CodeMapper: A Language-Agnostic Approach to Mapping Code Regions Across Commits

Huimin Hu
huhuimin236@gmail.com
CISPA Helmholtz Center for Information Security
Stuttgart, Germany

Michael Pradel
michael@binaervarianz.de
CISPA Helmholtz Center for Information Security
Stuttgart, Germany

Abstract

During software evolution, developers commonly face the problem of mapping a specific code region from one commit to another. For example, they may want to determine how the condition of an if-statement, a specific line in a configuration file, or the definition of a function changes. We call this the *code mapping problem*. Existing techniques, such as git diff, address this problem only insufficiently because they show all changes made to a file instead of focusing on a code region of the developer's choice. Other techniques focus on specific code elements and programming languages (e.g., methods in Java), limiting their applicability. This paper introduces CodeMapper, an approach to address the code mapping problem in a way that is independent of specific program elements and programming languages. Given a code region in one commit, CodeMapper finds the corresponding region in another commit. The approach consists of two phases: (i) computing candidate regions by analyzing diffs, detecting code movements, and searching for specific code fragments, and (ii) selecting the most likely target region by calculating similarities. Our evaluation applies CodeMapper to four datasets, including two new hand-annotated datasets containing code region pairs in ten popular programming languages. CodeMapper correctly identifies the expected target region in 71.0%–94.5% of all cases, improving over the best available baselines by 1.5–58.8 absolute percent points.

CCS Concepts

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

Keywords

Program analysis, evolution, version control

ACM Reference Format:
Huimin Hu and Michael Pradel. 2026. CodeMapper: A Language-Agnostic Approach to Mapping Code Regions Across Commits. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3744916.3773267

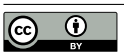
1 Introduction

Tracking code across different commits during the evolution of a project is a vital step for many software development tasks. An empirical study [4] reports that developers examine software histories for a variety of reasons, such as keeping up with changes done by others, understanding the impact of code that the developers are currently developing, and identifying changes that have introduced an error. The study highlights that developers are often most interested in changes affecting their current task, and that a common strategy is to traverse commits with a specific goal in mind. These observations imply that developers do not necessarily want to see all changes, but rather map specific code regions of interest from one commit to another. Questions on Stack Overflow also show that developers want to identify and track specific changes, e.g., by checking the evolution of a variable,[1] or by finding changes on a specific code region.[2] The discussions around these posts show that currently available tools do not fully address developers' needs.

Git, a popular version control system, provides git diff, with various options to show code changes. By default, git diff computes diffs at the line-level, i.e., showing which lines are removed and added. It also supports word-level diffs with the "–word-diff" option. While git diff is powerful, it has limitations when it comes to mapping specific code regions from one commit to another. One limitation is that it reports all changes. However, as noted by Codoban et al. [4] and the posts mentioned above, developers sometimes prefer to focus on specific parts of a change, and they struggle to find a tool for this purpose. Another limitation is that it may fail to accurately match code fragments, especially when the changes are complex. Git also provides "git log -L", which shows the commit history for a specified range of lines. However, it relies on Git's history tracking mechanisms, which also underlie git diff, and hence suffers from the same limitations.

Figure 1 shows an example demonstrating the limitations of git diff. From version 1 to version 2, the function `print` is modified and relocated. However, git diff identifies the changes as deleting code and adding new code (Figure 1a). Instead, a developer interested in mapping this function from version 1 to version 2 would benefit from a tool that accurately recognizes the movement (Figure 1b, yellow). The git diff output also makes it difficult to see how the usage of the `self.y` attribute in the `compute` function evolves, which ideally should be shown as illustrated in Figure 1b (orange).

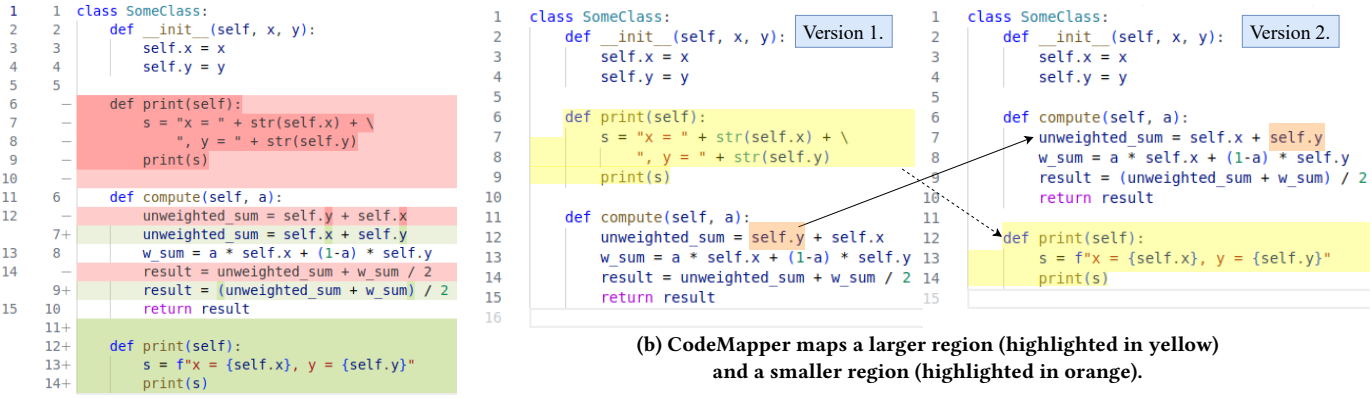
We refer to the problem of mapping a specific code region from one commit of a project to another commit as the *code mapping problem*. To address this problem, we present CodeMapper, a code mapping approach designed to focus on specific code regions of the developer's choice. Given a code region in one commit, CodeMapper finds the corresponding region in another commit. The



ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3773267

[1] How to track changes to a variable's value in a JavaScript file? [30] and How to track changes to a specific value in a Python config file? [31]
[2] Can git diff show only the lines around a specific term? [32]



(a) Line-level diff.

(b) CodeMapper maps a larger region (highlighted in yellow) and a smaller region (highlighted in orange).

Figure 1: Example of mapping code regions in Python. Red and green highlighting is code that got deleted and added according to git diff. The yellow and orange highlighting shows correctly mapped code regions, as provided by CodeMapper.

approach works in two phases. In the first phase, it computes candidate regions by running several techniques with complementary strengths. Specifically, the first phase runs git diff with several algorithms and levels of granularity to identify hunks that modify or relocate the source region. The approach then checks for any moved code and provides details if a region got relocated. Additionally, the approach performs a text search to find exact occurrences of the given code region in the new file. In the second phase, CodeMapper selects the most likely candidate region by computing the similarity between the source region and all candidate regions, while also considering the code around these regions.

Our work on addressing the code mapping problem is not meant to replace git diff, but rather to complement it by enabling developers to focus on a specific code region of their choice. While prior work has addressed related problems, we are not aware of any existing approach that addresses the general code mapping problem. One related line of work focuses on tracking specific code elements, such as a method or a variable, across the entire version history of a project [11, 12, 17]. That work gets evaluated based on its ability to find the commits that modify the specific code element and to identify the kinds of changes made to the code element in these commits. In contrast, our problem formulation assumes two commits to be given and focuses on mapping a specific code region from one commit to the other. Another line of work creates a new version history in which each method is stored in a separate file named by its fully qualified class and method name [13, 14]. In contrast, our approach does not require rewriting history, and it supports tracking arbitrary code regions, instead of focusing on specific code elements. CodeMapper also differs from all the above work by being language-agnostic, instead of relying on language-specific parsers and heuristics, making it more widely applicable.

We evaluate CodeMapper on four datasets: two newly created datasets with annotated code regions of various sizes and in ten popular languages, an existing dataset containing histories of Python comments that suppress static analysis warnings, and a dataset derived from prior work on tracking the history of code elements in Java [12, 17]. The results demonstrate CodeMapper's effectiveness in mapping code regions and clear improvements over tools that are currently used in practice (line-level and word-level git diff). Depending on the dataset, the approach achieves an exact match rate of 71.0%–94.5%, with a recall of 78.1%–97.7% and a precision of 76.4%–97.4%. The results improve over the best available baselines, e.g., by 1.5–58.8 absolute percent points in terms of exact match rate. In addition to being more effective, CodeMapper is also sufficiently efficient for interactive usage, with an average execution time of 2,327 milliseconds to map a code region.

In summary, this work makes the following contributions:

- The first approach to the code mapping problem that is independent of a specific programming language and the kind of code region to map.
- Two reusable datasets, together containing 200 carefully annotated pairs of code regions mapped across pairs of commits in ten popular programming languages.
- Experiments that demonstrate CodeMapper to clearly outperform the current state of the art (git diff) in terms of effectiveness, while providing an efficiency that is on par with currently used tools.

2 Approach

2.1 Terminology and Problem Definition

Before presenting our approach, we define important terms used in this paper and the problem we are addressing. When developers change code, they transform an old version of the code into a new version. Because CodeMapper supports mapping code regions both forward and backward in time, we avoid using terms like "old" and "new" to refer to versions. Instead, we refer to the version from where the region gets mapped as the *source* and the version where the region gets mapped to as the *target*. To identify a contiguous block of changed code, we use the common term *hunk*, i.e., a contiguous block of lines that have been added, deleted, or modified between two file versions:

Definition 1 (Hunk)**.** A hunk is a tuple, $H = (l^{start}_{source}, l^{end}_{source}, l^{start}_{target}, l^{end}_{target})$, where l^{start}_{source} and l^{end}_{source} are the first and last line of the changed block in the source version, and l^{start}_{target} and l^{end}_{target} are the first and the last line of the changed block in the target version.

To define a fragment of code to map, we use character-level granularity:

Definition 2 (Character range)**.** A character range is a tuple, $R = (l_1, c_1, l_2, c_2)$, where l_1 and c_1 are the line number and the character number where the range starts, and l_2 and c_2 are the line number and the character number where the range ends.

All numbers in such a tuple start at one, $l_1 <= l_2$, and a range includes at least one character. Based on the character range, we define a region of code:

Definition 3 (Region)**.** A region is a tuple, $G = (c, f, R)$ where c is a commit hash, f is the file path of the file containing the region, and R is the character range of the region.

Finally, we define the problem we are addressing, which is to map a given region from one commit to its corresponding region in another commit:

Definition 4 (Code mapping problem)**.** Given a source region $G_{source} = (c_{source}, f_{source}, R_{source})$ and a target commit c_{target}, determine the corresponding target region $G_{target} = (c_{target}, f_{target}, R_{target})$.

The target region can fall into one of three cases: First, the source region and target region are the same, because the code in f_{source} has not changed. Second, the target region differs from the source region. Third, the source region does not exist anymore in the target commit, e.g., because the code was deleted, which we represent with the special value $G_{target} = (\perp, \perp, \emptyset)$. Our work supports all three of these cases. Moreover, we address this problem without assuming a specific programming language or kind of program element to map, allowing developers to map arbitrary code regions across different versions of a project.

Note that the code mapping problem differs from the code tracking problem [11, 12, 17] in terms of what is assumed to be given, as well as when and how developers may want to use approaches that address these problems. For code tracking, a single commit c and a specific code element e (e.g., a variable or a method) are given, and the task is to find all commits that modify e and the kinds of changes made to e in these commits. This is useful when developers want to understand the history of a specific code element, e.g., to understand how a method evolved over the project's lifetime. In contrast, the code mapping problem, as addressed here, assumes two commits c_{source} and c_{target}, and a specific code region in the source commit, to be given. The code region may correspond to a specific code element, but it may also be a fragment of a code element or encompass multiple code elements. The code mapping problem is useful when developers want to understand how the differences between two commits impact a specific code region, e.g., when reviewing commits made by others.

2.2 Overview

Figure 2 gives an overview of our approach for tackling the mapping problem. The approach receives a source region and yields the corresponding target region. It consists of two phases: computing candidate regions and selecting the target region.

In phase 1, CodeMapper uses three techniques to compute candidates for the target region. The motivation for using multiple techniques is that no single technique is perfect on its own. Instead, by combining these techniques, we increase the ability of CodeMapper to identify the correct region. The first technique builds on a standard diff computation to obtain hunks, extracts those hunks that either modify the source region or affect its location, identifies candidate regions from those hunks, and then further refines them to precisely identify the beginning and end of the region. The second technique builds on the hunks to detect moved code. Finally, the third technique searches for occurrences of the text in the character range of the source region in the target file, which is motivated by the fact that the hunks may be inaccurate. Each of the three techniques produces a set of candidates, which the approach deduplicates and then gives to phase 2.

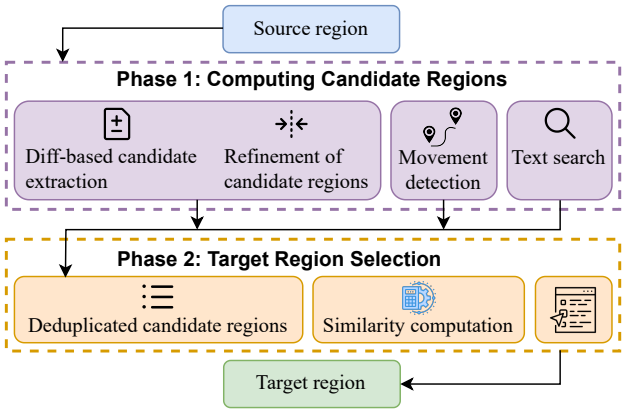


Figure 2: Overview of CodeMapper.

In phase 2, CodeMapper selects the most likely candidate as the target region by computing the similarity between each candidate region and the source region. To compute the similarity between two regions, the approach compares the code in the regions themselves, as well as some contextual code, using Levenshtein distance.

Algorithm 1 provides a more detailed summary of the two phases of our approach, with further discussion in Sections 2.3 and 2.4.

2.3 Computing Candidate Regions

CodeMapper employs three techniques to compute candidates for the target region, as described in the following.

2.3.1 Diff-Based Candidate Extraction. This technique uses hunks in a diff report to extract candidate regions. The approach performs four steps. At first, it computes the hunks between the source and target commits based on an existing "diff" tool (line 2 in Algorithm 1). For each hunk, the approach then determines the positional relationship between the source region and the hunk (lines 4 to 14). The third step computes candidate regions based on the positional relationship (lines 15 to 16). Finally, the approach refines the candidate regions to obtain more precise results. We present each of these four steps in more detail in the following.

Step 1: Extracting Hunks. Because different diff algorithms may produce different results, CodeMapper uses four different algorithms to increase the likelihood of finding the correct candidate region. The four algorithms are all implemented in the "git diff" tool: (i) *myers*, the default algorithm, (ii) *minimal*, spends extra time to make sure the smallest possible diff is produced, (iii) *patience*, tends to be more human-readable, and (iv) *histogram*, extends the patience algorithm to support low-occurrence common elements.

Algorithm 1: CodeMapper

Input: Source region G_{source} : (c_{source}, f_{source}, R_{source}), repository *repo*, target commit c_{target}
Output: Target region G_{target}: (c_{target}, f_{target}, R_{target})

```
// Phase 1: Computing candidate regions
1  Candidates, overlappingInfo ← ∅
2  diffs ← getDiffReports(repo, c_source, c_target, f_source)
3  for diff in diffs do
4      for H in getHunks(diff) do
5          overlapLoc ← checkOverlap(H, R_source)
6          if overlapLoc == "fully covered" then
7              addUniqueCandi(Candidates,
                 getRefinedRanges(H, f_source, R_source))
8          else if overlapLoc in {"top", "middle", "bottom"} then
9              overlappingInfo ← (overlapLoc, H)
10         else if overlapLoc == "disjoint" then
11             if H.l^end_source < R_source.l_1 then
12                 updateCandidateLineNums(R_source, H)
13         if lineNums_overlapping == lineNums_source then
14             break
15 for info in overlappingInfo do
16     addUniqueCandi(Candidates, getCandidates(info))
17 if R_source is fully_deleted then
18     addUniqueCandi(Candidates, detectMovements(R_source))
19 addUniqueCandi(Candidates, searchTexts(G_source))
   // Phase 2: Target region selection
20 similaries ← computeSimilarity(G_source, Candidates)
21 targetIdx ← similaries.index(max(similarities))
22 G_target ← Candidates[targetIdx]
23 return G_target
```

In addition, we configure "git diff" to apply these algorithms at the line-level and at the word-level, resulting in a total of eight diff reports. CodeMapper deduplicates these reports and proceeds to extract candidate regions from the deduplicated reports.

Step 2: Determining Positional Relationships. Given the hunks in a diff report, CodeMapper determines which hunks are relevant for the source region by analyzing their positional relationship. We consider a hunk to be *relevant* if it either modifies the source region or affects its location. To collect relevant hunks, CodeMapper considers three major categories of positional relationships between the source region and hunks, as shown in Figure 3, which we further describe in the following:

i *Fully covered*: The source region is fully covered by a hunk, i.e., the hunk is relevant.
ii *Overlapping*: The source region overlaps with one or more hunks at the top, middle, or bottom, i.e., the hunk is relevant. Specifically, we consider four cases: top overlapping, bottom overlapping, top-bottom overlapping, and middle overlapping.
iii *Disjoint*: The source region does not overlap with any hunk, i.e., its content remains unchanged. However, if the location of the source region change because of the hunks preceding it, then the hunk is relevant.

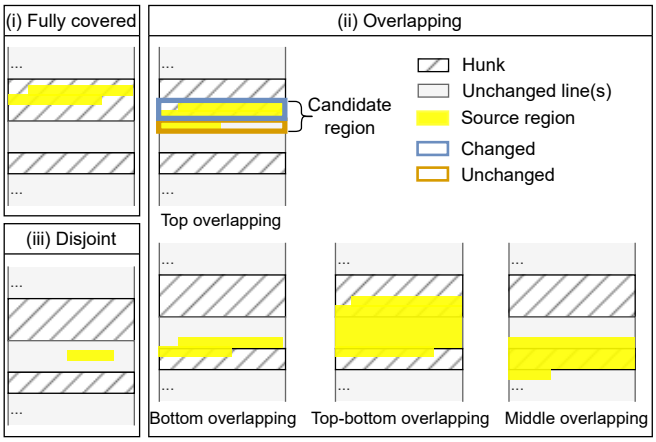


Figure 3: Positional relationships between a source region and hunks.

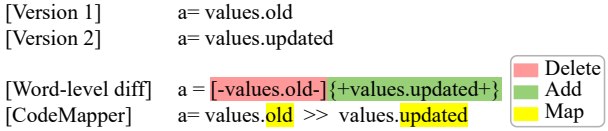


Figure 4: Refinement of candidate regions.

Step 3: Extracting Candidate Regions. After identifying the relevant hunks, CodeMapper extracts candidate regions from them (function *getCandidates* in line 16 of Algorithm 1). Recall from Definitions 2 and 3 that the range of a candidate region is represented as R_{target}=(l_1, c_1, l_2, c_2). For hunks that fully cover the source region, CodeMapper first checks whether the region has been removed in the target commit. To this end, the approach checks whether the target block (l^{start}_{target}, l^{end}_{target}) of a hunk is empty, and if so, generates the special candidate region ($\perp$, $\perp$, $\emptyset$). Otherwise, CodeMapper continues by determining a coarse-grained candidate range by assigning c_1 to a default value 1, c_2 to the length of line l^{end}_{target} in the target file, and updating the candidate range as (l^{start}_{target}, 1, l^{end}_{target}, c_2). Then, the refinement step described in the following paragraph will refine the range to make it more precise, e.g., by excluding unrelated characters. For overlapping hunks, e.g., in the case of top overlapping (refer to Figure 3), the candidate region is a combination of two parts: one where the source region overlaps with the top overlapping hunk (marked with a blue box), and another that remains unchanged (marked with an orange box). CodeMapper gets a coarse-grained start position of R_{target} by setting (l_1, c_1) as (l^{start}_{target}, 1), and searches the unchanged part to get the (l_2, c_2). Then, the approach runs the refinement step to get a refined start position. The other overlapping cases are handled similarly. For disjoint hunks, CodeMapper first initializes candidate ranges to be the same as the source range, and then updates the line numbers by accounting for the changes caused by the hunks located above the source region (line 12).

Step 4: Refinement of Candidate Regions. To motivate the fourth and final step of our diff-based candidate extraction, consider the example in Figure 4. The code change in the first two lines modifies

`old` into `updated`. Suppose the source region is `old`, i.e., ideally, we want to find `updated` as the target region. However, word-level diff gives the candidate region `values.updated`, even though it identifies changes at the word level, the candidate is still insufficiently precise.

To obtain more precise candidate regions, CodeMapper refines the candidate regions extracted from hunks that fully cover a source region and from overlapping hunks (excluding middle overlapping). Intuitively, the refinement step aims to adjust the start and end positions of the candidate regions to more closely match the source region. Algorithm 2 summarizes the refinement step, which takes three inputs: a source range R_{source}, a reference hunk H_{ref}, and a candidate range R_{target}. The H_{ref} depends on the scenario:

- If a hunk fully covers the source region, then H_{ref} is H_{fully_cover}, which is used to refine both the start and end positions of the candidate region.
- For top overlapping and bottom overlapping hunks, the reference hunk is H_{top} and H_{bottom}, respectively.
- For a top-bottom overlapping hunk, CodeMapper runs the algorithm twice: H_{top} as H_{ref} for refining the start position and H_{bottom} as H_{ref} for refining the end position.

The algorithm shows the details on how to get the refined start position of a candidate region. To refine the end of the region, CodeMapper follows a similar approach, but analyzing the H_{ref} in reverse. The algorithm starts by identifying in the reference hunk the first line containing a delete operation. This ensures that the refinement starts from the most significant point of change, aligning with the start line of the source region. Subsequently, the algorithm uses a word-level diff report to extract text fragments, which represent individual changes, such as deletions, unchanged segments, and additions in the line. For each such fragment, CodeMapper calculates fragment lengths and updates character indices (line 7). If the current character index exceeds the starting character position, the algorithm further refines the position at the character-level (line 5). At the character-level, the algorithm identifies the overlap between the current fragment and the start of the source region, and then uses this information to exclude any preceding characters not present in the source region. Finally, the algorithm outputs a refined candidate range R'_{target}.

For example, in Figure 4, CodeMapper checks the overlap between the fragment `values.old` and source characters `old`, excludes the `values.`, and gives a candidate `updated`.

2.3.2 Movement Detection. Developers sometimes move code fragments via cut-and-paste from one location to another. Unfortunately, git diff may not always detect such movements, which could lead CodeMapper to miss the correct target region. For example, the `print(s)` in line 9 is moved to line 14 in Figure 1b, but Figure 1a misidentifies it is deleted. To address this issue, CodeMapper includes a movement detection technique that identifies moved code fragments, which is loosely inspired by prior work on generating edit scripts [15]. We consider two kinds of movements: (i) *Vertical movements:* Lines are cut-and-pasted to another location, without modifying the lines. (ii) *Horizontal movements:* Lines are moved inside or outside a structural unit, such as a block of code.

To identify otherwise missed movements, CodeMapper marks a source region as potentially moved if it is fully deleted according to line-level git diff. For any such potentially moved source regions, the approach checks all hunks to find a corresponding movement. Specifically, it checks for hunks that contain all lines of the source region as newly added lines, which detects vertical movements, and for hunks that contain the source region's lines except for whitespace, which detects horizontal movements. If such a match is found, it adds those "added" lines as a candidate region. This step is summarized as *detectMovements* in line 18 of Algorithm 1.

Algorithm 2: Refinement of candidate ranges.

Input: Source region with range R_{source}: $(l_{s1}, c_{s1}, l_{s2}, c_{s2})$, reference hunk: H_{ref}, candidate range R_{target} : $(l_{t1}, c_{t1}, l_{t2}, c_{t2})$
Output: Refined candidate range: R'_{target} : $(l'_{t1}, c'_{t1}, l_{t2}, c_{t2})$

1 $l'_{t1} \leftarrow getFirstModifiedLine(H_{ref})$
// **Identify text fragments in the modified line**
2 $currentSrcCharIdx, cndCharIdx \leftarrow 0$
3 **for** *fragment in* l'_{t1} **do**
4 **if** $currentSrcCharIdx \geq c_{s1}$ **then**
// **Further refine character indices**
5 $c'_{t1} \leftarrow computeStartChar(R_{source}, fragment, cndCharIdx)$
6 **break**
7 $currentSrcCharIdx, cndCharIdx \leftarrow$ $updateCharIndices(fragment, currentSrcCharIdx, cndCharIdx)$
8 $R'_{target} \leftarrow (l'_{t1}, c'_{t1}, l_{t2}, c_{t2})$
9 **return** R'_{target}

2.3.3 Text Search. As a third technique to identify candidate target regions, CodeMapper searches for the exact string from the source region in the target file. Text searching is typically useful for relatively small source regions, e.g., a single variable or a short expression, which may get lost in a larger hunk. The approach adds any exactly identical occurrences of the source region in the target file as candidate regions (Algorithm 1, line 19).

2.4 Target Region Selection

The approach described so far (phase 1) results in three sets of candidates: diff-based, movement-detected, and searched. Phase 2 selects the most likely target region from these candidates. At first, CodeMapper merges and deduplicates these sets. Next, it computes the similarity (details in next paragraph) between the source region and each candidate region. Finally, the approach selects the candidate with the highest similarity score as the target region. If multiple candidates have the same priority, CodeMapper heuristically prioritizes the diff-based candidates over the movement-detected candidates, and then the searched candidates. A benefit of ranking all candidates and then selecting the most similar one is that we avoid the need to set a similarity threshold [5, 28], which can be challenging to determine and may lead to incorrect results [24].

To compute the similarity of a source region and a candidate region, CodeMapper considers both the content of the regions and their context. As a similarity metric, we use Levenshtein similarity, which is based on the number of character-level edits required to transform one string into another. Other metrics, e.g., based on neural code embeddings could be easily integrated into CodeMapper, but we have chosen Levenshtein similarity for its simplicity.

Because the correct target region may modify the code in the source region, computing the similarity only between these two regions may give misleadingly low similarities. Instead, CodeMapper also considers the context of these regions, i.e., a fixed number of unchanged lines before and after these regions. Function *computeSimilarity* (Algorithm 1, line 20) receives a source region G_{source} and a set of candidates as inputs. It iterates through each candidate region G_{target} and computes its similarity to the source region as follows:

$$simil(addContext(G_{source}), addContext(G_{target}))$$

where the function *addContext*() retrieves the context for each region. One exception to the above is that CodeMapper does not add any context when computing the similarity for movement-detected candidate regions. The rationale is that moved code fragments are likely to have different context lines in the source and target files, as the surrounding lines are often changed when the region is relocated. We set the context size as 15 lines before and 15 lines after the region. This size strikes a balance: A smaller size risks incorrect candidate regions occasionally sharing the same context lines, while a larger size may reduce the impact of the actual region. Section 3.4.1 evaluates the impact of the context size on our results.

3 Evaluation

3.1 Research Questions

Our evaluation investigates the following research questions:

RQ1 Effectiveness: How effective is CodeMapper in finding the target region for a given source region?

RQ2 Impact of parameters and components: How do different parameters and components impact CodeMapper's effectiveness?

RQ3 Efficiency: How much execution time does CodeMapper take?

3.2 Experimental Setup

3.2.1 Datasets. We apply CodeMapper to three datasets, which are all based on real-world commits. Table 1 provides an overview of the datasets, which we explain in more detail in the following.

Annotated Data A and B. As there is no existing dataset for the code mapping problem, we create and manually annotate two new datasets, called Data A and Data B. Each dataset consists of 100 pairs of a source region and its corresponding target region. We start by selecting 20 projects on GitHub based on programming language and popularity. Specifically, we pick ten programming languages that are popular on GitHub, randomly select two popular projects for each language, and then randomly sample from the latest 200 commits of each project. To avoid trivial code mapping tasks, we select only files that are modified in the chosen commits. To select source and target commits, we define *commit distance* as the number of commits between source commit and target commit that change the specified file. We sample half of the source-target pairs from neighboring commits, i.e., a commit distance of one, and the other half from non-neighboring commits with a maximum distance of five. Additionally, we randomly choose for each pair a mapping direction (forward or backward), i.e., whether the old or the new code version serves as the source.

Next, we annotate source-target pairs, covering four change operations: no change (25%) and change, move, and delete (75%). For Data A, we first manually select a source region and then its corresponding target region. To encompass various scenarios, we annotate source regions of different sizes (each contributing 20%): single identifier or word, single expression or part of a sentence, single line, multiple lines, and structural unit. For Data B, to further reduce bias, we select source regions automatically and annotate only the corresponding target regions manually. To prevent meaningless source ranges, e.g., a generated source region starting in the middle of a token, we compute the Abstract Syntax Tree (AST) of each selected file with tree-sitter, and then generate source regions with three strategies (each contributing 33.3%): (i) randomly select a single node as a source region, (ii) randomly select a sequence of consecutive sibling nodes as a source region, and (iii) randomly select a sequence of consecutive lines as a source region. The generated source regions vary in size, ranging from individual tokens, over partial and full lines, to multiple consecutive lines. Given the automatically generated source regions, we manually annotate the target regions. For both Data A and Data B, two of the authors annotate each example individually. One annotator is a PhD-level researcher, the other is a senior researcher. Both have over five years of experience in software development and extensive experience in software evolution. The annotation process leads to an initial agreement on 94/100 and 92/100 examples, followed by a discussion during which the annotators resolve any disagreements.

Data from Suppression Study. CodeMapper is valuable not only for helping developers find target regions based on a given source region, but also for enabling empirical studies on software evolution. For example, our approach could be used to support empirical studies on the evolution of specific kinds of code elements, such as type annotations [10], comments [16], or suppressions of static analysis warnings [16]. To evaluate this usage scenario, we reuse a dataset from a study on suppressions in Python code bases [16]. The term "suppression" here refers to the intentional practice of ignoring certain warnings generated by static analysis tools by adding an annotation or special comment into the code. The existing dataset contains histories of suppressions, i.e., a sequence of code changes that tracks a specific suppression across the lifetime of a project. The original dataset has been created by a custom tool that tracks suppressions in Python code bases, and has been validated for correctness by the original authors [16]. We here evaluate whether CodeMapper could have been used to map suppressions in Python code bases from one commit to another, which would have saved the effort of creating a custom tracking tool.

The suppression study data consists of 187 source-target pairs, each representing a change that modifies a suppression in a Python file. We target all code changes used in the original study where (i) the file containing the suppression still exists in the new version and (ii) the line containing the suppression is involved in a change. In addition, the original dataset contains changes where entire files are deleted and where the line containing the suppression is not changed at all. We exclude these changes here because those suppressions are trivial to map.

Data from Prior Work on Code Tracking. To further evaluate generalizability, we build on a dataset from CodeTracker, i.e., prior work on code tracking [12, 17]. Since our code mapping task differs from their code tracking task (cf. end of Section 2.1), we process

Table 1: Summary of datasets.

	Annotated data		Suppression study data	CodeTracker data		
	Data A	Data B		Variable	Block	Method
Programming language(s)	Python, Java, JavaScript, C#, C++, Go, Ruby, TypeScript, PHP, HTML		Python	Java		
Number of projects	20		8	10		
Number of source-target pairs	100	100	187	530	949	526
Avg. char size of source region	210.5	39.8	26.1	7.8	784.6	930.4
Avg. char size of target region	204.8	31.5	13.3	7.7	742.2	904.8
Avg. commit distance	2.1	2.0	–	15232.5	4626.6	3695.0
Direction of change	50% forward, 50% backward		100% forward	100% backward		

$$recall = \frac{overlapping\_chars}{all\_ground\_truth\_chars}$$

$$precision = \frac{overlapping\_chars}{all\_target\_region\_chars}$$

$$\text{F1-score} = 2 \cdot \frac{recall \cdot precision}{recall + precision}$$

* *overlapping_chars* is the number of common characters between the target region and the ground truth.

$$char\_dist = |i - i'| + |j - j'|$$

Figure 5: Evaluation metrics.

their data to retrofit it to the code mapping task. The original dataset contains ground truth histories of variables, blocks, and methods in Java projects, where each history consists of a sequence of commits $[c_1, c_2, ..., c_n]$ that modify the code element. To transform this data into a code mapping dataset, we extract all pairs of commits that are consecutive in the given code element history, i.e., (c_i, c_{i+1}) for $i = 1, ..., n-1$. Note that the project's history may contain many other commits in between such a pair of commits, which affect the code around the code element in question, but not the code element itself. Furthermore, to fit the existing data into our code mapping task, we require the code region (Definition 2) of the code element to be specified in both commits. To this end, we parse the corresponding code files and extract the detailed location of the changed code elements. Following the original work [12, 17], the task is to map the code regions backward in time, and we focus on the "test" partition of the entire dataset. After this processing, we obtain a total of 2,005 source-target pairs, including 530 for variables, 949 for blocks, and 526 for methods.

3.2.2 Baselines. We compare CodeMapper against two baselines: line-level git diff and word-level git diff, which we refer to as $\text{diff}_{\text{line}}$ and $\text{diff}_{\text{word}}$, respectively. We select these baselines because they are widely used in practice and because our approach builds on them. Given a source region, we use git diff to identify its relevant hunk(s). Then, we use the corresponding hunk(s) in the target commit as the target region. This process mimics what a developer trying to map a code region would do using git diff.

In addition to reusing data from prior work on code tracking (Section 3.2.1), we also considered directly comparing against work on code tracking. The two most recent approaches are CodeTracker [12, 17] and FinerGit [14]. However, these approaches are not directly comparable to CodeMapper because they address a different problem, namely the problem of finding the commits in which a method, variable, or code blocks gets changed (cf. Section 3.2 of [17] and Section 5.1 of [14]).[3] In contrast, our work assumes a source commit and a target commit to be given, and aims to identify the target region in the target commit that corresponds to a source region in the source commit.

3.2.3 Evaluation Metrics. We evaluate an identified target region by comparing it with a ground truth using several metrics. First,

[3]CodeTracker is also evaluated on its ability to determine the kind of code change, e.g., whether method's documentation or parameters have changed, but this is not the focus of our evaluation.

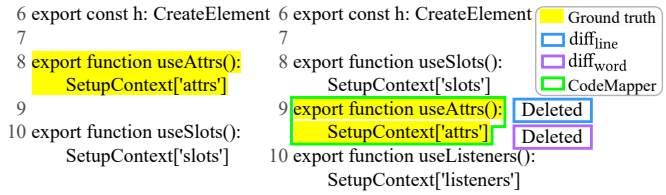


Figure 6: A source region involved in a vertical movement. Only CodeMapper finds an exact match of the target region.

we check whether the target region *overlaps* with the ground truth. If yes, we further distinguish between an *exact match*, i.e., the predicted target region is exactly the same as the ground truth and a *partial overlap*. In addition, we compute for all target regions the metrics in Figure 5. Finally, for partial overlaps, we compute the *character distance* between the target region and the ground truth. To compute this, we view the target region and the ground truth as substrings in the sequence of characters of a file, ranging from indices i to j and i' to j', respectively. The character distance is defined as shown in Figure 5. For example, if one region starts at 20 and ends at 55, and the other region starts at 18 and ends at 63, then the character distance is $|20 - 18| + |55 - 63| = 10$.

3.3 RQ1: Effectiveness

3.3.1 Results on Data A. The leftmost block (Data A block) of Table 2 shows the results of mapping code regions in the Data A. As there are 100 tasks in the dataset, all percentages mentioned in the following equal the absolute numbers. CodeMapper identifies 95% overlapping target regions, compared to $\text{diff}_{\text{line}}$ with 89% and $\text{diff}_{\text{word}}$ with 86%. Specially, our approach successfully identifies 77% exactly matched target regions, outperforming both $\text{diff}_{\text{line}}$ (43%) and $\text{diff}_{\text{word}}$ (54%). The exact matches include eight cases where both baselines fail to identify movements, while CodeMapper provides exact matches by utilizing its movement detection. The character distance indicates that the 18 partial overlaps of CodeMapper are 55.2 characters away from the expected regions, on average, which is clearly better than both baselines (147.0 and 71.1 characters). That is, even when the approach cannot exactly identify the target region, it still provides more precise regions than the baselines.

Figure 6 shows an example where a source region is moved vertically, with lines 8 and 10 swapped. The source and expected target regions are highlighted in yellow. Both baselines misidentify

Table 2: Results on Data A, Data B, and Suppression study data.

	Data A			Data B			Suppression study data		
	$\text{diff}_{\text{line}}$	$\text{diff}_{\text{word}}$	CodeMapper	$\text{diff}_{\text{line}}$	$\text{diff}_{\text{word}}$	CodeMapper	$\text{diff}_{\text{line}}$	$\text{diff}_{\text{word}}$	CodeMapper
Overlapping	89 (89.0%)	86 (86.0%)	**95 (95.0%)**	**91 (91.0%)**	87 (87.0%)	**91 (91.0%)**	133 (71.1%)	125 (66.8%)	**156 (83.4%)**
Exact matches	43 (43.0%)	54 (54.0%)	**77 (77.0%)**	44 (44.0%)	54 (54.0%)	**71 (71.0%)**	41 (21.9%)	41 (21.9%)	**151 (80.7%)**
Char. dist.	147.0	71.1	**55.2**	59.1	61.7	**18.5**	199.4	194.1	**29.2**
Recall	0.890	0.849	**0.932**	**0.906**	0.859	0.892	0.709	0.666	**0.834**
Precision	0.656	0.741	**0.889**	0.681	0.721	**0.883**	0.351	0.377	**0.824**
F1-score	0.705	0.766	**0.898**	0.721	0.741	**0.882**	0.412	0.433	**0.827**

Table 3: Results on CodeTracker data: Variable, Block, and Method.

	Variable			Block			Method		
	$\text{diff}_{\text{line}}$	$\text{diff}_{\text{word}}$	CodeMapper	$\text{diff}_{\text{line}}$	$\text{diff}_{\text{word}}$	CodeMapper	$\text{diff}_{\text{line}}$	$\text{diff}_{\text{word}}$	CodeMapper
Overlapping	**432 (81.5%)**	363 (68.5%)	414 (78.1%)	**890 (93.8%)**	**890 (93.8%)**	871 (91.8%)	**516 (98.1%)**	**516 (98.1%)**	**516 (98.1%)**
Exact matches	225 (42.5%)	226 (42.6%)	**401 (75.7%)**	702 (74.0%)	746 (78.6%)	**776 (81.8%)**	449 (85.4%)	489 (93.0%)	**497 (94.5%)**
Char. dist.	643.6	882.6	**75.8**	989.6	1287.2	**126.4**	615.8	1522.0	**384.3**
Recall	**0.815**	0.685	0.781	**0.927**	0.924	0.904	**0.977**	**0.977**	**0.977**
Precision	0.449	0.455	**0.764**	0.877	0.880	**0.900**	0.962	0.962	**0.974**
F1-score	0.468	0.469	**0.767**	0.885	0.886	**0.898**	0.962	0.962	**0.972**

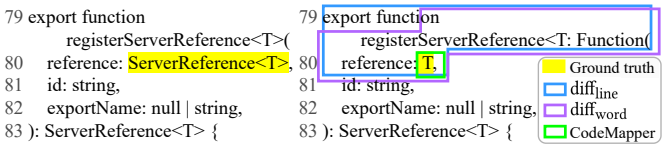


Figure 7: A close match found by refining candidate regions.

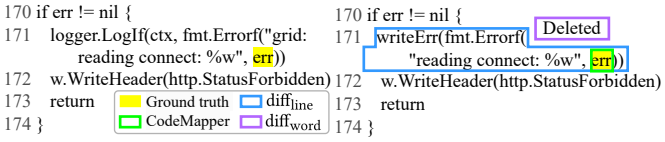


Figure 8: An exact match found by searching texts.

the source region as deleted. Instead, CodeMapper benefits from its movement detection (Section 2.3.2) and finds the exact target region, highlighted in green.

Figure 7 illustrates the importance of refining candidate regions. Both baselines report a region much larger than the expected target region. Although CodeMapper does not achieve an exact match, its output is very close to the ground truth. This is mainly because it successfully refines a diff-based candidate (Step 4 of Section 2.3.1).

3.3.2 Results on Data B. The Data B block of Table 2 presents the results on this dataset. CodeMapper identifies 71% exact matches, surpassing $\text{diff}_{\text{line}}$ at 44% and $\text{diff}_{\text{word}}$ at 54%. It also achieves the lowest average character distance of 18.5 characters. $\text{diff}_{\text{word}}$ has a larger average character distance than $\text{diff}_{\text{line}}$ because character distance is calculated only for partial overlaps, and $\text{diff}_{\text{line}}$ has fewer partial overlaps. This indicates that $\text{diff}_{\text{line}}$ incorrectly maps some cases, while $\text{diff}_{\text{word}}$ provides a partial overlap. Additionally, CodeMapper achieves the highest precision (0.883) and the highest F1-score (0.882). However, $\text{diff}_{\text{line}}$ attains the highest recall (0.906), followed by CodeMapper with a recall of 0.892. This difference occurs because $\text{diff}_{\text{line}}$ always assigns a line-level target region, even when only a single token within the line is affected. These results, consistent with those for Data A, show that CodeMapper is effective at finding target regions for varying sizes of source regions across different programming languages.

Figure 8 provides an example where the line containing the source region has changed, but the source region itself remains the same. $\text{diff}_{\text{line}}$ reports the entire modified line (highlighted in blue) as the target region, and $\text{diff}_{\text{word}}$ identifies the change as a deletion. Meanwhile, CodeMapper benefits from text search (Section 2.3.3) and accurately detects an exact match, marked with green color.

3.3.3 Results on Suppression Study Data. The "Suppression study data" block of Table 2 summarizes the results of mapping Python suppressions. CodeMapper identifies 151 (80.7%) exact matches, outperforming both $\text{diff}_{\text{line}}$ and $\text{diff}_{\text{word}}$, which both identify only 41 (21.9%) exact matches. This improvement of 58.8 absolute percent points is mainly due to refinements of candidate regions, the movement detection, and the text search. CodeMapper also significantly reduces the character distance to 29.2, showing an improvement over $\text{diff}_{\text{line}}$ (199.4) and $\text{diff}_{\text{word}}$ (194.1). It also achieves the highest recall at 0.834, the highest precision at 0.824, and the highest F1-score at 0.827. Excluding the 21.9% exact matches found by all three approaches, the difference between the partial overlaps arises because $\text{diff}_{\text{word}}$ focuses on detailed intra-line changes, whereas $\text{diff}_{\text{line}}$ considers entire lines. Entire lines have a higher chance of overlapping with the expected target regions. In contrast, by trying to identify detailed changes, $\text{diff}_{\text{word}}$ achieves a lower character distance and a higher precision of 0.377 ($\text{diff}_{\text{line}}$ with 0.351).

Figure 9 shows an example where a suppression is deleted in the target file. The baselines give incorrect mappings for the source

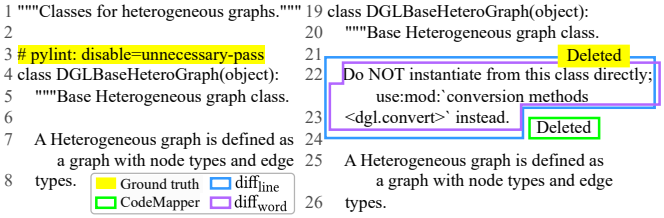


Figure 9: Only CodeMapper correctly recognizes that the source region gets deleted.

region. This could mislead an empirical study on suppression evolution by suggesting that the suppression is moved or changed. Instead, CodeMapper correctly identifies the deletion, matching the expected ground truth.

3.3.4 Results on CodeTracker data. Table 3 presents the mapping results on the CodeTracker data. CodeMapper identifies the most exact matches and the smallest character distances for partial overlaps across all three types of program elements. Specifically, CodeMapper achieves exact match rates of 75.7% for variables, 81.8% for blocks, and 94.5% for methods, outperforming the baselines by up to 33.2 percentage points. Compared to the other three datasets, the results show relatively larger character distances, which can be attributed to the characteristics of CodeTracker data: Variables are generally small and may occur multiple times, causing partial overlaps to cover larger code regions. For blocks and methods, since their source and target regions are much larger (as shown in Table 1), a partial overlap that fails to correctly identify the expected start or end lines may include multiple extra lines, which increases the character distance. In line with the other datasets, CodeMapper achieves the highest precision and F1-score, showing its contribution over the state of the art.

3.3.5 Analysis of Incorrect Target Regions. To gain insights into cases where CodeMapper selects incorrect target regions, we analyze 219 corresponding cases from the four datasets. We observe three recurring reasons for incorrect target regions: (i) *Semantic changes not accurately captured by diff*: This category includes 98 cases where semantic changes are not accurately captured by diff. For example, in one case, `x:CompileBindings="True"` was removed, and `ClassModifier="internal"` was added, which diff incorrectly reported as a match, even though the source region was actually deleted. (ii) *Coincidental occurrences of the source region*: This comprises 75 cases where coincidental occurrences of the source region appear at unrelated locations. CodeMapper mitigates this kind of mis-selection by using unchanged lines as context. (iii) *Detecting file renames and movements*: Detecting file renames and movements is generally challenging, especially when a file undergoes significant changes. CodeMapper uses git log to detect file renames and movements, which sometimes (46 cases) fails to identify the correct file in the target commit. Language-specific techniques for detecting file names and movements, such as RefactoringMiner [29], as used by CodeTracker [12, 17], could mititgate this issue, but is inherently limited to the programming language it supports.

3.3.6 Impact of Programming Language. To investigate whether the mapping results differ across programming languages, we analyze

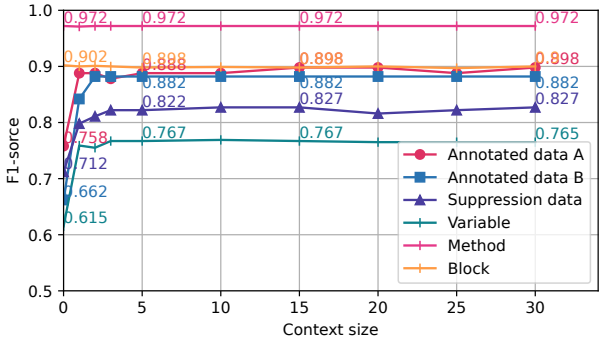


Figure 10: F1-scores with different context sizes.

the results of Data A and B grouped by language. For Data A, F1-scores range from 0.698 to 1.000, with examples including Python (0.950), Java (0.853), and Ruby (1.000). For Data B, F1-scores range from 0.750 to 1.000, with examples including JavaScript (0.870), C++ (0.871), and Go (0.980). We attribute these variations in F1-scores to the randomized procedure of creating these datasets. For instance, a particularly complex source region appearing in one language could lower the F1-score for that language. Additionally, the fixed annotation ratios (e.g., 25% no-change, 50% forward time-order) and the inclusion of non-code files (e.g., Markdown and YAML) contribute to an unbalanced number of source regions across languages. Although the possibility of language-specific trends cannot be entirely excluded, the present results do not provide consistent or statistically significant evidence to support this conclusion.

Answer to RQ1: CodeMapper is effective at mapping code regions for various kinds of source regions across different programming languages. The approach identifies 77.0% exact matches for Data A, 71.0% for Data B, 80.7% for the suppression study data, 75.7% for CodeTracker data variable, 81.8% for block, and 94.5% for method, surpassing the best available baselines by 1.5–58.8 absolute percent points.

3.4 RQ2: Impact of Parameters and Components of CodeMapper

To better understand the impact of different parameters and components of CodeMapper on its effectiveness, we conduct two ablation studies: one examining different context sizes and another analyzing specific components of the approach.

3.4.1 Context Size. CodeMapper uses code context to help identify target regions (Section 2.4). We evaluate different context sizes and present the corresponding F1-scores in Figure 10. We plot the F1-score as it represents the overall trend and in line with the other metrics. The results show that using a context size of five or more provides better results than small values, meaning that CodeMapper effectively leverages contextual information. At the same time, context sizes between 5 and 20 provide similar effectiveness, i.e., the approach is robust to minor changes of this parameter. We use 15 as the default context size in all other experiments.

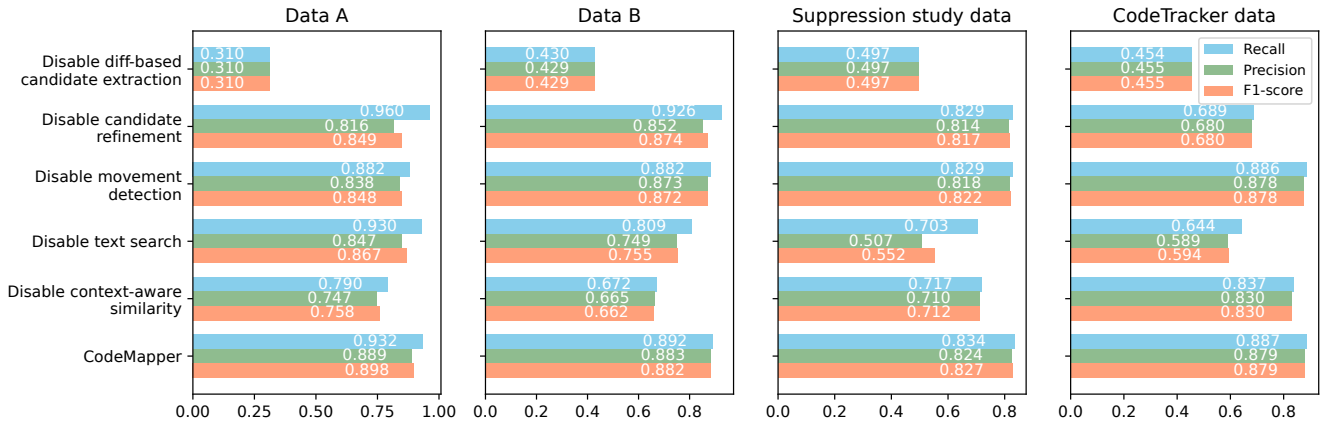


Figure 11: Ablation study of different components.

3.4.2 Importance of Different Components. We study five variants of the approach, each with one part disabled. Specifically, we disable diff-based candidate extraction, refinement of candidate regions, movement detection, text search, and the context-aware similarity metric. Figure 11 shows the results of the ablation study for the four datasets. The results show that diff-based candidate extraction is the most critical component, as disabling it leads to the largest decrease in all metrics. This is unsurprised since CodeMapper relies on diff reports for basic line mappings. Without it, the approach relies solely on movement detection, which helps only when the code region actually was moved, and text search, which helps only when the code region is unchanged. Disabling text search leads to a relatively large drop for the suppression study data and the CodeTracker data, with F1-scores decreasing from 0.827 to 0.552 and from 0.879 to 0.594, respectively. This is mainly because text search remains effective at mapping unchanged code regions, such as some suppressions and variables, highlighting its contribution to the overall performance. Disabling the other components also reduces the effectiveness, either on some or all datasets. For example, disabling movement detection significantly reduces recall on Data A, missing seven moved code regions that CodeMapper identifies as an exact match. Another example is that refining candidate regions increases precision, e.g., from 0.816 to 0.889 on Data A, which is exactly what this component is designed for.

We further analyze the contribution of the four git diff algorithms. The results indicate that CodeMapper identifies 2,405 candidates solely due to one of the three non-default algorithms: 1,955 from *patience*, 305 from *histogram*, and 145 from *minimal* (Section 2.3.1). From these 2,405 candidates, phase 2 of our approach selects 74 as target regions, leading to 22 exact matches and 23 partial overlaps. Given the relatively low computational cost of running multiple diff algorithms (Section 3.5), CodeMapper uses all four algorithms to increase the chance of collecting more candidate regions.

Answer to RQ2: Considering contextual lines helps in selecting the most likely target region, and CodeMapper is robust to minor changes of context size. Each component of CodeMapper enhances its overall effectiveness, with the diff-based candidate extraction and text search being the most critical.

3.5 RQ3: Efficiency of CodeMapper

To measure efficiency, we record the execution times of CodeMapper. Figure 12 visualizes the overall execution time, averaged over five runs. In its default configuration, CodeMapper takes 2,327 milliseconds to compute a target region. Because the baselines do not first compute candidate regions and then select the best target region, they are faster, with an average of 1,624 and 1,628 milliseconds for $\text{diff}_{\text{line}}$ and $\text{diff}_{\text{word}}$, respectively. Despite the increased effort performed by CodeMapper, our approach is fast enough for interactive usage. Because smaller context sizes than our default of 15 also provide competitive results (Figure 10), Figure 12 also shows the execution time with smaller context sizes. Reducing the context significantly reduces the execution time, offering users a knob for trading slightly reduced effectiveness for even higher efficiency.

The breakdown of where time is spent, as shown with distinct colors in Figure 12, reveals that CodeMapper takes 1,665 milliseconds to compute candidates for a source region and 661 milliseconds to select a target region. While computing candidates, the approach runs diff eight times (Section 2.3.1), whereas the baselines use only one diff result. The time taken for "target region selection" is mostly due to computing the Levenshtein distances.

Answer to RQ3: CodeMapper takes an average of 2,327 milliseconds to identify a target region, which, despite being slower than the baselines, is sufficiently fast for interactive usage. Reducing the context size further reduces the execution time.

4 Threats to Validity

The primary threat to validity concerns the datasets. Data A is created by the authors, which may introduce bias. To reduce this bias, Data B starts from randomly selected source regions. Each

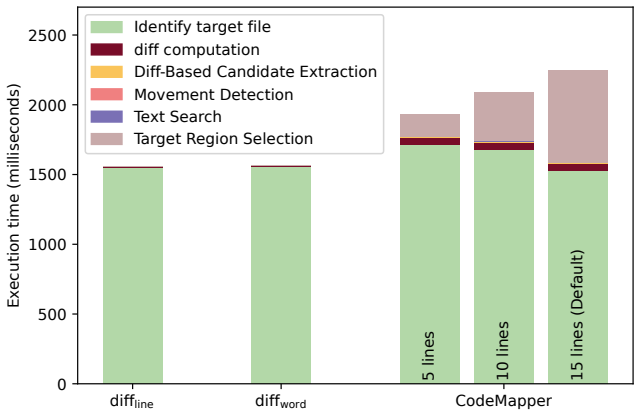


Figure 12: Execution time across the four datasets.

of the 200 examples has been annotated independently by two authors, followed by a discussion to reach consensus. With only 6 and 8 initial disagreements in Data A and B, respectively, we consider the annotation results highly reliable. A related threat is using the suppression study data [16] as ground truth. To mitigate this, we randomly inspect 20 history pairs, all of which we found correct. Another potential threat is the limited size of the first three datasets. We mitigate this by also evaluating CodeMapper on the larger CodeTracker data [12, 17], which was created independently. Finally, our approach uses git diff as a component and baseline, but other diff implementations exist. We selected git diff because it is widely used and represents the state of the art in diff computation.

5 Related Work

Studies of Developer Needs and Behaviors. LaToza et al. [19] conduct a survey to identify questions about code that developers find challenging to answer. Their findings indicate that developers are interested in code histories at the code snippet level, whereas many existing tools require navigating through all changes at the file level to locate specific smaller changes of interest. CodeMapper addresses this gap by enabling developers to map customized code regions from one commit to another. Lin et al. [21] investigate the characteristics of fine-grained change types and find that changes to functions and statements are the most prevalent. Codoban et al. [4] study why developers check program histories. They observe different motivations, which can be categorized into three groups: finding specific commits, being aware of changes, and finding safe points for backtracking. Our approach could help developers in all three categories by providing a more detailed view of code changes.

Tracking of Code Elements. Previous work [11, 12, 17] focuses on tracking specific program elements across version histories. CodeShovel [11] supports Java methods. CodeTracker [17] supports Java methods and variables, while CodeTracker 2.0 [12] extends this to blocks. Our work differs by addressing the code mapping problem, where two commits are given, and the task is to find the target region for a source region. We also present a language-agnostic approach applicable to arbitrary code regions. FinerGit [14] and Historage [13] utilize a finer-grained Git repository structure by reorganizing Java methods into individual files for precise tracking. FinerGit enhances Historage's ability by breaking down method lines into single tokens for token-based diff computation. However, this reorganization is time-consuming and increases disk space usage in large repositories. In contrast, CodeMapper achieves character-level tracking on existing repositories.

Approaches based on Diffs. MergeGen [7] divides line-level conflicts into smaller, more precise conflicts. Language-independent diff techniques [1, 3, 25] operate on textual code representation. HyperDiff [20] computes scalable AST-level diffs over large code histories. Unlike our work, these approaches cannot track specific regions of interest. Matsumoto et al. [23] combine AST structures and line differences to improve diff comprehension. Higo et al. [15] consider copy-and-paste actions to enhance diff understandability. These works emphasize either language independence or fine-grained change capture. In contrast, CodeMapper addresses both by employing text-based diffs and character-level regions.

Reasoning about Code Changes. Etemadi et al. [8] enhance code diffs with runtime information. DiffSearch [9] allows developers to query code changes. B2SFINDER [34] addresses binary-to-source code matching. Wu et al. [33] manage evolving software artifacts using differential facts for unified analysis. LibvDiff [6] identifies software versions by leveraging symbol information and detecting function-level differences. These works emphasize the importance of detailed code analysis and suggest techniques for future enhancements of CodeMapper. Integrating dynamic information may further improve mapping accuracy in future versions of CodeMapper.

Reasoning about Code Line Evolution. Meta-differencing [22] represents source code with abstract syntax information, enabling queries and searches for changes but requires parsing code into an AST and operates only at the line level. Other line-level approaches identify related changes [35], detect change types applied to lines over time [2], and track related code lines across versions [26, 27]. In contrast, CodeMapper maps code regions between commits. Kim et al. [18] surveys matching techniques for cross-version analysis, noting limitations such as applicability only to parsable projects and fixed granularity. Our work addresses these by supporting arbitrary code regions and being language-agnostic.

6 Conclusion

CodeMapper empowers developers to map arbitrary code regions between commits. Given a source region, it combines novel techniques to identify candidates for the target region and selects the most likely one based on code similarity. Like the git diff tool, CodeMapper works across all text-based programming languages. Unlike git diff, our approach allows developers to focus on specific code regions rather than showing all changes at once. Our evaluation across hundreds of code changes in ten languages demonstrates the effectiveness and efficiency of CodeMapper.

Data Availability

https://github.com/sola-st/CodeMapper

Acknowledgments

Supported by the European Research Council (ERC; grant agreements 851895 and 101155832) and by the German Research Foundation (DFG; projects 492507603, 516334526, and 526259073).

References

[1] Muhammad Asaduzzaman, Chanchal K. Roy, Kevin A. Schneider, and Massimiliano Di Penta. 2013. LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM '13)*. IEEE Computer Society, USA, 230–239. doi:10.1109/ICSM.2013.34

[2] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. 2007. Identifying Changed Source Code Lines from Version Repositories. In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*. 14–14. doi:10.1109/MSR.2007.14

[3] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. 2008. Tracking your changes: A language-independent approach. *IEEE software* 26, 1, 50–57.

[4] M. Codoban, S. S. Ragavan, D. Dig, and B. Bailey. 2015. Software history under the lens: A study on why and how developers examine it. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 1–10.

[5] Breanna Devore-McDonald and Emery D. Berger. 2020. Mossad: defeating software plagiarism detection. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 138 (nov 2020), 28 pages. doi:10.1145/3428206

[6] Chaopeng Dong, Siyuan Li, Shouguo Yang, Yang Xiao, Yongpan Wang, Hong Li, Zhi Li, and Limin Sun. 2024. LibvDiff: Library Version Difference Guided OSS Version Identification in Binaries. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 66, 12 pages. doi:10.1145/3597503.3623336

[7] Jinhao Dong, Qihao Zhu, Zeyu Sun, Yiling Lou, and Dan Hao. 2023. Merge Conflict Resolution: Classification or Generation?. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1652–1663. doi:10.1109/ASE56229.2023.00155

[8] Khashayar Etemadi, Aman Sharma, Fernanda Madeiral, and Martin Monperrus. 2023. Augmenting Diffs With Runtime Information. *IEEE Transactions on Software Engineering* 49, 11 (2023), 4988–5007. doi:10.1109/TSE.2023.3324258

[9] Luca Di Grazia, Paul Bredl, and Michael Pradel. 2023. DiffSearch: A Scalable and Precise Search Engine for Code Changes. *IEEE Trans. Softw. Eng.* 49, 4 (apr 2023), 2366–2380. doi:10.1109/TSE.2022.3218859

[10] Luca Di Grazia and Michael Pradel. 2022. The Evolution of Type Annotations in Python: An Empirical Study. In *ESEC/FSE*.

[11] Felix Grund, Shaiful Alam Chowdhury, Nick Bradley, Braxton Hall, and Reid Holmes. 2021. CodeShovel: Constructing Method-Level Source Code Histories. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[12] Mohammed Tayeeb Hasan, Nikolaos Tsantalis, and Pouria Alikhanifard. 2024. Refactoring-aware Block Tracking in Commit History. *IEEE Transactions on Software Engineering* (2024), 1–20. doi:10.1109/TSE.2024.3484586

[13] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. 2011. Historage: fine-grained version control system for Java. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution* (Szeged, Hungary) *(IWPSE-EVOL '11)*. Association for Computing Machinery, New York, NY, USA, 96–100. doi:10.1145/2024445.2024463

[14] Yoshiki Higo, Shinpei Hayashi, and Shinji Kusumoto. 2020. On Tracking Java Methods with Git Mechanisms. *Journal of Systems and Software* 165 (2020), 13 pages. doi:10.1016/j.jss.2020.110571

[15] Yoshiki Higo, Akio Ohtani, and Shinji Kusumoto. 2017. Generating simpler ast edit scripts by considering copy-and-paste. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 532–542.

[16] Huimin Hu, Yingying Wang, Julia Rubin, and Michael Pradel. 2025. An Empirical Study of Suppressed Static Analysis Warnings. In *FSE*.

[17] Mehran Jodavi and Nikolaos Tsantalis. 2022. Accurate Method and Variable Tracking in Commit History. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 183–195. doi:10.1145/3540250.3549079

[18] Miryung Kim and David Notkin. 2006. Program element matching for multi-version program analyses. In *Proceedings of the 2006 International Workshop on Mining Software Repositories* (Shanghai, China) *(MSR '06)*. Association for Computing Machinery, New York, NY, USA, 58–64. doi:10.1145/1137983.1137999

[19] Thomas D. LaToza and Brad A. Myers. 2010. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools* (Reno, Nevada) *(PLATEAU '10)*. Association for Computing Machinery, New York, NY, USA, Article 8, 6 pages. doi:10.1145/1937117.1937125

[20] Quentin Le Dilavrec, Djamel Eddine Khelladi, Arnaud Blouin, and Jean-Marc Jézéquel. 2023. HyperDiff: Computing Source Code Diffs at Scale. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (<conf-loc>, <city>San Francisco</city>, <state>CA</state>, <country>USA</country>, </conf-loc>) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 288–299. doi:10.1145/3611643.3616312

[21] Wei Lin, Zhifei Chen, Wanwangying Ma, Lin Chen, Lei Xu, and Baowen Xu. 2016. An Empirical Study on the Characteristics of Python Fine-Grained Source Code Change Types. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 188–199. doi:10.1109/ICSME.2016.25

[22] Jonathan I. Maletic and Michael L. Collard. 2004. Supporting Source Code Difference Analysis. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM '04)*. IEEE Computer Society, USA, 210–219.

[23] Junnosuke Matsumoto, Yoshiki Higo, and Shinji Kusumoto. 2019. Beyond GumTree: a hybrid approach to generate edit scripts. In *Proceedings of the 16th International Conference on Mining Software Repositories* (Montreal, Quebec, Canada) *(MSR '19)*. IEEE Press, 550–554. doi:10.1109/MSR.2019.00082

[24] Matija Novak, Mike Joy, and Dragutin Kermek. 2019. Source-code Similarity Detection and Detection Tools Used in Academia: A Systematic Review. *ACM Trans. Comput. Educ.* 19, 3, Article 27 (may 2019), 37 pages. doi:10.1145/3313290

[25] Steven P. Reiss. 2008. Tracking source locations. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 11–20. doi:10.1145/1368088.1368091

[26] Francisco Servant and James A. Jones. 2012. History slicing: assisting code-evolution tasks. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 43, 11 pages. doi:10.1145/2393596.2393646

[27] Francisco Servant and James A. Jones. 2017. Fuzzy fine-grained code-history analysis. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 746–757. doi:10.1109/ICSE.2017.74

[28] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F. Bissyandé. 2021. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 981–992. doi:10.1145/3324884.3416532

[29] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* 48, 3 (2022), 930–950. doi:10.1109/TSE.2020.3007722

[30] Stack Overflow User. 2012. Git: Show all of the various changes to a single line in a specified file over the entire git history. https://stackoverflow.com/questions/9935379/git-show-all-of-the-various-changes-to-a-single-line-in-a-specified-file-over-t

[31] Stack Overflow User. 2016. git: grep file in all previous versions. https://stackoverflow.com/questions/34576699/git-grep-file-in-all-previous-versions?noredirect=1

[32] Stack Overflow User. 2016. How can I use git log and only output the matching lines? https://stackoverflow.com/questions/40936797/how-can-i-use-git-log-and-only-output-the-matching-lines?rq=3

[33] Xiuheng Wu, Chenguang Zhu, and Yi Li. 2021. DIFFBASE: a differential factbase for effective software evolution management. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 503–515. doi:10.1145/3468264.3468605

[34] Zimu Yuan, Muyue Feng, Feng Li, Gu Ban, Yang Xiao, Shiyang Wang, Qian Tang, He Su, Chendong Yu, Jiahuan Xu, Aihua Piao, Jingling Xuey, and Wei Huo. 2019. B2SFinder: Detecting Open-Source Software Reuse in COTS Software. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1038–1049. doi:10.1109/ASE.2019.00100

[35] Thomas Zimmermann, Sunghun Kim, Andreas Zeller, and E. James Whitehead. 2006. Mining version archives for co-changed lines. In *Proceedings of the 2006 International Workshop on Mining Software Repositories* (Shanghai, China) *(MSR '06)*. Association for Computing Machinery, New York, NY, USA, 72–75. doi:10.1145/1137983.1138001