as a graph and finds the lowest-cost route from the initial state to the final state of the diff. Diffmagic uses Difftastic to obtain a structural diff.

Static name resolution. The *name resolution* phase of compilation is where all uses of identifiers (e.g., variable, function, or class name) are associated with their definitions in the program [4, Ch. 5]. This information is then used throughout the rest of the compilation pipeline e.g., to check type correctness or to allocate room on the stack for variables. IDE toolkits like Eclipse Java Development Toolkit [23] can perform name resolution on partially compiling code, and, importantly in Diffmagic's case, when files are missing.

B. Understanding Code Changes in Practice

Understanding code is a crucial part of software development. 2022 study found that developers spend 41 minutes per day on code reviewing and understanding code changes, which is almost as much time as they spend writing code (52 minutes per day) [50]. The rise of I-assisted coding may further increase the importance of code understanding and review, as developers increasingly rely on I tools yet continue to have limited trust in I-generated code [28, 51]. To understand how professional programmers understand code changes, researchers conducted a large-scale user study of software engineers at Microsoft [52]. They studied scenarios in which developers need to understand a code change, the frequency of these scenarios, and the kind of information developers look for when trying to understand a change. The authors note that many study participants struggled to understand the effect of a change and found it difficult to associate changes across different parts of the codebase. This motivates Diffmagic, and its goal to aid comprehension beyond line-level edits by grouping together related changes.

C. Tools for Rich Diff Generation

Prior work has produced many tools for generating richer code diffs, annotating diffs with additional information, and classifying types of changes.

DiffDetective [5] makes existing diff tools "variability-aware", i.e., it distinguishes changes to a program's actual logic from changes to annotations like those used for conditional compilation. Likewise, refactoring-aware program diffing [2, 6, 37] aims to remove from the diff changes that are *semantics-preserving*, i.e., do not change the program's meaning. Ge et al. [25] isolate these changes and display them separately. Diffmagic's identification of renames parallels these tools, but it both supports non-refactoring changes, and explicitly visualizes renames instead of hiding them.

Collector-Sahab [17] enhances existing Java code diffs with dynamic information, running test cases on both the original and modified versions of the code to generate a runtime diff of the program's state. Dufloer et al. [15] display this information in a debugger to track divergence in executions caused by a change. SymDiff [38] and Glock [26] check the behavior differences symbolically via a logic encoding and generate *counterexamples* highlighting differences. Unlike these, Diffmagic does not reason about the program's behavior, it focuses on organizing source-level diffs.

Many tools summarize the changes in a diff and produce summary statistics [27, 29, 44, 46] using semantic information such as type information as well as name resolution similarly to Diffmagic. Wang et al. [53] use information retrieval methods for the same goal. Diffmagic uses the semantic information to partition the diff, in order to *show* it, not *summarize* it, making Diffmagic complementary to these tools.

CLDiff [32]'s goals are similar to ours: like Diffmagic, it improves diff understandability by computing a syntactic diff, then grouping related changes. However, unlike Diffmagic, CLDiff relies on string-matching heuristics to connect diff hunks where the same names appear, and uses a fixed set of five predefined change links. Diffmagic, in contrast, derives the grouping from name resolution information and organizes them into causally motivated batches rooted at definition changes. ll but one of the change links in CLDiff are captured by Diffmagic's semantic structure, but Diffmagic does not depend on brittle string-based matching. Kim et al. [36]'s goals are also conceptually close to Diffmagic's in that both tools aim to show high-level structure in diffs. While Diffmagic batches changes through causality, Kim et al. [36] produce concise summaries of code changes as "change rules" by using a rule-inference algorithm to search the space of instantiated candidate rules.

D. Diff Visualization

Finally, we discuss prior work that focuses on novel ways to visualize diffs.

DiffViz [24] is a diff visualization tool that offers features for better code navigation. Its navigation does not rely on semantics, only on displaying the syntactic edit script created by some syntactic diff algorithms [13, 19].

zurite [55] displays a history of changes on a timeline, showing the individual diffs as a function of time. Chronos [49]'s visualization is also timeline-based: it lets developers select arbitrary lines of code across one or more files and then visualizes the complete history of just those lines. The goal is to support interactive querying and exploration of code evolution. Storyteller [42] takes a unique approach to helping developers understand the evolution of code. It records and replays code changes across programming sessions, allowing

developers to annotate and share the evolution of their code as web pages. While these tools focus on displaying a sequence of changes, Diffmagic focuses on making a single diff with many related changes easier to understand.

Deknop et al. [9] visualize the difference in an application's log printouts, an implicitly-structured format, and visualize the impact of code changes as a graph. Diffmagic's focus is entirely on showing syntactic changes to code and it does not require running the code at all.

III. Motiv ting Ex mple

We follow Kiran, a Java programmer, as they examine the diff of a commit[2] in the pache Kafka project, a distributed message streaming platform. The commit that Kiran is inspecting comprises two Java files, `Mirror oordinator.java` and `MirrorMetadataManager.java`. It also includes several atomic changes: several methods are renamed, as are some local variables, and the `MirrorMetadataManager` class deletes two helpers, a logger and a scheduler, as well as their initialization and usages. In addition, there are non-functional changes in the form of whitespace and comment styles.

Tracking exactly what happens in the commit using a line diff viewer requires Kiran to constantly scan up and down the files to track the cause of the various changes. For example, the 22nd diff hunk in the `MirrorMetadataManager.java` file replaces the call to `updateRemote lusterNodes` with a call to `updateRemote lusterNodes ache`. On its own, this change could suggest that the call was redirected to a different method. Only after scrolling past several more hunks can Kiran see that the change actually stems from a method rename. Likewise, the deleted import for the `LogManager` type is separated by several hunks from the removal of the field of this type, even though the two changes must be considered together to understand that the change is not replacing the field's type or switching to a different library.

Instead, Kiran opens the diff in Diffmagic, shown in Figure 1, to observe the diff in a way that conveys the causality between its individual changes.

◯ **List of change batches.** Kiran starts in the left panel, where related changes are grouped into *change batches*. By default, the option Show ll is selected, which shows the raw line diff in the center panel. Batches are ① grouped into Deletions, dditions, and Renames. n additional Miscellaneous category contains diff hunks that are not part of any batch.

Kiran inspects the list of Renames in the diff, then selects ② "Method renamed from `updateRemote lusterNodes` to `updateRemote lusterNodes ache` in `MirrorMetadataManager`".

Ⓑ **Viewing diff hunks in Diffmagic.** Once Kiran selects the method rename change batch, the center panel changes from showing the full diff to showing only the changes in this batch.

Diffmagic shows Kiran all the lines where individual changes in the batch occur, leaving a *context window* around them. Other changed lines are hidden. Kiran sees that the current batch contains two changes: ④ the method rename and ③ its use.

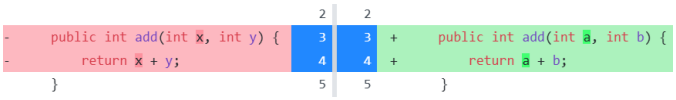

Fig. 2: diff with two variables renamed. Only one change on each line is included in the change batch.

Ⓒ **Navigating the changes in a batch.** Kiran can see every change in the current batch in Diffmagic's right panel. There are four changes: two deletions and two additions. While the diff hunks in the center are sorted as they would be in most diff viewers, i.e., hunks by their location in the file and files by their path in the project, the list of changes on the right panel is instead sorted by the *causality score*, the number of changes the current change is the transitive cause of. Kiran notices that in this batch, later changes in the file are listed first. Kiran clicks on ⑤ "likely source", the highest-scoring change in the batch which also gives the batch its name. Clicking the likely source highlights that change in the center panel, ④ the rename of the method from `updateRemote lusterNodes` to `updateRemote lusterNodes ache`.

Kiran then clicks on other changes in the right panel to look through them. The third change sends Kiran to the center panel, highlighting ③ the use of the function. If the change is outside the text shown in the center panel, this also scrolls to it. By following changes in descending order of causality score, Kiran can follow the cause to its cascading effects, then move on to the next batch until they've fully inspected the diff.

Separating nearby changes. When selecting a specific batch, Diffmagic narrows the hunks shown in the center panel to that batch. But Diffmagic's hunks are also displayed with a context window around them, and a change in the batch can contain changes from other batches or Miscellaneous changes, e.g., the comment above ④.

Diffmagic displays lines that are changed but are not included in the currently displayed change batch in a faint red or green. That way Kiran can tell which changes are in the selected batch while still retaining a context window. This is particularly important if Kiran uses the expander arrows between hunks to enlarge the context window to encompass other changes.

If code changes that belong to different change batches appear on the same line, Diffmagic uses a brighter color to highlight which parts of the line are contained in the current batch. n example of this is shown in Figure 2: the code in view contains changes to both parameters of the function, but only the first is in the currently viewed batch. The second parameter, which is in a different batch, is not highlighted.

Overall, Diffmagic allows Kiran to quickly understand the gist of the changes in the inspected diff, and to easily find the changes they are interested in.

IV. Org nizing Diffs

We implemented the front-end for Diffmagic on top of GitHub Desktop. To display a diff, Diffmagic takes as input

[2]https://github.com/showuon/kafka/commit/32674acf4680feb0c4942141a983d46b1f470acb

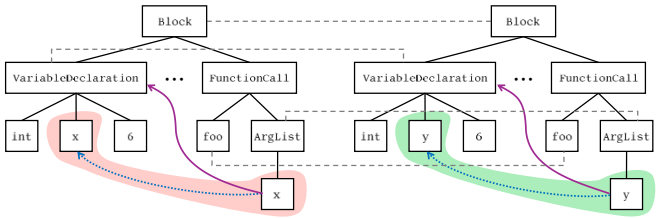

Fig. 3: (Left) ST for the Java snippet `int x = 6; foo(x);` annotated with name resolution edges connecting the identifier `x` at the call site to its declaration. (Right) Similar ST annotated with name resolution edges but for the snippet `int y = 6; foo(y);` showing a variable rename. The dashed lines between the two STs are examples of the alignments between unchanged ST nodes in the programs before and after the change.

all files that are changed in the diff in their versions before and after the change. We will refer to these as the *lhs* (left-hand side) and *rhs* (right-hand side) versions of the files, based on the side of the diff in which they are displayed. In this section, we detail how Diffmagic (1) applies name resolution to the changes, (2) computes the change batches, and (3) selects a likely source, so that they can be displayed by Diffmagic's UI. Our implementation organizes diffs of Java code, but its base ideas are language-agnostic, so we conclude this section by outlining the requirements for Diffmagic to handle additional programming languages.

. *pplying Name Resolution*

In this section, we detail how Diffmagic overlays the program STs for the lhs and rhs with *name resolution edges* that indicate causality between different changes. We call this overlay a *causality graph*.

s a first step, Diffmagic applies structural diffing to the two versions using Difftastic's [34] structural differencing implementation. Difftastic first aligns unchanged ST nodes—a sample of these is shown as dashed gray lines in Figure 3. It then labels unaligned nodes (e.g., `x` in the lhs of Figure 3 and `y` in the rhs) as *novel* nodes, which we refer to as being *in the diff*.

We now consider the lhs and rhs trees separately. For each version's ST, Diffmagic identifies all nodes that describe a *name*. These can be names of variables (e.g., `x` in the lhs of Figure 3), of functions (e.g., `foo` on both sides of Figure 3), and of non-primitive types (e.g., `String`).

For each name node that was marked as novel by the structural diff, Diffmagic applies name resolution to recover the edges that connect the name's use to the program element that declares it. Diffmagic uses the Eclipse Java Development Tools (JDT) [23] toolkit for name resolution. s Diffmagic only gives the JDT the changed files, compilation errors can occur, for example when class definitions remain unchanged and are therefore absent from the changed files. However, Eclipse JDT's error recovery means name resolution remains effective even when such errors are present. s long as both a definition and its uses are within the changed files, Eclipse JDT recovers and produces the relevant name resolution information successfully.

The name `x` in the lhs of Figure 3 will be resolved by the Eclipse JDT to the `VariableDeclaration` node that defines the variable `x`, indicated in Figure 3 by a purple arrow. However, this name resolution edge is not going to help Diffmagic: while its source is in the diff, its target, the `VariableDeclaration`, is not, so we cannot consider the causality it represents as part of the change. We are interested in edges *between two novel nodes*: such an edge indicates that its source (the use) was changed to accommodate a change in its target (the declaration).

Since Figure 3 shows a rename, only the name of the declared variable is novel. To still capture this causality, Diffmagic supplements the name resolution edges with several additional *parallel edges*. These are edges that preserve the same use–definition relationship as the original edge, but start from or target a different ST subnode. In Figure 3, Diffmagic adds the blue, dotted parallel edges from the name being resolved to the name within the resolved declaration, i.e. between the two `x` nodes and between the two `y` nodes. Other examples of parallel edges in Diffmagic are:

- from arguments of a method call to the parameters of the method, e.g., from `x` to the declaration of the parameter inside `foo`'s `MethodDeclaration`;
- from inherited methods to their version in a parent class or interface, paralleling the resolution of the name following `implements` or `extends`;
- from the method name of a constructor to the name of the class it constructs.

Notice that, unlike standard name resolution, which resolves names to a single declaration, parallel edges can connect a node to multiple relevant nodes in the ST.

Diffmagic builds the *causality graph* by iterating over all novel name nodes and collecting their name resolution edges and parallel edges. It then retains only those edges whose targets are also novel. The resulting set of novel nodes and set of connecting edges give us the *causality graph* for that version of the program. Diffmagic constructs one causality graph for the lhs and one for the rhs.

B. Batching Changes

This section describes how Diffmagic uses causality graphs to compute the *change batches* displayed to the user.

Diffmagic begins by finding the connected components of the lhs and rhs causality graphs separately, using a BFS-based algorithm [30]. Each connected component groups novel nodes on one side of the diff that are linked by causality and are therefore considered by Diffmagic as effects of the same underlying change. In Figure 3, this yields the red connected component on the left-hand side, and the green connected component on the right-hand side.

t this point, Diffmagic has only grouped related changes *within* each version. We do not want to separate the rename in Figure 3 into two batches, a deletion and a separate addition, so we must determine whether an lhs connected component and an rhs connected component are two independent batches or two sides of a single rename/substitution.

To understand how Diffmagic unifies them, we observe that novel nodes in the structural diff are not connected to each

other, whereas unchanged nodes are. Diffmagic therefore looks for pairs of novel nodes that are children of aligned unchanged nodes. For example, in Figure 3, the `VariableDeclaration` nodes are aligned. This is an indicator that the connected components of `x` and `y` below them need to be unified and considered together as one batch.

Once all pairings have been made, Diffmagic determines the category of the batch by examining nodes in the batch with an outgoing degree of zero (i.e., "sinks" into which causality flows). If the sinks on the rhs do not have a pairing, the change batch is rooted in an *addition*, if those on the lhs have no pairing, the batch is rooted in a *deletion*, and if both are matched it is a *rename/substitution*.

C. Computing Causality Scores

Once the changes are divided into batches, Diffmagic finds the *cause* of the change in each batch. This allows Diffmagic to both sort the right panel by logical distance from the root cause and to create the batch's *display name*.

For every ST node in a change batch, Diffmagic computes the *causality score* by counting the number of nodes that it is reachable from in the causality graph. For instance, the unified connected components of the change batch in Figure 3 have four nodes and two edges. Each of the nodes that is the target of an edge has a causality score of 1. The two nodes that are the source of the edges have a causality score of . The right panel is then sorted by the causality score and the top-most change is marked as the likely source. If there is a tie, as in the case of Figure 3, the pairings computed while building the batch are used to check whether the tie is between a pair, i.e., between the two halves of a rename, and, if it is, then one of them is arbitrarily chosen as the source. If there is a tie between more than two changes, no likely source is denoted.

Finally, we use the likely source to create a display name for the batch. The change in Figure 3 is identified as a rename, and its tied top-scoring nodes are `x` and `y`, so the naming template "Variable renamed from to " is used. If there is no likely source, we compose the name from all tied top-scoring nodes.

D. Diffmagic for Other Languages

Diffmagic is implemented for diffs of Java programs, but its technique is language-agnostic. In this section, we detail the requirements for Diffmagic to support other languages.

Most crucially, the idea behind Diffmagic is aimed at statically typed languages or languages where there is a concrete definition site for variables and functions (e.g., C#, C++, Rust, Scala, Haskell, Ocaml, etc.). For example, in Python, where a variable does not have a single definition, Diffmagic's technique will not be able to handle variables.

Next, an implementation of Diffmagic requires a compiler front-end that resolves names when files are missing. The Eclipse JDT is an IDE toolkit, designed to deal with projects in an unstable state; other languages have similar toolkits [1, 41].

Finally, the parallel edges needed to supplement the name resolution edges may vary by programming paradigm and the structure of the ST for a given language.

T BLE I: Statistics about the commits used in our study. Java files in project are counted for the committed version. Line diff changes are the per-line statistics computed by `git`.

Task	Project	Java files in project	Changed files in diff	Line diff changes
Tutorial[1]	pache Kafka	5,584	2	+121 -95
Task 1[2]	pache Maven SCM	449	95	+307 -336
Task 2[3]	pache Flink	11,007	6	+93 -15

[2]https://github.com/apache/maven-scm/commit/d6391e43b7ea7e51ab50a1a71fc8c325e32b79ad
[3]https://github.com/apache/flink/commit/e2e10a062a779aabc7717c643623c74ff4cbd5cb

V. User Study

To evaluate Diffmagic, we compared it in a user study with GitHub Desktop, the viewer on which our tool is based. Viewing changes through GitHub is considered the industry standard for viewing diffs, and was reported by 9 out of the 14 participants as the way they usually look at diffs. We asked the following research questions:

- RQ1. Does Diffmagic help programmers get a better understanding of a change?
- RQ2. Is it faster to use Diffmagic for inspecting diffs than a baseline line diff viewer?
- RQ3. What role do change batches play when inspecting a diff?

This research was performed under the oversight of the institutional review board at Technion, approval number 2026-015.

. Participants

We recruited 14 participants, industry developers with between 2 and 29 years of Java experience (mean = 9, stdev = 8.83), who perform code reviews at work (all: at least weekly, 13/14: multiple times per week), ensuring they have experience inspecting commits made by others. We randomly assigned seven to the control condition, viewing diffs through a baseline tool, and seven to use Diffmagic. Participants were compensated for their time with a $3 gift card upon completion.

B. Tasks

For our study tasks, we selected two commits made in pache projects between July and December of 2025 that were sufficiently interesting for diff comprehension tasks but not so large as to be impractical for a 2 -minute task. For the tutorial, we used the commit shown in Section III. Statistics for the commits and their projects are shown in Table I.

Participants were only given the diff, without the rest of the project or the commit message, so that their comprehension of the diff would rely solely on the quality of code differencing. For each of the tasks, participants were given a short prompt with a reason to inspect the commit in depth, then asked comprehension questions about the content of the diff.

Task 1: pache Maven SCM. The Maven SCM library integrates source control systems (e.g., git) into the build process. The commit used in the task is a refactoring that replaces a base class with a generic class. This changes the return types of methods for all inheritors from a base type to a concrete type

instantiated in the generic. This change is widely propagated to implementing classes and their uses, comprising 95 files.
Task type: code review.
Prompt: Participants were asked whether to merge the commit, as it could introduce a *backward-compatibility issue*.
Comprehension questions:

1) *What was the goal of the commit?*
2) *What was the main change you saw in the code?*
3) *What design idea is the change introducing?*

Task 2: pache Flink. Flink is a distributed data processing framework for large-scale data in real time. The commit used in the task adds a control flag, `ignoreParseError`, reads it, and propagates its logic throughout the code.
Task type: debugging/fault localization.
Prompt: Participants were asked to judge whether the commit is the cause of a `NullPointerException`, as `git bisect` hinted it may be the cause.
Comprehension questions:

1) *What was the goal of the commit?*
2) *What was the main change you saw in the code?*
3) *What is the role of the flag that was added to the system?*

C. Procedure

We conducted the study via Zoom. Participants were randomly assigned to either use Diffmagic, or the baseline tool, GitHub Desktop. Participants were first given a brief tutorial of their assigned tool, then given the two tasks in Section V-B. Participants were instructed to do what they usually do when inspecting diffs, e.g., searching the web; they were encouraged to think out loud, but were not prompted if they chose not to. Each task ended when the participant came to a conclusion about the provided prompt, or at a timeout of 2 minutes. Time to completion for each task was recorded. fter each task, participants were asked the task-specific comprehension questions. nswers to the comprehension questions were scored in bulk by the last author, who did not know the assigned group for each answer. Scores assigned for each question were 1 (correct), .5 (correct but incomplete), or (incorrect), yielding a total comprehension score between and 3. fter both tasks, participants filled out a post-study survey with five Likert-scale questions. Finally, participants in the Diffmagic group who were interested were given additional information about how the tool works, and GitHub participants who were interested were shown Diffmagic or had it explained to them, as the remaining time permitted.

VI. Results

We recorded the task outcomes for our study participants, shown in Figure 4. Responses to the post-study survey are shown in Figure 5. We tested statistical significance using a Wilcoxon rank-sum test [54], which is appropriate for interval data (correctness scores) and ordinal data (survey results). The threshold for statistical significance was set at $p <$ .1. We also tested effect size using Cohen's d [10]. The first and last authors individually coded participant actions and quotes during sessions, then collaboratively grouped these codes. We report on the observed behavioral patterns.

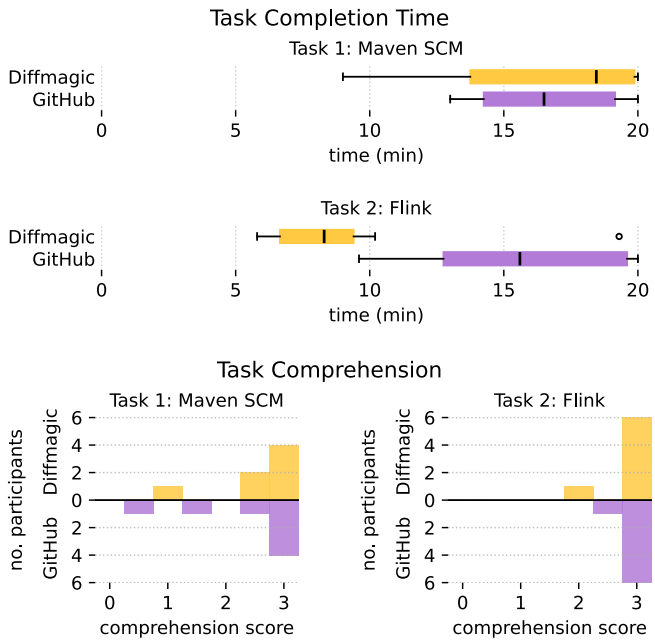

Fig. 4: Task completion times (timeout at 2 minutes) and comprehension scores (range 0–3) for tasks 1 and 2.

. RQ1: Comprehension of the Diff

Task 1: Maven SCM. Task 1 was to review a change spanning 95 files. In the post-task comprehension survey, Diffmagic users scored higher, with an average score of 2.6 (stdev = .73) to GitHub users' average score of 2.3 (stdev = .99). These results are not statistically significant ($p =$ .84, $d =$ .25).

While inspecting the diff, every participant in the GitHub group except P7 made mention of the change being repetitive, and of the similar cascading changes being "more of the same" (P13). P11 described it as "a pretty targeted formulaic change" with "a Cartesian product of annoyances going on". P9 said that what they are interested in are changes "not following the pattern of the previous classes". In contrast, in the Diffmagic group, only P5 made reference to the repetition of the changes, saying of the many classes changing the way they implement a modified abstract class, "these are all similar".
Task 2: Flink. ll participants but two got the maximum score for comprehension after inspecting Task 2: one Diffmagic user scored .5 (correct but incomplete) on two questions, and one GitHub user scored .5 on one question, which, as expected, is not statistically significant ($p =$ .96, $d =$ .24).

In this smaller task, GitHub users were more explicit about the friction caused by the ordering of files by path: "the entry point was towards the bottom of the file list" (P1), and "the actual change [...] was in the fourth file" (P8). Diffmagic users, on the other hand, moved between Show ll and specific batches to gain a quicker understanding of the diff (P3, P4, P5).
Post-study survey. GitHub users felt it was less easy to understand the cause of changes (Q1) ($p =$ .38, $d =$ .62) and more difficult to locate the interesting part of the diff (Q3)

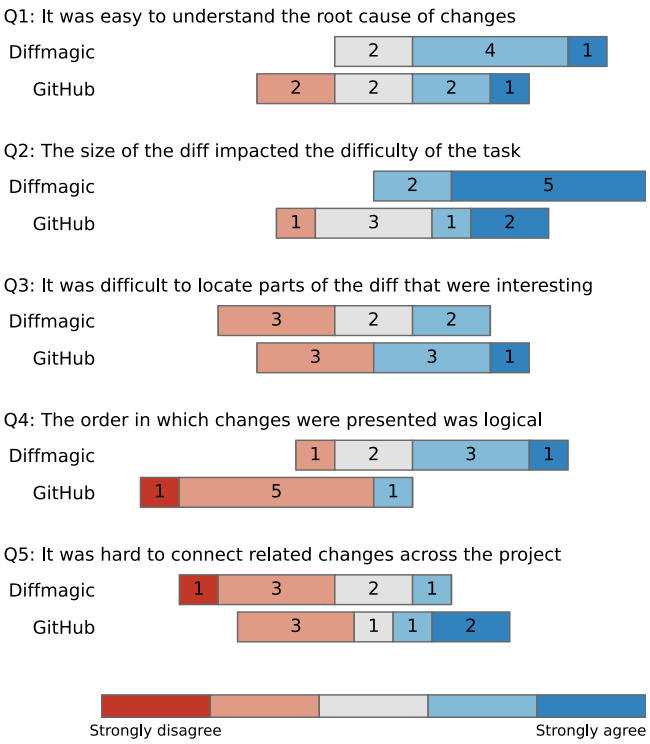

Fig. 5: Participants' ratings in post-study survey. For Q1 and Q4 higher is better, for Q2, Q3 and Q5, lower is better.

($p=$ .6, $d=$ .39). P8 and P11 mentioned that they would like "semantic changes" that are based on the goal and on "why are we changing to this". P6, a Diffmagic user, said, "this list of changes that are related was super helpful for figuring out what it was doing in context". P10 noted that Diffmagic aligned with the way they think about diffs: "somebody touches something, I'm thinking about many of the places where it's used".

In Q4, GitHub users almost uniformly disagreed with "the order in which changes were presented was logical", while Diffmagic users were significantly more positive ($p=$. 26, $d=1.5$). P1, P2, P8, P11, and P13 all made reference to the alphabetical ordering of file names being unhelpful. P1 also noted that at work, developers overcome this shortcoming by adjusting the way file paths and methods are ordered so that a diff would "go from a higher to a lower level of abstraction". P11 mentioned a proprietary code review tool where the diff is supplemented by a list of files ordered by the committer, saying it is important "to label file groupings as semantically effectful". P5 referred to Diffmagic's ordering as a "semi-intelligent grouping", and found it useful, though they felt like they "didn't leverage it enough, especially when I was doing my first pass-through".

GitHub users also answered it was harder to connect related changes across the project (Q5) compared to Diffmagic users ($p=$ .31, $d=$ .72). Few users elaborated on this specifically, but during Task 1, P14 noted that this is easier in Diffmagic, "which is nice, seeing which changes are all just related to this type parameter".

RQ1: Comprehension

Diffmagic helped participants better understand the root cause of changes and associate related changes within the allotted 20 minutes. Participants using Diffmagic also found its organizational structure helpful, and, overall, did not call the large diff repetitive.

B. RQ2: Speed of inspecting the diff

Task 1: Maven SCM. Diffmagic users came to a conclusion about whether to merge the large change within an average of 16.4 minutes (stdev$=$4.3). GitHub users came to a conclusion in an average of 16.6 minutes (stdev$=$2.9). These results are not statistically significant ($p=1.$, $d=$. 6). One Diffmagic user timed out compared to two GitHub users.

Task 2: Flink. Task 2 involved reviewing a much smaller change, only six files. Diffmagic users decided whether the error could originate from this change within an average of 9.3 minutes (stdev$=$4.6). GitHub users came to a conclusion in an average of 15.7 minutes (stdev $=$ 4.4). These results are statistically significant ($p=$. 18, $d=1.4$). No Diffmagic participant timed out, compared to two GitHub participants.

This is particularly interesting given the experience gap between the two groups: Diffmagic participants have, on average, 6.9 years of experience in Java (stdev$=$4.67), while GitHub participants have almost double that, an average of 11.1 years (stdev $=$ 11.6). That they were not considerably slower in performing Task 1 *and* were significantly faster performing Task 2 may speak to the tool's benefits.

Post-study survey. mong the post-study questions, only in Q2, "the size of the diff impacted the difficulty of the task", did Diffmagic not mark an improvement over GitHub; Diffmagic users found size to be significantly more impactful ($p=$. 72, $d=1.3$). This can be because, despite the improved comprehension, the large diff was still a time-consuming task for users. It is also possible that, having cleared away other obstacles, the size of the diff remains the main one.

RQ2: Time to Task Completion

Diffmagic helped participants complete tasks faster and time out less.

C. RQ3: Role of Change Batches

For each Diffmagic user, we computed the proportion of the task time spent in each of Diffmagic's views: Show ‖, change batches, and Miscellaneous. The results are shown in Figure 6. In the larger and more repetitive Task 1, participants spent less time in Show ‖ and more time in change batches. In the smaller Task 2, participants used Show ‖ more, likely because it was possible to look through the entire diff. Participants spent more time in the Miscellaneous view than we anticipated: P14 explained that they are just "able to ignore all of that [the change batches], and then look at the miscellaneous ones" to make sure they did not miss anything.

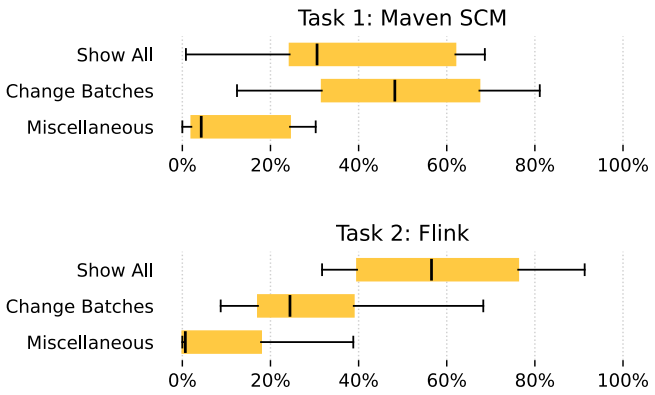

Fig. 6: Usage statistics for the different views in Diffmagic.

Using change batches instead of file lists. We observed Diffmagic users interacting with the list of change batches like they would with the list of files.

The first instance of this was in a style of navigating diffs that we observed for several participants (P1, P2, P5, P13) who rapidly clicked through the interface to change their view of the displayed diff. t times, this was as quick as 1-2 clicks per second for long stretches of time. Early in the task, this took the form of selecting what to click on almost at random, seemingly getting a sample of the diff, with P1 even focusing on a specific cross-file change by clicking to rapidly change between 2–3 files. P5 did this mainly during the first task, and P1 and P2 did this far more heavily during the first task than the second. P13 displayed this behavior consistently throughout their session. For GitHub users, this clicking was between different files in the diff. The one Diffmagic user who displayed this behavior (P5), who had both a file list and a list of change batches available, rapidly clicked between change batches and not files. (This rapid clicking style contrasts with quietly staring for long periods of time without scrolling or clicking, displayed by P3, P4, P7, and P9.)

The second instance was to get a sense of the project's structure. GitHub users P1, P7, and P9 did this by looking through the file list, P1 explaining, "let's just look at the package names, and try and get a sense of the structure". Both P1 and P7 spent a long time doing this, particularly in the larger Task 1. In contrast, Diffmagic users P4, P10, P12, and P14 looked at change batches to the same effect. P4 reasoned about types of changes as categories, hypothesizing in Task 1 that deletions are more important to start with than additions for a backwards compatibility problem. In Task 2, P10 went through every change batch, then went back to Show ll for a wider view. P12 called this "getting an overview" of the diff, and P14 called it "getting the lay of the land".

Using batches to validate hypotheses. complementary pattern to starting with change batches and then moving to Show ll was participants spending the majority of their time in Show ll, and only entering change batches briefly to understand connections to what they were currently inspecting or to validate a hypothesis. ll Diffmagic users except P14 did this at some point during their session: P3, P4, and P5 as their main strategy for Task 2. t the end of their session, P6 said they think this is the better strategy for the bigger diff (i.e., Task 1). Similarly, P10 and P12 used this strategy in Task 1.

Proposing to ignore the content of change batches. When inspecting the diff for Task 1, P14 referred to Diffmagic's grouping of similar updates, saying "what I would most likely use this for is just to be able to ignore all of that". P8, when shown Diffmagic after their GitHub session, hypothesized how they would use Diffmagic outside the study and said that if they used Diffmagic for code review, they would only use the left panel: "It's a lot more important to understand how the idea works within the system than where they got the value for a new argument, because I trust whoever did that."

RQ3: Change Batches

Diffmagic users used change batches to:
- Gain insight into the structure of the project.
- Rapidly switch back and forth between views.
- Validate hypotheses about things they saw in Show ll.

In addition, users hypothesized that in a more realistic code review setting, they would only use the title of change batches, not their content.

VII. Discussion

In this section, we reflect on the results of our study and lay out the limitations of the study and of Diffmagic.

. *dvanced ways to use Diffmagic*

When building Diffmagic, we envisioned it as a way to structurally and methodically explore diffs, one batch at a time, as demonstrated in Section III. While this behavior was observed in the study, we were surprised by some methods participants adopted immediately upon encountering the tool.

One such behavior was using change batches to validate hypotheses formed while viewing the diff in Show ll. This employs Diffmagic's causal structure as an advanced form of the navigation to definition usually available in IDEs. Instead of simply navigating to a definition, switching to the view of the specific change batch shows a wider picture of related changes. This also hints that an easier way to skip from Show ll to the specific change batch could aid in this strategy.

The most interesting suggestion, however, was made by P8 and P14: that in the context of code review within a project's team, the list of change batches is, on its own, sufficient for the review. Doing this collapses a code review into a very low-level design review, where *what* changes were made is under review, and developers are trusted to perform them correctly. P14 went so far as to assign the Miscellaneous view the role of catching changes outside the list of expected changes, to make sure nothing else is hiding in the diff.

B. Building Diffmagic on Semantics

Recent advancements in I, specifically in large language models (LLMs), are already being leveraged for tasks related to code change, including automated commit messages [20, 31], reviewing code [7, 20], automatically resolving

issues [33], and advanced diffing [16]. s LLMs continue to improve, we will continue to witness more adoption of these techniques in code diff comprehension and future work can compare Diffmagic with generative- I tools, and even explore the benefits of integrating Diffmagic with such tools.

However, fundamentally, LLMs rely on patterns observed in text (in our context, code) which can work in many cases but due to their inherently non-deterministic nature, they cannot always produce the same result, nor can they be relied upon to always succeed. Diffmagic takes a more principled approach: since the compiler front-end will return the same ST and the same name resolution edges for the same input diff, Diffmagic yields deterministic batches. Diffmagic also has practical benefits: it does not require external PIs (or, indeed, a network connection), and avoids inference costs and reliance on model services. While in some parts of Diffmagic, LLMs may be leveraged effectively, e.g., for generating intuitive display names of the change batches or enhancing the output of Diffmagic with additional natural language explanations of the changes, the choice to rely only on semantic information for the batching process is deliberate. More experienced users share our view in this regard: P14, who asked for an explanation of how Diffmagic works at the end of their session, said that they were worried Diffmagic is LLM-based. They appreciated the reliance on semantics as "then I have some guarantees about what's there or not. s much as LLMs are super useful, I wouldn't always trust it for that, right?"

C. Limitations and Future Work

Ecological validity of our study. The setup of our study compromises ecological validity by asking participants to inspect diffs of unfamiliar code: this is a rarity for developers. Several participants (P1, P10, P11) remarked on this during their sessions, while P1, P2, P8, and P14 also said that they would usually have the ability to run the test suite. This limits the generalizability of our findings to code review, but we believe our results of diff comprehension in isolation inform us about the qualities Diffmagic would have when reviewing familiar code. Moreover, P5, P8, and P14 theorized about how they would use Diffmagic for familiar code, showing that users foresee the tool having value in their day-to-day work.

Ways to strengthen our results. The comprehension tasks were scored only by the last author, who was blinded to the treatment. Scoring reliability could be further strengthened by using multiple independent judges and reporting inter-rater reliability measures. dditionally, while we evaluated Diffmagic on one code review task and one debugging task, additional studies can be designed to focus on a wider range of tasks and commit types.

The "compiling-commit" assumption. Diffmagic relies on the assumption that the project compiles both before and after the change. If this is not the case, Diffmagic will produce partial results. If either the lhs or rhs has syntactic errors, the created ST will be only partially correct around those regions. Because Eclipse JDT still resolves the identifiers that remain available, Diffmagic can still recover part of the causal structure, but some causal edges may be missing or incorrect. In practice, this can produce fewer batches overall, and in some cases slightly inaccurate batch contents. From our testing, degradation is gradual: the closer each side is to compiling, the more complete and reliable the resulting batches are.

Limitations of the causal structure. The causal structure of the diff is our main organizing principle, but it has several shortcomings. First, it partitions changes into batches, but does not order the batches themselves. When reflecting on Task 1, P6 noted that the main change of the diff "didn't appear until, I think, all the way down near the bottom of the list". Organizing the batches is an important challenge and requires additional design work. Possible approaches include sorting them by size or number of roots, or using LLMs to determine effective strategies. We leave further exploration for future work.

dditionally, the strict directionality of the causal structure does not always fit with user perception. When an import is added or removed, it necessarily becomes the source of its chain of causality, often making it the likely source of its batch. P4 specifically pointed out that this is wrong to them, as a change to the import should be caused by the type's use even though name resolution directs in the opposite way. We have not encountered additional language constructs where users felt this conceptual ambiguity exists, but they may exist.

Finally, in some cases, Diffmagic can fail to group together related changes from the lhs and rhs into the same batch. s discussed in Section IV-B, unifying connected components into a batch operates by pairing the causality sinks, i.e., nodes with no outgoing causal edges. If at least one of the nodes is not a sink, they will not be merged and will wind up in separate batches, even though they represent both sides of a change. For example, if a type substitution was made, and the old or new type has an import statement, Diffmagic finds a causality edge from the type to the import, making the import the causality sink, which means the type change itself does not trigger a union, and the type substitution will not be captured. When P10 and P14 encountered this specific case they noted it was confusing, though P10 said that "if I used it for a week, I would have a very fine-tuned sense of how it works and why it works". Implementing P4's suggestion for the causality direction for imports may solve this instance of the problem.

dditional context within the diff tool. Several participants (P7, P8, P12, P13) reported that they use tools that allow them quick access to additional context about the project from within the diff view. P5 explained that they do not like to have more than one application open at a time, which makes the lack of context limiting. Many mentioned they would have liked access to additional information outside the diff, like documentation (P7), the rest of the project (P6, P8, P11, P12, P13), a call graph (P8), and the build system (P1). P5, P7 and P8 looked up documentation or code for the classes they encountered within the diff. While such improvements are not commonly available in diff viewers, and are orthogonal to the key idea behind Diffmagic, we agree that as part of making Diffmagic ready for the real world, some of these improvements could be useful to its future users.

CKNOWLEDGMENTS

We thank the participants of our study for their enthusiastic engagement. We are grateful to Yotam M. Y. Feldman and Valkyrie Savage for giving us feedback on earlier drafts of this paper, and the reviewers for their valuable feedback in making it better. Many thanks to Zachary Tatlock who originally brought this team together. Mooly Sagiv generously supported Chandrakana Nandi's pursuit of this research alongside her responsibilities at Certora. This work was funded by the European Union (ERC, EXPLOSYN, 101117232). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive gency. Neither the European Union nor the granting authority can be held responsible for them.

REFERENCES

[1] (2026, Mar.) rust-analyzer. ccessed 2026-03-31. [Online]. vailable: https://rust-analyzer.github.io/

[2] P. likhanifard and N. Tsantalis, " novel refactoring and semantic aware abstract syntax tree differencing tool and a benchmark for evaluating the accuracy of diff tools," *CM Trans. Softw. Eng. Methodol.*, vol. 34, no. 2, Jan. 2025. [Online]. vailable: https://doi.org/10.1145/3696002

[3] T. piwattanapong, . Orso, and M. J. Harrold, "Jdiff: differencing technique and tool for object-oriented programs," vol. 14, no. 1, p. 3–36, Mar. 2007. [Online]. vailable: https://doi.org/10.1007/s10515-006-0002-0

[4] . W. ppel and J. Palsberg, *Modern Compiler Implementation in Java, 2nd edition*. Cambridge University Press, 2002.

[5] P. M. Bittner, . Schultheiß, B. Moosherr, T. Kehrer, and T. Thüm, "Variability-aware differencing with diffdetective," in *Companion Proceedings of the 32nd CM International Conference on the Foundations of Software Engineering*, ser. FSE 2024. New York, NY, US : ssociation for Computing Machinery, 2024, p. 632–636. [Online]. vailable: https://doi.org/10.1145/3663529.3663813

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

[7] CodeRabbit, Inc., "CodeRabbit: i code reviews," https://www.coderabbit.ai/, 2026, accessed: 2026-07-17.

[8] M. J. Decker, M. L. Collard, L. G. Volkert, and J. I. Maletic, "srcDiff: syntactic differencing approach to improve the understandability of deltas," *Journal of Software: Evolution and Process*, vol. 32, no. 4, p. e2226, 2020, e2226 JSME-19-0050.R1. [Online]. vailable: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2226

[9] C. Deknop, K. Mens, . Bergel, J. Fabry, and V. Zaytsev, " scalable log differencing visualisation applied to cobol refactoring," in *2021 Working Conference on Software Visualization (VISSOFT)*, 2021, pp. 1–11.

[10] M. J. Diener, *Cohen's d*. John Wiley & Sons, Ltd, 2010, pp. 1–1. [Online]. vailable: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470479216.corpsy0200

[11] G. diffutils. (2025) Output formats (comparing and merging files). ccessed 2026-03-08. [Online]. vailable: https://www.gnu.org/software/diffutils/manual/html_node/Output-Formats.html

[12] G. Docs. (2026) Committing and reviewing changes to your project in github desktop. ccessed 2026-03-08. [Online]. vailable: https://docs.github.com/en/desktop/making-changes-in-a-branch/committing-and-reviewing-changes-to-your-project-in-github-desktop#choosing-how-to-display-diffs

[13] ——. (2026) Comparing commits. ccessed 2026-03-08. [Online]. vailable: https://docs.github.com/en/pull-requests/committing-changes-to-your-project/viewing-and-comparing-commits/comparing-commits

[14] G. Dotzler and M. Philippsen, "Move-optimized source code tree differencing," in *Proceedings of the 31st IEEE/ CM International Conference on utomated Software Engineering*, ser. SE '16. New York, NY, US : ssociation for Computing Machinery, 2016, p. 660–671. [Online]. vailable: https://doi.org/10.1145/2970276.2970315

[15] R. Dufloer, I. Sayar, . Etien, and S. Costiou, "Divergence-driven debugging: Understanding behavioral changes between two program versions," in *2025 IEEE/ CM 33rd International Conference on Program Comprehension (ICPC)*, 2025, pp. 221–225.

[16] G. Eisenkot. (2025, ug.) Building an I code review agent: dvanced diffing, parsing, and agentic workflows. Baz Blog. ccessed 2026-03-29. [Online]. vailable: https://baz.co/resources/building-an-ai-code-review-agent-advanced-diffing-parsing-and-agentic-workflows

[17] K. Etemadi, . Sharma, F. Madeiral, and M. Monperrus, " ugmenting diffs with runtime information," *IEEE Transactions on Software Engineering*, vol. 49, no. 11, pp. 4988–5007, 2023.

[18] J.-R. Falleri and M. Martinez, "Fine-grained, accurate and scalable source differencing," in *Proceedings of the IEEE/ CM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, US : ssociation for Computing Machinery, 2024. [Online]. vailable: https://doi.org/10.1145/3597503.3639148

[19] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th CM/IEEE International Conference on utomated Software Engineering*, ser. SE '14. New York, NY, US : ssociation for Computing Machinery, 2014, p. 313–324. [Online]. vailable: https://doi.org/10.1145/2642937.2642982

[20] L. Fan, J. Liu, Z. Liu, D. Lo, X. Xia, and S. Li, "Exploring the capabilities of llms for code-change-related tasks," *CM Trans. Softw. Eng. Methodol.*, vol. 34, no. 6, Jul. 2025. [Online]. vailable: https:

//doi.org/10.1145/3709358

[21] B. Fluri, M. Wursch, M. PInzger, and H. Gall, "Change distilling:tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[22] B. Fluri, M. Würsch, M. Pinzger, and H. Gall, " retrospective of changedistiller: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 51, no. 3, pp. 852–857, 2025.

[23] E. Foundation. (2026) Eclipse jdt (java development tools) — projects.eclipse.org. ccessed 2026-03-24. [Online]. vailable: https://projects.eclipse.org/projects/eclipse.jdt

[24] V. Frick, C. Wedenig, and M. Pinzger, "Diffviz: diff algorithm independent visualization tool for edit scripts," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 705–709.

[25] X. Ge, S. Sarkar, J. Witschey, and E. Murphy-Hill, "Refactoring-aware code review," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2017, pp. 71–79.

[26] J. Glock, " iding developer understanding of software changes via symbolic execution-based semantic differencing," in *Proceedings of the 2024 IEEE/ CM 46th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE-Companion '24. New York, NY, US : ssociation for Computing Machinery, 2024, pp. 142–144. [Online]. vailable: https://doi.org/10.1145/3639478.3639783

[27] F. Grund, S. . Chowdhury, N. C. Bradley, B. Hall, and R. Holmes, "Codeshovel: Constructing method-level source code histories," in *2021 IEEE/ CM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1510–1522.

[28] N. Harvey and D. DeBellis. (2024, Oct.) Highlights from the 10th DOR report. Google Cloud Blog. ccessed 2026-03-27. [Online]. vailable: https://cloud.google.com/blog/products/devops-sre/announcing-the-2024-dora-report

[29] M. Hashimoto and . Mori, "Diff/ts: tool for fine-grained structural change analysis," in *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, ser. WCRE '08. US : IEEE Computer Society, 2008, p. 279–288. [Online]. vailable: https://doi.org/10.1109/WCRE.2008.44

[30] J. Hopcroft and R. Tarjan, " lgorithm 447: efficient algorithms for graph manipulation," *Commun. CM*, vol. 16, no. 6, p. 372–378, Jun. 1973. [Online]. vailable: https://doi.org/10.1145/362248.362272

[31] J. Houghton. (2023, Nov.) Write your git commits with GitHub copilot. Visual Studio Blog. ccessed 2026-03-29. [Online]. vailable: https://devblogs.microsoft.com/visualstudio/write-your-git-commits-with-github-copilot/

[32] K. Huang, B. Chen, X. Peng, D. Zhou, Y. Wang, Y. Liu, and W. Zhao, "Cldiff: generating concise linked code differences," in *Proceedings of the 33rd CM/IEEE International Conference on utomated Software Engineering*, ser. SE '18. New York, NY, US : ssociation for Computing Machinery, 2018, p. 679–690. [Online]. vailable: https://doi.org/10.1145/3238147.3238219

[33] C. E. Jimenez, J. Yang, . Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" 2024. [Online]. vailable: https://arxiv.org/abs/2310.06770

[34] W. Jones. (2022, Sep.) Difftastic: the fantastic diff. ccessed 2026-02-19. [Online]. vailable: https://www.wilfred.me.uk/blog/2022/09/06/difftastic-the-fantastic-diff/

[35] N. Kerzazi, F. Khomh, and B. dams, "Why do automated builds break? an empirical study," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 41–50.

[36] M. Kim, D. Notkin, D. Grossman, and G. Wilson Jr., "Identifying and summarizing systematic code changes via rule inference," *IEEE Trans. Softw. Eng.*, vol. 39, no. 1, p. 45–62, Jan. 2013. [Online]. vailable: https://doi.org/10.1109/TSE.2012.16

[37] Z. Kurbatova, V. Kovalenko, I. Savu, B. Brockbernd, D. ndreescu, M. nton, R. Venediktov, E. Tikhomirova, and T. Bryksin, "Refactorinsight: Enhancing ide representation of changes in git with refactorings information," in *2021 36th IEEE/ CM International Conference on utomated Software Engineering (SE)*, 2021, pp. 1276–1280.

[38] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, "Symdiff: a language-agnostic semantic diff tool for imperative programs," in *Proceedings of the 24th International Conference on Computer ided Verification*, ser. C V'12. Berlin, Heidelberg: Springer-Verlag, 2012, p. 712–717. [Online]. vailable: https://doi.org/10.1007/978-3-642-31424-7_54

[39] Q. Le Dilavrec, D. E. Khelladi, . Blouin, and J.-M. Jézéquel, "Hyperast: Enabling efficient analysis of software histories at scale," in *Proceedings of the 37th IEEE/ CM International Conference on utomated Software Engineering*, ser. SE '22. New York, NY, US : ssociation for Computing Machinery, 2023. [Online]. vailable: https://doi.org/10.1145/3551349.3560423

[40] ——, "Hyperdiff: Computing source code diffs at scale," in *Proceedings of the 31st CM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, US : ssociation for Computing Machinery, 2023, p. 288–299. [Online]. vailable: https://doi.org/10.1145/3611643.3616312

[41] M. Learn. (2026, Mar.) The .NET compiler platform sdk (roslyn apis) - c#. ccessed 2026-03-31. [Online]. vailable: https://learn.microsoft.com/en-us/dotnet/cshar

p/roslyn-sdk/

[42] M. Mahoney, "The storyteller version control system: tackling version control, code comments, and team learning," in *Proceedings of the 3rd nnual Conference on Systems, Programming, and pplications: Software for Humanity*, ser. SPL SH '12. New York, NY, US : ssociation for Computing Machinery, 2012, p. 17–18. [Online]. vailable: https://doi.org/10.1145/2384716.2384725

[43] M. Martinez, J.-R. Falleri, and M. Monperrus, "Hyperparameter optimization for ast differencing," *IEEE Transactions on Software Engineering*, vol. 49, no. 10, pp. 4814–4828, 2023.

[44] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, ser. MSR '05. New York, NY, US : ssociation for Computing Machinery, 2005, p. 1–5. [Online]. vailable: https://doi.org/10.1145/1083142.1083143

[45] Y. S. Nugroho, H. Hata, and K. Matsumoto, "How different are different diff algorithms in git?: Use –histogram for code changes," *Empirical Software Engineering*, vol. 25, no. 1, p. 790–823, Sep. 2019. [Online]. vailable: http://dx.doi.org/10.1007/s10664-019-09772-z

[46] S. Raghavan, R. Rohana, D. Leon, . Podgurski, and V. ugustine, "Dex: a semantic-graph differencing tool for studying changes in large code bases," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, 2004, pp. 188–197.

[47] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and . Bacchelli, "Modern code review: a case study at google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '18. New York, NY, US : ssociation for Computing Machinery, 2018, p. 181–190. [Online]. vailable: https://doi.org/10.1145/3183519.3183525

[48] Scooter Software. (2026) Beyond compare. ccessed 2026-03-10. [Online]. vailable: https://www.scootersoftware.com/

[49] F. Servant and J. . Jones, "Chronos: Visualizing slices of source-code history," in *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, 2013, pp. 1–4.

[50] Software, "Global code time report," Software.com, Tech. Rep., Jan. 2022, accessed 2026-02-18. [Online]. vailable: https://cdn.prod.website-files.com/6080d45a6168d45abde5cba9/61e068fc15f8dd69e5d836db_Code%20Time%20Report%20by%20Software.pdf

[51] Stack Overflow. (2025) I sentiments and usage. 2025 Stack Overflow Developer Survey. ccessed 2026-03-27. [Online]. vailable: https://survey.stackoverflow.co/2025/ai/

[52] 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 CM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, US : ssociation for Computing Machinery, 2012. [Online]. vailable: https://doi.org/10.1145/2393596.2393656

[53] M. Wang, Z. Lin, Y. Zou, and B. Xie, "Cora: Decomposing and describing tangled code changes for reviewer," in *2019 34th IEEE/ CM International Conference on utomated Software Engineering (SE)*, 2019, pp. 1050–1061.

[54] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[55] Y. Yoon, B. . Myers, and S. Koo, "Visualization of fine-grained code change history," in *2013 IEEE Symposium on Visual Languages and Human Centric Computing*, 2013, pp. 119–126.

[56] G. Zeng, C.- . Sun, K. Gao, and H. Liu, "Difffix: Incrementally fixing ast diffs via context and type information," in *2025 40th IEEE/ CM International Conference on utomated Software Engineering (SE)*, 2025, pp. 2882–2893.