ActRef: Enhancing the Understanding of Python Code Refactoring with Action-Based Analysis

SIQI WANG, Zhejiang University, China
XING HU*, Zhejiang University, China
XIN XIA, Zhejiang University, China
XINYU WANG, Zhejiang University, China

Refactoring, the process of improving the code structure of a software system without altering its behavior, is crucial for managing code evolution in software development. Identifying refactoring actions in source code is essential for understanding software evolution and guiding developers in maintaining and improving the code quality. However, existing tools frequently struggle to detect complex and diverse refactoring types in Python, because most of them are designed for Java-centric environments. This limitation hinders their ability to address Python's dynamic and flexible syntax effectively. This study presents an action-based Refactoring Analysis Framework named ActRef, a novel algorithm designed to advance the detection and understanding of Python refactorings through a unique code change action-based analysis of code changes. ActRef mining multiple refactoring types (e.g., move, rename, extract, and inline operations) based on diff actions, covering multiple granularity levels including variable, method, class, and module levels. By focusing on the code change actions, ActRef provides a Python-adaptive solution to detect intricate refactoring patterns. Our evaluation, conducted on 1,914 manually validated refactoring instances from 136 open-source Python projects. The evaluation results show that ActRef achieves high precision (0.80) and recall (0.92), effectively identifying multiple refactoring types. Compared with leading baselines, including PyRef, PyRef with MLRefScanner, DeepSeek-R1 and ChatGPT-4, ActRef consistently demonstrates superior performance in detecting Python refactorings across various types. While matching PyRef in runtime efficiency, ActRef supports a broader spectrum of refactoring types and more refactoring mining levels. ActRef shows an effective and scalable approach for mining refactorings in dynamic Python codebases and introduces a new perspective on understanding code.

Additional Key Words and Phrases: Refactoring, Software Maintenance, Python

ACM Reference Format:
Siqi Wang, Xing Hu, Xin Xia, and Xinyu Wang. 2025. ActRef: Enhancing the Understanding of Python Code Refactoring with Action-Based Analysis. 1, 1 (June 2025), 21 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Refactoring is a crucial practice in software development that involves the restructuring of existing code without changing its external behavior [8, 15]. As software systems grow in scale and complexity, refactoring becomes indispensable for reducing technical debt, improving code readability, reducing complexity, enhancing maintainability, and sustaining long-term software

*Corresponding Author

Authors' addresses: Siqi Wang, Zhejiang University, Hangzhou, Zhejiang, China, siqiwang@zju.edu.cn; Xing Hu, Zhejiang University, Hangzhou, Zhejiang, China, xinghu@zju.edu.cn; Xin Xia, Zhejiang University, Hangzhou, Zhejiang, China, xin.xia@acm.org; Xinyu Wang, Zhejiang University, Hangzhou, Zhejiang, China, wangxinyu@zju.edu.cn.

XXXX-XXXX/2025/6-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

evolution [4, 5, 17, 18, 21]. Beyond these immediate benefits, analyzing refactoring practices provides deeper insights into software maintenance, selecting regression tests, good design principles, and patterns of code evolution over time [9, 11, 27, 28, 36].

However, identifying and understanding refactoring operations (e.g., Inline Variable and Extract Class) remains a challenging task. Refactorings are rarely documented systematically by developers [3] and are often entangled with other code changes [31]. As a result, it becomes difficult to separate refactorings from unrelated changes and to understand their individual impact. Manual identification of refactorings in large-scale projects is time-consuming and error-prone [16], further complicating efforts to understand their motivations and impact. To overcome these limitations, several automated tools (e.g., RefactoringMiner [34, 35], Ref-Finder [20], and RefDiff [30, 32]) are widely used to mine refactoring operations from software projects. These tools have proven effective in statically typed languages such as Java, where type information and rigid syntax help identify structural code changes with high precision.

However, these techniques are not directly applicable to dynamically typed languages like Python [22], which is the most widely used programming language [33]. To support Python refactoring detection, two tools have been proposed: Python-Adaptive-RefactoringMiner [12] and PyRef [7], which are both inspired by the Java-based RefactoringMiner. Python-Adaptive-RefactoringMiner translates Python code AST into Java and applies RefactoringMiner's original rules, but this translation can lose Python-specific semantics and is supported by inferred type information. PyRef avoids translation and applies Java-derived matching mechanisms directly to Python syntax trees. Despite their differences, both tools rely heavily on statement-matching, which aligns entire code statements across versions to detect refactorings. Statement-matching is built on the assumption that code structure remains stable across changes and that type information helps disambiguate code elements. While this assumption often holds in statically typed languages, it breaks down in Python, where developers commonly make fine-grained edits within a single statement—for example, modifying a nested expression or changing the structure of a list comprehension [23, 41]. In such cases, statement-matching fails to match fine-grain changed statements, leading to missed or incorrect refactoring detection.

To address these limitations, we propose ActRef, a multi-granularity action-based approach for detecting refactoring in Python code. Instead of aligning full statements, ActRef detects refactorings by analyzing sequences of fine-grained code change actions. Specifically:

- To address the fragility of statement-level matching in dynamic languages, ActRef builds on and extends GumTree to compute type-agnostic AST-level actions—insertions, deletions, moves, and updates—that tolerate fine-grained edits while preserving semantic intent.
- To reduce inaccuracies caused by incorrect AST mappings, ActRef incorporates a context-aware post-processing step that refines actions using syntax-aware heuristics and signature consistency.
- To support coarse-grained refactorings' detection, ActRef calculates module-level actions that integrate semantic similarity metrics and code slicing techniques across files, allowing it to track code movement and reuse at the module level.

Different from existing tools, ActRef does not rely on static type information or rigid structure assumptions, which makes it better equipped to handle the flexible features of Python. This makes it highly applicable to dynamic, type-agnostic environments commonly found in Python projects. Building on this flexibility, ActRef provides detailed insights into how code is refactored, offering developers and researchers a structured way to observe, analyze, and understand patterns of code change. By interpreting sequences of code change actions using structured refactoring rules, ActRef more closely aligns with a developer's perspective, offering a nuanced approach to uncovering refactoring operations. This action-based method is particularly adept at capturing complex code modification patterns, making it well-suited for dynamic and evolving projects. ActRef's ability to track complex code adjustments provides critical insights into the evolution and refinement of Python code, especially valuable in fields like machine learning and data science, where code changes are rapid and complex.

To evaluate the effectiveness of ActRef, we compare it with state-of-the-art tools, including PyRef [7], PyRef with MLRefScanner [24]. We also compare ActRef with two popular Large Language Models (LLMs): DeepSeek-R1 [10] and ChatGPT-4 [25]. We evaluate ActRef on a custom-constructed dataset, developed to represent a range of real-world Python refactorings by manually analyzing and addressing limitations found in existing datasets. Our analysis demonstrates that ActRef significantly outperforms these baselines in Python refactoring detection across diverse refactoring types, establishing its value for complex, real-world projects. Moreover, ActRef's runtime performance matches PyRef's efficiency, while offering a wider range of supported refactorings and enhanced detection granularity. These results underscore ActRef's practicality and scalability, presenting it as a robust tool for developers in industry settings. Our paper makes the following contributions:

- We propose ActRef, an action-based refactoring detection approach specifically designed for Python. ActRef proposes a specialized Python refactoring detection strategy, achieving greater performance than state-of-the-art tools.
- To evaluate ActRef's effectiveness, we extended a manually analyzed dataset representing a diverse set of real-world Python refactorings, which can be found in our replication package [1]. This dataset addresses the limitations of existing datasets by capturing complex, real-world refactoring cases, which enhances the relevance and rigor of our evaluation.

Paper Organization: Section 2 describes the motivation and terminologies of our research. Section 3 describes the methodology of our research. Section 4 describes the evaluation of our search. Section 5 shows the results of our study. Section 6 discusses the implications and threats of our results. Section 7 discusses related work. Section 8 concludes our study and mentions future work.

2 PRELIMINARY STUDY

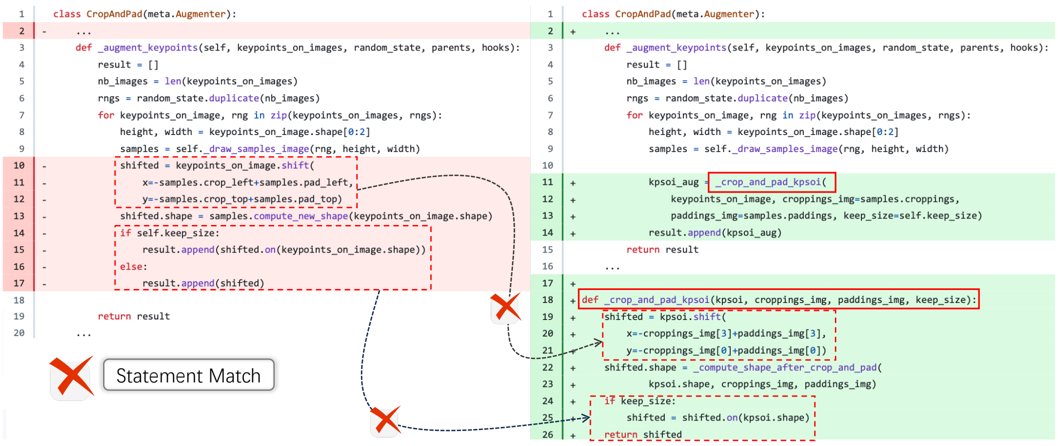
In this section, we explain the motivation and terminologies used in our paper.

2.1 Motivation

Detecting refactoring in Python code presents unique challenges due to Python's dynamic typing and its flexible syntax. Figure 1 illustrates this challenge with an *Extract Method* example from the `imgaug` [2] project, and Figure 2 provides its corresponding Abstract Syntax Tree (AST).

In this *Extract Method* example, part of the logic in the `_augment_keypoints` method (lines 10–17) is moved into a newly created method `_crop_and_pad_kpsoi`. Existing statement-based tools fail to detect this refactoring due to significant syntax changes and fine-grained variations in the code. The extracted logic introduces a new method with additional parameters, modifies variable assignments, and alters the return mechanism, resulting in statements' structural differences that obscure the relationship between the original and the new code. Moreover, the handling of variables, such as replacing assignments to `shifted` with direct returns, further highlights the superficial syntactic differences while masking the underlying logical consistency and intent of the developer.

This consistency becomes evident when analyzing the corresponding AST shown in Figure 2. Although some AST nodes change during the refactoring, portions of the subtree structure remain consistent, reflecting the underlying logical relationship between the original and the extracted code. From the perspective of AST tree-edit actions, the extraction of logic into a new method can be represented as a combination of "delete" subtree actions (removing the logic from the



```
class CropAndPad(meta.Augmenter):
-    ...
    def _augment_keypoints(self, keypoints_on_images, random_state, parents, hooks):
        result = []
        nb_images = len(keypoints_on_images)
        rngs = random_state.duplicate(nb_images)
        for keypoints_on_image, rng in zip(keypoints_on_images, rngs):
            height, width = keypoints_on_image.shape[0:2]
            samples = self._draw_samples_image(rng, height, width)
-           shifted = keypoints_on_image.shift(
-               x=-samples.crop_left+samples.pad_left,
-               y=-samples.crop_top+samples.pad_top)
-           shifted.shape = samples.compute_new_shape(keypoints_on_image.shape)
-           if self.keep_size:
-               result.append(shifted.on(keypoints_on_image.shape))
-           else:
-               result.append(shifted)

        return result
    ...
```

```
class CropAndPad(meta.Augmenter):
+    ...
    def _augment_keypoints(self, keypoints_on_images, random_state, parents, hooks):
        result = []
        nb_images = len(keypoints_on_images)
        rngs = random_state.duplicate(nb_images)
        for keypoints_on_image, rng in zip(keypoints_on_images, rngs):
            height, width = keypoints_on_image.shape[0:2]
            samples = self._draw_samples_image(rng, height, width)

+           kpsoi_aug = _crop_and_pad_kpsoi(
+               keypoints_on_image, croppings_img=samples.croppings,
+               paddings_img=samples.paddings, keep_size=self.keep_size)
+           result.append(kpsoi_aug)
        return result
    ...
+
+def _crop_and_pad_kpsoi(kpsoi, croppings_img, paddings_img, keep_size):
+   shifted = kpsoi.shift(
+       x=-croppings_img[3]+paddings_img[3],
+       y=-croppings_img[0]+paddings_img[0])
+   shifted.shape = _compute_shape_after_crop_and_pad(
+           kpsoi.shape, croppings_img, paddings_img)
+   if keep_size:
+       shifted = shifted.on(kpsoi.shape)
+   return shifted
```

Fig. 1. An Extract Method Example from **imgaug**

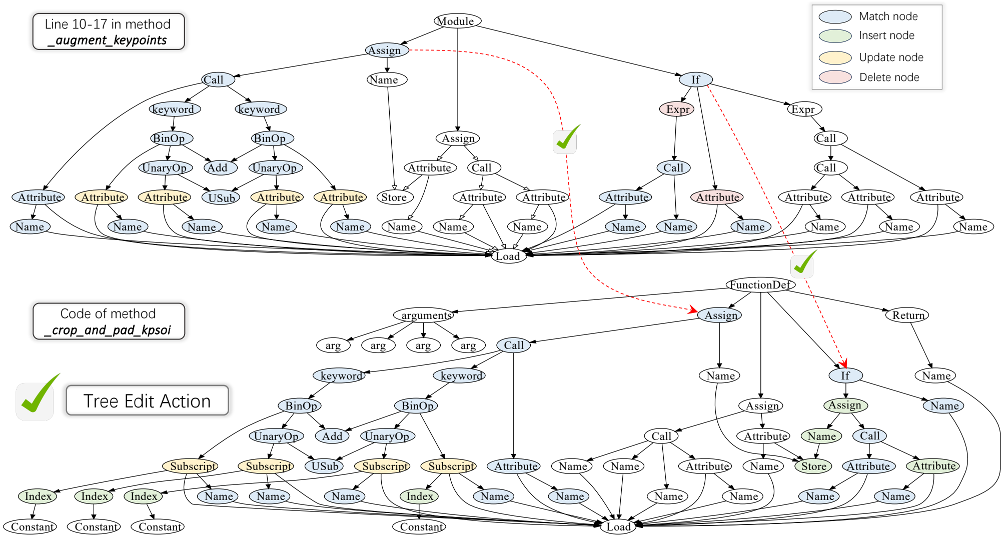


Fig. 2. An Extract Method Example AST from imgaug

original method) and "insert" subtree actions (adding the new method). These actions also capture changes to method signatures, such as parameter additions and return value handling, while preserving the structural flow of the code. This example highlights the limitations of existing tools and emphasizes the importance of analyzing structural consistency through ASTs. Different from statement-matching approaches, which rely heavily on surface-level syntax, the action-based analysis captures the intent behind code changes by leveraging tree-edit operations.

2.2 Code Elements

We extract the needed code elements from AST, including the following elements:

- **Module:** A module is defined as a Python file, containing classes and methods, and directly affiliated definitions and statements (i.e., those not defined within a method).

- **Class:** A class is a blueprint for creating objects, encapsulating methods, statements, and variables. It supports object-oriented features such as inheritance, encapsulation, and polymorphism. Classes are defined with the '`class`' keyword in Python.
- **Method:** A method is a reusable block of code associated with a class or module, typically performing a specific operation. It consists of a declaration, an optional list of parameters, and a suit of executable statements. Methods are defined with the '`def`' keyword in Python.
- **Statement:** A statement is the smallest executable unit in Python, representing a single action or command. It is terminated by a line break or a semicolon and can be categorized into simple statements (e.g., assignments and expressions) or compound statements (e.g., conditionals, loops, and function definitions).
- **Variable:** A variable is an identifier used to store data in memory. It is the smallest nameable storage element in a Python program, capable of referencing various data types such as integers, strings, or objects.

2.3 Action

In code refactoring detection, **action** represents the atomic changes that occur during the modification of code elements. These actions serve as the foundation for understanding how code evolves during refactoring, capturing both structural and semantic intentions behind each change. Rather than relying solely on statement-matching or token diffing, our method computes multi-level actions that integrate both fine-grained and coarse-grained perspectives on code changes. To capture these actions, we design a multi-level action modeling strategy based on the granularity of code transformation:

- **AST-level actions:**
  For fine-grained changes, we analyze differences between the abstract syntax trees (ASTs) of the original and modified files. These actions are enriched with contextual and semantic information, including the signatures of methods or classes and their surrounding code structure. The output consists of atomic operations such as:
  - $\textbf{delete}(d)$: delete code element d.
  - $\textbf{insert}(i, n)$: insert code element i at position n.
  - $\textbf{move}(m, o, n)$: move code element m from position o to n.
  - $\textbf{update}(o, n)$: change the code value from o to n.

  We build our AST-level action calculator on top of GumTree [13, 14], a widely used AST differencing engine, and enhance it with semantic-aware extensions that consider code signatures, usage context, and cross-file coordination. This action set provides the foundation for detecting intra-file and cross-file fine-grained refactorings (class/method/variable level).
- **Module-level actions:** For coarse-grained code transformations (e.g., moving or extracting large code blocks across modules), we introduce *module-level actions*. We build our module-level action calculator combining program slicing and similarity to overcome the limitations of AST matching in scenarios involving entire file additions, deletions, or large-scale relocations. We define the following atomic operations at the coarse-grained refactoring (module level):
  - $\textbf{insert}(s, f)$: code slice s is newly added to file f.
  - $\textbf{delete}(s, f)$: code slice s is deleted from file f.
  - $\textbf{move}(s, f_o, f_n)$: code slice s is moved from f_o to f_n.

By analyzing both fine-grained AST-level and coarse-grained module-level actions, our multi-level action analysis framework enables a broader and more accurate coverage of refactoring detection. This design not only improves fidelity to developer behavior but also enhances the robustness of refactoring detection in dynamic languages such as Python.

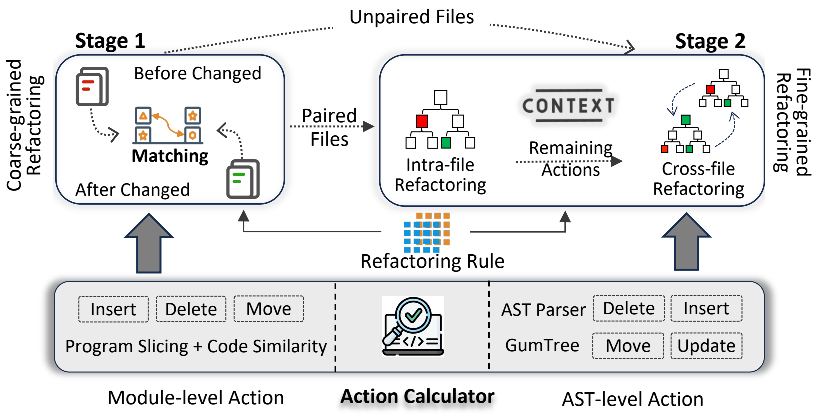


Fig. 3. Methodology Overview

3 METHODOLOGY

In this section, we introduce an action-based approach to detect code refactoring operations. Unlike existing methods that rely on statement-level matching, ActRef focuses on action, which can indicate the developer's intent, treating changes as purposeful operations such as "*moving code to extract a new method*". By capturing refactorings from this perspective, ActRef offers a more intuitive and semantically meaningful view of code evolution.

3.1 Overall Framework

Figure 3 shows the overview framework of our approach ActRef. By analysing Sets of files before and after a commit, we detect both coarse-grained refactorings (module level) and fine-grained refactorings (class/method/variable level) in this commit. It involves decomposing code modifications into a series of actions, using staged refactoring mining to identify refactoring. We employ a model-level action calculator and an AST-level action calculator to get the code change actions. By further matching these actions with predefined refactoring rules, it is possible to accurately identify 15 types of refactoring operations (e.g., Extract Variable, Rename Module, Move Method). This section provides a comprehensive overview of the algorithmic design, detection rules, and the rationale behind ActRef.

As shown in algorithm 1, the input of ActRef is two sets of files, designated as $\mathcal{F}^-$ and $\mathcal{F}^+$, which represent the state before and after the code is modified. We first detect the coarse-grained refactoring, the function `ModuleLevelRefactoringDetection` is responsible for pairing up files and detecting module-level refactorings from $\mathcal{F}^-$ and $\mathcal{F}^+$ based on their names, paths, and code similarity. It returns four sets: (1) $\mathcal{P}$: The set of paired files, (2) $\mathcal{D}_{\text{unpaired}}$: Deletions that exist only in $\mathcal{F}^-$ (deleted files), (3) $\mathcal{I}_{\text{unpaired}}$: Insertions that exist only in $\mathcal{F}^+$ (newly added files) and (4) $\mathcal{R}_{\text{module}}$: module-level refactorings. Then, we detect the fine-grained refactorings. We generate the intra-file code change actions using AST-level action calculator and process these actions to mining refactoring operations. Subsequently, the unmatched insert and delete actions, which have not been calculated in the intra-file detection step, are passed to the cross-file detection step. Finally, we combine the remaining actions and unpaired files, and generate the cross-file code change actions using AST-level action calculator to detect cross-file refactorings.

Algorithm 1: Overall Algorithm of Refactoring Detection

```
Input: $\mathcal{F}^-$, $\mathcal{F}^+$: Sets of files before and after a commit
Output: $\mathcal{R}$: Detected Refactorings
$\mathcal{R} \leftarrow \emptyset$ ;                                   // Initialize refactoring set
$\mathcal{I}_{\text{unpaired}} \leftarrow \emptyset$, $\mathcal{D}_{\text{unpaired}} \leftarrow \emptyset$ ;   // Unpaired insert/delete actions for cross-file matching
// Step 1: Coarse-grained refactoring detection
$\mathcal{P}, \mathcal{D}_{\text{unpaired}}, \mathcal{I}_{\text{unpaired}}, \mathcal{R}_{\text{module}} \leftarrow$ ModuleLevelRefactoringDetection($\mathcal{F}^-, \mathcal{F}^+$)
// Step 2: Fine-grained refactoring detection
for $(f_i^-, f_i^+) \in \mathcal{P}$ do
    $\mathcal{R}_{\text{intra}}, Action_i^{\text{rem}} \leftarrow$ IntraFileRefactoringDetection($f_i^-, f_i^+$);
$Action_{\text{remaining}} \leftarrow Action_1^{rem} \cup \cdots \cup Action_n^{rem}$
$\mathcal{R}_{\text{cross}} \leftarrow$ CrossFileRefactoringDetection($\mathcal{I}_{\text{unpaired}}, \mathcal{D}_{\text{unpaired}}, Action_{\text{remaining}}$);
$\mathcal{R} \leftarrow \mathcal{R}_{\text{intra}} \cup \mathcal{R}_{\text{cross}} \cup \mathcal{R}_{\text{module}}$;
return $\mathcal{R}$;
```

Algorithm 2: Module-level Refactoring Detection

```
Input: $\mathcal{F}^-$, $\mathcal{F}^+$: Sets of files before and after a commit
Output: $\mathcal{R}_{\text{module}}$: Detected module-level refactorings
$\mathcal{R}_{\text{module}} \leftarrow \emptyset$
$\mathcal{P} \leftarrow \emptyset$, $\mathcal{D}_{\text{unpaired}} \leftarrow \emptyset$, $\mathcal{I}_{\text{unpaired}} \leftarrow \emptyset$
// Step 1: Match files using name, path, content similarity
for $f_b \in \mathcal{F}^-, f_a \in \mathcal{F}^+$ do
    if is_same_name($f_b, f_a$) then
        $\mathcal{P} \leftarrow \mathcal{P} \cup (f_b, f_a)$
    else if $(f_a, f_b)$ can match rule in Table1 then
        $\mathcal{R}_{\text{module}} \leftarrow \mathcal{R}_{\text{module}} \cup$ corresponding refactoring;
        $\mathcal{P} \leftarrow \mathcal{P} \cup (f_b, f_a)$
    else
        $\mathcal{D}_{\text{unpaired}} \leftarrow f_b$ , $\mathcal{I}_{\text{unpaired}} \leftarrow f_a$;
// Step 2: Detect Extract/Inline Module
for $f_a \in \mathcal{I}_{unpaired}$ do
    if $f_a$ can match rule in Table1 then
        $\mathcal{R}_{\text{module}} \leftarrow \mathcal{R}_{\text{module}} \cup$ ExtractModule($f_a$);
        Remove $f_a$ from $\mathcal{I}_{\text{unpaired}}$
for $f_b \in \mathcal{D}_{unpaired}$ do
    if $f_b$ can match rule in Table1 then
        $\mathcal{R}_{\text{module}} \leftarrow \mathcal{R}_{\text{module}} \cup$ InlineModule($f_b$);
        Remove $f_b$ from $\mathcal{D}_{\text{unpaired}}$
return $\mathcal{R}_{module}, \mathcal{D}, \mathcal{D}_{unpaired}, \mathcal{I}_{unpaired}$
```

3.2 Coarse-grained Refactoring Detection

Algorithm 2 aims to detect coarse-grained refactorings. The input to this procedure is the complete set of files before ($\mathcal{F}^-$) and after ($\mathcal{F}^+$) a commit. The algorithm proceeds in two main stages: The first step iterate over all possible pairs (f_b, f_a), where $f_b \in \mathcal{F}^-$ and $f_a \in \mathcal{F}^+$. If the two files share the same name and path, they are directly paired and added to the matched file set $\mathcal{P}$. Otherwise, we check whether the pair (f_b, f_a) satisfies any module-level transformation rules defined in Table 1 (e.g., Rename Module, Move Module). If a rule matches, the corresponding refactoring is recorded and the pair is also marked as matched. Files that do not satisfy these rules are considered unmatched and added to the sets $\mathcal{D}_{\text{unpaired}}$ and $\mathcal{I}_{\text{unpaired}}$. The second step capture extract and inline module patterns, we analyze the unmatched files collected in the previous step. For each unpaired file, we check whether it satisfies Extract Module or Inline Module rule. If such a match exists, we add the corresponding refactoring operation to $\mathcal{R}_{\text{module}}$ and remove this file from unpaird file set.

Table 1. Detection Rules for Refactoring Types

Refactoring	**Rule**
Extract Module	$\exists f \in \mathcal{F}^+, \text{slices}(f) = s_1, \ldots, s_n, \forall s_i \mid is\_move\_silce(s_i)$
Extract Class	(1) $\exists d \in$ delete_actions, $i \in$ insert_actions, $c \in$ insert_actions | is_similar(body(i), d) $\wedge$ is_class(c) (2) $\exists m \in$ move_actions, $i \in$ insert_actions, $c \in$ insert_actions | is_class(c) $\wedge$ same_name(i, c) $\wedge$ is_body(m, c)
Extract Method	(1) $\exists d \in$ delete_actions, $i \in$ insert_actions, $c \in$ insert_actions | is_similar(body(i), d) $\wedge$ is_method(i) (2) $\exists m \in$ move_actions, $i \in$ insert_actions, $c \in$ insert_actions | is_method(i) $\wedge$ same_name(i, c)
Extract Variable	(1) $\exists d \in$ delete_actions, $i \in$ insert_actions, $c \in$ insert_actions | is_similar(RHS(i), d) $\wedge$ is_variable(i) (2) $\exists m \in$ move_actions, $i \in$ insert_actions, $c \in$ insert_actions | is_expression(m)
Move Module	$\exists f_b \in \mathcal{F}^-, f_a \in \mathcal{F}^+ \mid (f_b, f_a) \notin \mathcal{P} \wedge$ is_similar(f_a, f_b) $\wedge$ different_path(f_b, f_a)
Move Class	(1) $\exists m \in$ move_actions | is_class(m) (2) $\exists d \in$ delete_actions, $i \in$ insert_actions | is_class(d) $\wedge$ is_class(i) $\wedge$ is_similar(d, i)
Move Method	(1) $\exists m \in$ move_actions | is_method(m) (2) $\exists d \in$ delete_actions, $i \in$ insert_actions | is_method(d) $\wedge$ is_method(i) $\wedge$ is_similar(d, i)
Inline Module	$\exists f \in \mathcal{F}^-, \text{slices}(f) = s_1, \ldots, s_n, \forall s_i \mid is\_move\_silce(s_i)$
Inline Class	(1) $\exists d \in$ delete_actions, $c \in$ delete_actions, $i \in$ insert_actions | is_similar(body(d), i) $\wedge$ is_class(d) (2) $\exists m \in$ move_actions, $d \in$ delete_actions, $c \in$ delete_actions | is_class(d) $\wedge$ same_name(d, c)
Inline Method	(1) $\exists d \in$ delete_actions, $c \in$ delete_actions, $i \in$ insert_actions | is_similar(body(d), i) $\wedge$ is_method(d) (2) $\exists m \in$ move_actions, $d \in$ delete_actions, $c \in$ delete_actions | is_method(d) $\wedge$ same_name(d, c)
Inline Variable	(1) $\exists d \in$ delete_actions, $i \in$ insert_actions, $c \in$ delete_actions | is_similar(RHS(d), i) $\wedge$ is_variable(i) (2) $\exists m \in$ move_actions, $d \in$ delete_actions, $c \in$ delete_actions | is_expression(m)
Rename Module	$\exists f_b \in \mathcal{F}^-, f_a \in \mathcal{F}^+ \mid (f_b, f_a) \notin \mathcal{P} \wedge$ is_similar(f_a, f_b) $\wedge$ different_name(f_b, f_a)
Rename Class	$\exists u \in$ update_actions $\wedge$ is_class_name(u)
Rename Method	$\exists u \in$ update_actions $\wedge$ is_method_name(u)
Rename Variable	$\exists u \in$ update_actions $\wedge$ is_expression_LHS(u)

* LHS: the left hand side of an expression statement.

3.3 Fine-grained Refactoring Detection

Algorithm 3 and Algorithm 4 aim to detect fine-grained refactorings.

Algorithm 3: Intra-file Refactoring Detection

```
Input: $f_i^-, f_i^+$ ;                                        // Matched file pair
Output: $\mathcal{R}_{\text{intra}}$: Detected refactorings
1  $\mathcal{R}_{\text{intra}} \leftarrow \emptyset, Action_i \leftarrow \emptyset, Acions_i^{\text{rem}}$ ;   // Initialize refactoring and action sets
   // Step 1: Action Generation
2  $AST_i^- \leftarrow$ GenerateAST($f_i^-$)
3  $AST_i^+ \leftarrow$ GenerateAST($f_i^+$)
4  $Action_i \leftarrow$ ActionCalculation($AST_i^-, AST_i^+$)
   // Step 2: Intra-file Refactoring Detection:
5  for $\alpha \in Action_i$ do
6      if $\alpha$ is move or update then
7          if $\alpha$ can match rule in Table 1 then
8              $\mathcal{R}_{\text{intra}} \leftarrow \mathcal{R}_{\text{intra}} \cup$ corresponding refactoring;
9              Remove $\alpha$ from $Action_i$;
10 $Actions_i^{\text{rem}} \leftarrow Actions_i$
11 return $R_{intra}, Actions_i^{rem}$
```

3.3.1 Intra-file Refactoring Detection. Algorithm 3 aims to detect intra-file refactorings based on fine-grained edit actions between two versions of the same file. The input to the algorithm is a pair of files, designated as f_{before} and f_{after}, representing the files before and after the change. The initial step (lines 2-3) generates an abstract syntax tree of the input file and entails a comparison of the abstract syntax trees of the preceding and subsequent versions to calculate specific change actions. These actions encompass the *insert*, *delete*, *move*, and *update* of code. After obtaining the set of edit actions, we match them against a predefined set of refactoring rules as shown in Table 1. Each rule defines a structural pattern of actions that together constitute a recognizable refactoring operation. Once a rule is successfully matched, we add the corresponding refactoring operation r to the set R_{intra}, and remove all actions involved in the match from the working action set to avoid duplication in subsequent rule matches.

Algorithm 4: Cross-file Refactoring Detection

```
Input: $\mathcal{D}_{\text{unpaired}}, \mathcal{I}_{\text{unpaired}}, Acions_{\text{remaining}}$
Output: $\mathcal{R}_{\text{cross}}$
1  $\mathcal{R}_{\text{cross}} \leftarrow \emptyset$ ;                         // Initialize set for cross-file detection
2  for $f_{del} \in \mathcal{D}_{unpaired}$ do
3      Convert $f_{\text{del}}$ to a delete action and add it to $Actions_{\text{remaining}}$ ;
4  foreach $f_{add} \in \mathcal{I}_{unpaired}$ do
5      Convert $f_{\text{add}}$ to an add action and add it to $Actions_{\text{remaining}}$ ;
6  for pair of delete action $\alpha_{del} \in Acions_{remaining}$ and add action $\alpha_{add} \in Actions_{remaining}$ do
7      Extract code snippets $C_{\text{del}}$ from $\alpha_{\text{del}}$ and $C_{\text{add}}$ from $\alpha_{\text{add}}$ ;
8      Compute actions between $C_{\text{del}}$ and $C_{\text{add}}$ using GumTree ;
9      if actions satisfy refactoring rules then
10         Add the detected refactoring to $R_{\text{cross}}$ ;
11         Remove matched actions from $Actions_{\text{remaining}}$ ;
12 return $R_{cross}$
```

3.3.2 Cross-file Refactoring Detection. Algorithm 4 outlines the algorithm for cross-file refactoring detection, which handles code changes across file boundaries. The input of cross-file refactoring detection includes unpaired files and remaining actions. For the unpaired deleted file set *D_unpaired* (lines 2-3), each file is transformed into a deletion action and incorporated into the set of remaining actions. Similarly, the unpaired add file set *I_unpaired* (lines 4-5) is treated analogously, with each file converted to an add action and processed accordingly. The central processing step (lines 6-11) entails an iterative process involving the remaining pairs of delete and add actions. For each pair of actions (lines 7-8), the algorithm extracts the pertinent code segments and employs the AST-level action calculator to get the precise actions between the two code segments. If the actions align with the predefined refactoring rules (line 9) shown in table 1, they are identified as a valid cross-file refactoring, and this refactoring is incorporated into *R_cross* (line 10). Upon completion of the aforementioned steps, the matched actions are removed from the action set (line 11) to avoid any potential for duplicate processing.

4 EVALUATION

4.1 Baselines

To evaluate effectiveness of ActRef in detecting Python refactorings, we compare it with the following baseline tools: **PyRef** [7], **PyRef** with **MLRefScanner** [24], **DeepSeek-R1** [10], and **ChatGPT-4** [25]. PyRef is the state-of-the-art rule-based refactoring detection tool, specifically from RefactoringMiner2 [34] adapted for Python, which supports several key refactoring types (e.g., Move Method, Inline Method, and Rename Class). MLRefScanner is a classifier to detect python refactoring commits, which can ensemble with PyRef. On the other hand, DeepSeek-R1 and ChatGPT-4 are the LLMs pre-trained on diverse codebases, offering the flexibility of pattern recognition informed by general knowledge of refactoring. Adaptive-Python-Refactoringminer [12] was considered, we excluded it due to replication challenges.

4.1.1 PyRef. PyRef [7] is the state-of-the-art tool that automatically detects method-level refactoring operations in Python projects. The tool was inspired by RefactoringMiner [34], using the same AST-based statement matching algorithm that determines refactoring candidates without requiring user-defined thresholds. Although RefactoringMiner2 was originally developed for Java, PyRef incorporates Python's syntax and semantics to support accurate refactoring detection in Python projects. However, PyRef only supports several key refactoring types in method-level refactoring and extended Rename Class recently.

4.1.2 MLRefScanner. MLRefScanner [24] is a prototype tool that applies ML techniques to detect refactoring commits in ML Python projects. MLRefScanner extracts textual, process, and code features to train classifiers for detecting commits. However, MLRefScanner relies on high-quality commit messages and is unable to detect specific types of refactorings. In the original setting, MLRefScanner and PyRef are combined by taking the union of refactoring-labeled commits, enabling coarse-grained (binary) detection. However, this approach cannot provide specific refactoring types and is thus not suitable for fine-grained evaluation. To make this baseline comparable to our method, we take the intersection of commits detected by both MLRefScanner and PyRef. We then use PyRef's output to label the specific refactoring operations. This ensures that the baseline only reports refactorings with higher confidence (both tools agree) and includes type-level details, enabling a fair comparison.

4.1.3 DeepSeek-R1 and ChatGPT-4. DeepSeek-R1 is an open-source LLM trained on extensive software-related data, including Python codebases. Unlike rule-based tools, DeepSeek-R1 exhibits strong pattern recognition abilities for both structural and semantic changes in code. ChatGPT-4 is a general-purpose LLM developed by OpenAI, pre-trained on diverse natural language and code corpora. Although it is not specifically designed for refactoring detection, its ability to understand complex code transformations and provide natural language explanations makes it a valuable baseline for evaluating semantic-level refactorings. We provide DeepSeek-R1 and ChatGPT-4 with full before-and-after file comtexts and a well-designed prompt with chain of thought (CoT) as instruction which can be found in the replication package [1]

4.2 Metrics

Following previous work [26, 34], we use four widely-used evaluation metrics including precision, recall, F1 score [26], defined as follows:

$$Precision_i = \frac{TP_i}{TP_i + FP_i} \tag{1}$$

$$Recall_i = \frac{TP_i}{TP_i + FN_i} \tag{2}$$

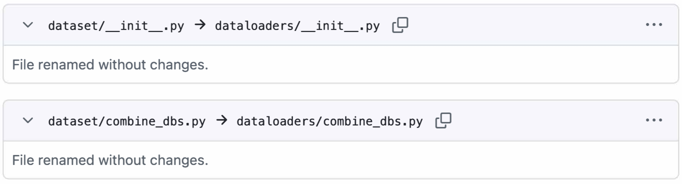
$$F1_i = \frac{2 \times Precision_i \times Recall_i}{Precision_i + Recall_i} \tag{3}$$

where TP_i is the number of valid refactoring operations mined by a tool on refactoring type i, FP_i is the number of invalid refactoring operations reported by a tool on refactoring type i, FN_i is the number of valid refactoring operations missed by a tool on refactoring type i, and N is the total number of refactoring types. Precision indicates the proportion of detected refactorings that are accurate, while recall measures the extent of actual refactorings detected. The F1-score provides a balanced measure of these two aspects.

4.3 Dataset

The original dataset is derived from Python-Adaptive-RefactoringMiner [12] and PyRef [7], which is available in our replication package [1]. We selected all commits from these datasets that contain refactoring types supported by ActRef in Table 1. We focus on a subset of common and structurally observable refactoring types, including class, method, and variable-level operations such as Rename, Move, Extract, and Inline. These types are well-supported by our action-based analysis framework and can be reliably detected through static structural and contextual changes. In contrast, we exclude certain behavior-affecting or inheritance-sensitive refactorings, such as Pull Up/Push Down Method or Add/Remove/Rename Parameter. Detecting these operations in Python is challenging and harmful due to the language's dynamic typing and flexible inheritance mechanisms [24]. Moreover, changes to method parameters often imply API evolution or semantic modifications, which are difficult to assess purely through static analysis and may exceed the scope of structural refactoring. As a result, we construct a dataset of 500 commits drawn from 136 open-source Python projects.

To extend the oracle, we executed ActRef, PyRef, DeepSeek-R1, and ChatGPT-4 on all 500 commits. We then manually validated the detected refactorings. During the manual detection phase, we directly classified results that matched the original dataset as true positives. Additionally, refactoring instances that were consistently detected by both ActRef and PyRef were also considered true positives without further review. However, due to LLM's tendency to over-detect, we did not automatically classify their results as true positives, even when they overlapped with ActRef or PyRef. Instead, for cases where there were differences, such as refactorings detected by LLM, newly identified refactorings, or missed refactorings compared to the original dataset annotations, we performed manual validation. The validation process followed a predefined set of rigorous rules (shown in Table 1) to ensure consistency. In rare instances of ambiguity, a second validator was consulted for confirmation.


dataset/__init__.py → dataloaders/__init__.py

File renamed without changes.

dataset/combine_dbs.py → dataloaders/combine_dbs.py

File renamed without changes.

Fig. 4. An Example of Move Module

During this process, we observed several instances of initial misclassification. For example, certain operations categorized as *Move Method* or *Move Class* were more accurately classified as *Move Module*, *Inline Module*, *Rename Module*, or *Extract Module*. Such cases were particularly common for file-level changes, where file renaming or relocation occurred without modifications to the file content (as shown in Figure 4). These cases were reclassified based on refined definitions to better reflect their actual semantics. Additionally, new refactoring instances detected by the LLM and verified as valid through manual inspection were also incorporated. All discrepancies and borderline cases were resolved through a secondary review by two authors to ensure consistency. After this thorough validation and reclassification process, we constructed a high-quality dataset containing 1,914 refactoring instances.

5 RESULTS

To evaluate the performance of ActRef, we answer the following research question :

- RQ1: How does ActRef perform compared to current tools for refactoring detection?
  This question investigates ActRef's detection accuracy, comparing it to existing refactoring detection tools in terms of precision, recall, and F1 scores.
- RQ2: How does ActRef's runtime compare to the state-of-the-art tool?
  This question examines the runtime efficiency of ActRef to understand whether its approach can achieve competitive performance in a practical setting.

Each RQ is explored through carefully designed experiments that are aimed at evaluating various aspects of ActRef. For each RQ, we detail the experimental setup, the metrics used for evaluation, and the results obtained.

5.1 RQ1: How does ActRef perform compared to current tools for refactoring detection?

This research question aims to evaluate the effectiveness of ActRef in detecting Python refactorings, comparing its accuracy and comprehensiveness against established refactoring detection tools. For this study, we selected rule-based tools and LLMs for comparison: **PyRef** [7], **PyRef** with **MLRefScanner** [24], **DeepSeek-R1** [10], and **ChatGPT-4** [25]. The results of different tools in terms of precision, recall, and F1-score are shown in table 2.

5.1.1 RQ1-1: How does ActRef perform compared to PyRef and PyRef with MLRefScanner? To assess the relative effectiveness of ActRef, PyRef and PyRef with MLRefScanner (PR+MS), we only analyze their performance on the five refactoring types both tools support: *Move Method*, *Extract Method*, *Inline Method*, *Rename Method*, and *Rename Class*.

ActRef consistently outperforms PyRef across five refactoring types, as reflected in its superior F1 scores of 0.84 compared to PyRef's 0.64. These higher scores indicate that ActRef provides a more balanced and accurate detection capability across a broader range of refactorings. Both

Table 2. Precision and Recall of ActRef and State-of-the-Art Approaches Per Refactoring Type

		ActRef			PyRef			PR+MS			DeepSeek-R1			ChatGPT-4		
Refactoring Type	Num	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Move Method	276	.79	.84	.81	.39	.38	.38	.38	.35	.37	.43	.37	.39	.42	.01	.16
Extract Method	167	.56	.84	.67	.94	.35	.51	.93	.30	.45	.58	.77	.67	.46	.72	.57
Inline Method	38	.34	.79	.47	.67	.24	.36	.67	.24	.36	.28	.57	.38	.08	.05	.06
Rename Method	432	.90	.98	.94	.95	.68	.80	.94	.54	.69	.96	.71	.82	.91	.56	.70
Rename Class	207	.97	.95	.96	.94	.74	.83	.94	.70	.80	.87	.79	.83	.78	.65	.71
Total (RQ1.1)	1,111	.78	.91	.84	.76	.55	.64	.72	.48	.58	–	–	–	–	–	–
Move Class	34	.94	.91	.93	–	–	–	–	–	–	.35	.43	.39	.18	.10	.13
Extract Class	79	.77	.51	.61	–	–	–	–	–	–	.46	.81	.59	.39	.70	.50
Inline Class	1	1.00	1.00	1.00	–	–	–	–	–	–	.03	1.00	.07	.00	.00	.00
Move Module	258	1.00	1.00	1.00	–	–	–	–	–	–	.81	.93	.86	.96	.59	.73
Extract Module	25	.65	.80	.71	–	–	–	–	–	–	.12	.47	.19	.19	.23	.20
Inline Module	11	.69	.82	.75	–	–	–	–	–	–	.07	.20	.10	.17	.09	.12
Rename Module	27	1.00	1.00	1.00	–	–	–	–	–	–	.46	.95	.62	.36	.67	.47
Extract Variable	88	.70	.86	.78	–	–	–	–	–	–	.25	.08	.12	.26	.11	.16
Inline Variable	35	.37	.94	.53	–	–	–	–	–	–	.06	.03	.04	.06	.03	.04
Rename Variable	243	.86	1.00	.92	–	–	–	–	–	–	.44	.61	.51	.46	.34	.39
Total (RQ1.2)	1,921	.80	.92	.85	–	–	–	–	–	–	.58	.63	.60	.57	.43	.49

tools demonstrate high precision, with ActRef surpassing PyRef in recall (0.91 vs. 0.55), capturing a significantly higher number of true positives in these categories. PyRef's relatively low recall suggests it may miss a substantial number of applicable refactorings, likely due to its stricter matching criteria and limited adaptability to varied code structures, which we have discussed in Chapter 2. However, ActRef's precision in certain categories is lower than baseline tools, such as *Extract Method* and *Inline Method.* This discrepancy can be attributed to the limitations of GumTree's SimpleMatcher algorithm. SimpleMatcher, which ActRef relies on, tends to misidentify nodes with similar names, leading to false positives. For example, when a substantial portion of a function's body is extracted into a new method, or an old method is inlined to another method, GumTree may mistakenly interpret this *Extract Method* or *Inline Method* as a *Rename Method* operation due to the similarity in node names and structure. In these cases, ActRef may incorrectly report a *Rename Method*, overlooking differences in parameters or calling context and focusing only on the changes in name and abstract syntax tree similarity. To mitigate this issue, we incorporated a check on the context of the action's subject, which we introduce in Chapter 6 to reduce such false positives. Although this adjustment has helped, it has not entirely eliminated the problem. However, it remains valuable of ActRef's detection on *Inline Method* and *Extract Method*, because it might be easier for developers to verify a lot of refactorings rather than identify missed refactorings. Future improvements could focus on further enhancing context-aware analysis to improve precision in complex refactorings. Additionally, PyRef detected 26 instances that should be classified as *Extract Class*, *Move Class*, or other module-level refactoring, but it incorrectly identified them as *Move Method.* This misclassification occurred because PyRef does not support the detection of these refactoring types. Due to this limitation, we have decided to omit these instances from PyRef's result.

In summary, ActRef stands as the preferred choice for scenarios demanding thorough refactoring detection and a balance between precision and recall. Its ability to adapt to varied refactoring patterns makes it particularly suitable for large-scale, diverse datasets, providing actionable insights that surpass PyRef's narrower focus on precision.

In addition to comparing ActRef with PyRef, we also evaluated the combination of PyRef and MLRefScanner (denoted as MS+PR). MLRefScanner is a classifier trained to identify whether a commit contains refactoring, and when combined with PyRef, it aims to improve detection by filtering or re-ranking results. However, our results show that MS+PR underperforms PyRef in precision, recall and F1 scores, highlighting that the classifier does not significantly enhance PyRef's structural matching capabilities. MS+PR consistently demonstrates lower recall across all five refactoring types, particularly in *Extract Method* and *Inline Method*, where the recall dropped to 0.30 and 0.24. These findings suggest that MLRefScanner, which is optimized for commit-level classification, is not well-suited for identifying fine-grained method-level refactorings. It may introduce noise or overly conservative filtering, resulting in missed detections. The lack of contextual understanding at the statement level limits the effectiveness of this hybrid approach. Therefore, while MS+PR may be useful for commit-level refactoring detection, it offers little advantage in detecting specific refactoring operations compared to using PyRef alone. These observations further reinforce the advantage of ActRef 's action-based strategy. Unlike MS+PR, which relies on high-quality commit messages, ActRef analyzes concrete AST-level changes action, yielding a better balance between precision and recall, particularly in complex or dynamic codebases.

5.1.2 RQ1-2: How does ActRef *perform compared to LLMs?* To evaluate whether LLMs can effectively detect Python refactorings, we compared ActRef against two strong LLM-based baselines: DeepSeek-R1, an open-source code model fine-tuned for reasoning, and ChatGPT-4, a state-of-the-art commercial LLM. The results, summarized in Table 2, demonstrate a significant performance gap between ActRef and both LLMs across all 15 refactoring types. It is worth noting that, due to the input length constraints of LLMs, a small number of commits could not be processed by LLMs: 21 commits exceeded the token limit for DeepSeek-R1, and 5 for ChatGPT-4. We excluded the out-of-limit-token commits from the evaluation of referenced LLM. ActRef achieves an F1 score of 0.85, outperforming DeepSeek-R1 (F1: 0.60) and ChatGPT-4 (F1: 0.49) by large margins. In terms of recall, ActRef achieves 0.92 compared to 0.63 and 0.43 for DeepSeek-R1 and ChatGPT-4. This shows that ActRef is far more capable of recovering actual refactoring instances. Moreover, even in terms of precision, where LLMs are often assumed to excel due to their semantic understanding, ActRef still outperforms both (P: 0.80 vs. 0.58 and 0.57). These findings suggest that current LLMs tend to miss many refactorings (i.e., low recall) and sometimes hallucinate patterns that do not correspond to actual structural changes (moderate precision).

We also observe that ActRef 's precision or recall is relatively lower in a few categories (e.g., Extract Method and Extract Class). Extract Method's precision is lower than DeepSeek-R1 (0.56 vs. 0.58), recall drops to 0.51 in Extract Class, indicating that ActRef may miss extractions with significant transformations or multi-step restructuring. Nevertheless, the F1 remains reasonable (0.61), demonstrating that most detected cases are still correct.

Despite such challenges, ActRef consistently outperforms both LLMs on every single refactoring type in terms of F1 score. LLMs like DeepSeek-R1 and ChatGPT-4 exhibit inconsistent behavior: they may achieve relatively high recall in a few categories (e.g., ChatGPT-4 on Extract Class), but often at the cost of extremely low precision. This leads to significantly lower F1 scores, suggesting that LLMs still struggle to distinguish refactoring patterns from general edits in code.

Answer to RQ1: ActRef consistently outperforms both rule-based tools and LLMs in detecting Python refactorings across a range of types. ActRef's higherF1 suggest that it provides more balanced and accurate detection.

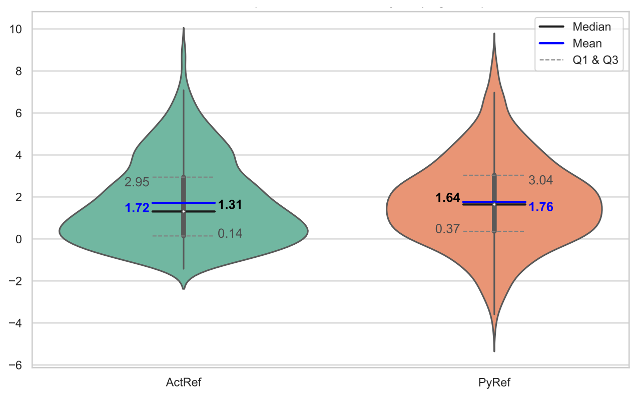


Fig. 5. Runtime Comparison of ActRef and PyRef

5.2 RQ2: How does ActRef's runtime compare to state-of-the-art tool?

In this research question, we aim to evaluate the runtime performance of ActRef in comparison to PyRef. We conducted an experiment using the same dataset in RQ1, which contains commits from multiple open source Python projects, covering different domains and sizes. We executed separately each tool on the same machine with the following specifications: Intel Xeon Platinum 8358P CPU 2.60GHz, 2.0 TiB of DDR4 memory, running Ubuntu 20.04.6 LTS and Python 3.7. In order to avoid the impact of network fluctuations on the experiment, we downloaded all the Python codes involved in the commit to the local computer in advance, so we bypassed the git parsing module when executing PyRef. For each commit in the dataset, both ActRef and PyRef were executed and the time taken to complete the analysis was measured. This approach provides a precise measurement of how each tool performs on individual commits, avoiding aggregate measurements that could obscure commit-level performance variations. Each commit was analyzed three times to reduce the impact of random fluctuations in runtime caused by external factors such as system load. The average runtime across these trials was then calculated for each tool, ensuring that the reported results reflect consistent performance.

We use violin plots to illustrate the runtime distribution of ActRef and PyRef. To intuitively show the runtime performance of ActRef and PyRef on all test cases, we logarithmically transformed the runtime. As shown in Figure 5, although ActRef supports more refactoring types (15, compared to 10 for PyRef), the runtimes of the two are very close in terms of average and median. This shows that ActRef can maintain comparable performance to the existing tool while supporting more refactoring types' detection. The distribution of ActRef has some large outliers at the top, which are mainly due to the fact that ActRef supports more complex refactoring types, such as cross-file refactoring or more complex extract and move operations. The detection of these operations requires more computational resources, thus resulting in longer runtimes in some more complex code commits. ActRef uses the SampleMatch-gumtree action calculation algorithm, which can lock the changed areas in the code more quickly and accurately to avoid redundant operations.

This is particularly critical in complex refactoring scenarios (such as extraction, movement, and cross-file operations), so even if more refactoring types are supported, the running time of ActRef is still similar as PyRef.

Answer to RQ2: ActRef matches PyRef in runtime performance while supporting a broader range of refactoring types and finer detection granularity. This balance suggests that ActRef can deliver comprehensive refactoring insights without added runtime cost, enhancing its practicality for detailed, real-world code analysis.

6 DISCUSSION

In this section we provide a comprehensive discussion of the research, examining both the implications and potential threats. The implications section highlights the broader impact of ActRef on Python code refactoring detection, while the threats section addresses possible limitations, ensuring a balanced view of the results and their reliability.

6.1 Implications

The implications of our work are summarized as follows:

- The introduction of an action-based analysis approach to refactoring behaviours enables the observation and evaluation of these behaviours from a new perspective, while also revealing the complexity and diversity of refactoring operations. This novel perspective helps to address the current knowledge gap in refactoring analysis within the academic community, offering a theoretical foundation for future research.
- The staged analysis pipeline from module-level to AST-level actions demonstrates a robust and scalable strategy for detecting a wide range of refactorings, from coarse-grained refactoring to fine-grained refactoring. This layered design not only improves detection accuracy but also enhances extensibility, making it suitable for analyzing large-scale, real-world software projects.
- ActRef is inherently extensible, offering a unified representation for detecting diverse refactoring types. As new refactoring patterns emerge, they can be accommodated by augmenting the rule set without redesigning the entire system. This design paves the way for building more intelligent, adaptable refactoring detectors across programming languages and ecosystems. Extending the approach taken by ActRef to adapt to other programming languages or frameworks is our ongoing work.

6.2 Internal Threats

The internal threat comes from the choice of the action calculation algorithm. Although GumTree has shown reliable performance in previous studies and our tests, it is not flawless and may sometimes misclassify changes, resulting in false positives. The detected action can vary significantly for the same code snippets based on the selected threshold [6, 29]. Although we tested alternative strategies of GumTree (ClassicMatcher and GreedyMatcher) and thresholds for calculating actions, the experimental results indicated that SimpleMatcher with a self-adaption threshold offered the most balanced trade-off between accuracy and computational efficiency and comprehensive results for our specific use case. We also implement several filters to reduce the risk of such misclassifications, yet some level of error remains inevitable. Future improvements may involve integrating other matching algorithms to further enhance the precision of action detection. Still, the current use of SimpleMatcher paired with our context-aware post-processing mechanism provides a robust foundation for this study's findings.

GumTree tends to generate the shortest edit script by maximizing subtree matches, which may cause false positives in Rename Method and false negatives in Extract Method when large code fragments are extracted. To mitigate this, we apply post-processing rules that verify whether the original node has been fully or partially deleted, and whether similar fragments are added elsewhere, before classifying a change as an update or a move. Additionally, we enhanced GumTree to consider semantic signatures when pairing top-level declarations, which mitigates the common false positives in Inline Method and Extract Method cases that arise from similar subtree structures.

Additionally, we initially considered using Python-Adaptive-Refactoring Miner [12] and were able to successfully execute its replication package with assistance from the original authors, we ultimately excluded it from our evaluation due to significant reliability issues. Specifically, the tool failed to detect any refactoring operations, even when applied to the original dataset provided by the authors. Furthermore, this limitation is not unique to our experience, similar issues have been reported by other users on the tool's official replication package repository[1], suggesting a broader reproducibility problem.

6.3 External Threats

A key external threat to the validity of our results lies in the scope and representativeness of the dataset. Our dataset is drawn from previous work [7, 12], which identified refactorings in real-world development scenarios. These datasets were curated using state-of-the-art detection tools, ensuring that the commits represent authentic refactoring practices in open-source projects. However, as the initial datasets only include commits with refactorings detected by prior tools, they are inherently biased toward simpler refactoring cases. This bias stems from the limitations of the detection tools themselves, which often struggle to identify more complex or nuanced refactoring patterns, resulting in an overrepresentation of straightforward scenarios.

To address this limitation and enrich the dataset, we performed additional detection on the original commits, uncovering and incorporating new refactoring instances that were not initially identified. While this process expanded the dataset and increased its diversity, the overall composition might still lean towards simpler refactoring cases, as the starting point was shaped by the detection capabilities of the tools used in earlier works. This tendency toward simpler cases highlights a broader challenge in refactoring research: mainstream detection tools frequently perform well on well-defined refactorings but face significant difficulties in detecting more intricate or multi-step transformations. Consequently, our evaluation, while reflective of the characteristics of the dataset, might not fully capture the performance of the tools in complex real-world scenarios. Future work should focus on extending the dataset to better represent the full spectrum of refactoring practices in software development, including those involving intricate transformations and interactions across multiple files or modules. Such efforts would provide deeper insights into the challenges faced by detection tools and further enhance the robustness of their evaluation.

Another external threat is the nature of using action-based detection for identifying refactorings, especially when compared to established tools like PyRef. While action-based detection allows for a more flexible and human-centric analysis of code transformations, it may struggle with certain refactorings that involve more nuanced semantic changes not captured in the action set. This could result in some refactorings going undetected. While our approach aims to overcome some limitations of previous tools by focusing on both structural and semantic aspects of code changes, future work should include the evaluation of our tool on more diverse datasets and across different programming languages to assess its broader applicability.

[1]https://github.com/mlcodepatterns/PythonTypeInformation/issues/2

7 RELATED WORK

Weißgerber and Diehl [37] introduced a pioneering technique for detecting local-scope and class-level refactorings in the commit histories of CVS repositories. Their approach begins by identifying potential refactoring candidates through pairs of code elements (e.g., classes, methods, fields) that exhibit signature differences. To analyze these candidates further, they employed the clone detection tool CCFinder [19], configured to tolerate variations such as whitespace and comment changes, as well as consistent renaming of variables, methods, and member references. To evaluate their method, they manually inspected commit log messages from two open-source projects to identify documented refactorings and calculate recall. Additionally, they used random sampling to estimate the precision of their technique.

Ref-Finder [20] utilizes a logic programming approach to detect a wide range of refactorings, offering the most comprehensive coverage among tools based on Fowler's catalog, supporting 63 refactoring types. It defines each refactoring as a set of logic query templates that capture structural constraints before and after the refactoring. This approach excels at identifying both atomic (e.g., Rename Method) and composite (e.g., Extract Superclass) refactorings, making it highly versatile for Java projects. However, its reliance on logic-based templates and a formalized structure heavily ties it to Java's statically typed environment, which limits its application to dynamic languages like Python, where structure may be less strictly defined. Moreover, Ref-Finder's approach can become computationally expensive and less practical for real-world commits that mix refactorings with bug fixes or feature additions. Its dependency on the correctness of structural facts means that overlapping or noisy changes are harder to filter out, making it less suitable for detecting refactorings in mixed or complex commits.

JDevAn [39, 40] proposed a method for detecting refactorings by analyzing fine-grained changes in Abstract Syntax Trees across commits. Their approach is based on the design-level changes reported by UMLDIFF [38], which is a domain-specific structural differencing algorithm that analyzes two class models, reverse-engineered from consecutive versions of an object-oriented system, to identify structural changes in packages, classes, interfaces, fields, and methods. The authors evaluated their technique by validating detected refactorings in several open-source Java projects, demonstrating its effectiveness in capturing a broad range of documented refactorings.

RefactoringMiner 2.0 [34] is a state-of-the-art tool for refactoring detection focused primarily on Java. It is renowned for its precision and recall, achieving a high accuracy without relying on code similarity thresholds, unlike many of its predecessors. The tool specializes in detecting both high-level and low-level refactorings, including Extract Method and Rename Method, as well as submethod-level refactorings such as Extract Variable. This makes it particularly effective for Java projects. However, RefactoringMiner's reliance on Java's static type system and specific statement-matching techniques makes it language-specific, limiting its direct applicability to dynamic languages like Python. The approach is tied closely to object-oriented constructs typical of statically typed languages, which makes it harder to generalize to dynamic languages that might not have strict type declarations or consistent structural patterns.

Refdiff 2.0 [30] is a multi-language refactoring detection tool designed to overcome the limitation of being restricted to a single language, such as Java, by providing support for Java, JavaScript, and C. RefDiff abstracts code structure into a Code Structure Tree (CST), which enables it to identify high-level refactorings, including renames, moves, and extracts. However, while it can generalize across multiple languages, RefDiff is still reliant on a token-based similarity measure that may miss more granular or subtle changes, such as method body refactorings (like Extract Variable) that occur frequently in dynamic languages like Python. RefDiff focuses on detecting low-level refactorings and handling more complex code changes common in languages like Python, makes it less suited for cases where a detailed, intra-method refactoring analysis is needed, or when there are overlapping changes that mix refactoring with other types of modifications

Python-Adaptive-RefactoringMiner [12], adapted from RefactoringMiner, focuses on detecting refactorings in Python code by transform Python AST to Java AST and building on the strengths of RefactoringMiner's statement-matching techniques. It retains high precision for detecting certain refactorings, such as method extraction and renaming, through statement matching algorithms. However, like RefactoringMiner, this tool struggles with Python's dynamic nature. Python lacks the strict typing and structural rigidity found in Java, making it harder for tools like this to precisely capture refactorings when code undergoes significant structural changes or dynamic behavior alterations. Additionally, Python-Adaptive-RefactoringMiner may be challenged by real-world Python commits, which often blend functional changes with refactorings. Its statement-matching algorithm, while effective for straightforward refactorings, might misinterpret or miss complex refactorings embedded in larger modifications.

PyRef [7], based on the RefactoringMiner framework, focuses on detecting a limited set of refactorings. While these refactorings are essential, PyRef's limited scope restricts its ability to handle the wide variety of refactorings often seen in real-world Python projects, which may involve more intricate or composite changes. PyRef's reliance on AST-based matching methods poses challenges when handling Python's dynamic features, such as duck typing, flexible function signatures, and frequent structural changes. This limitation becomes particularly pronounced in commits that blend refactoring with other functional changes, leading to a reduced ability to detect complex refactorings across multiple files or in larger codebases.

MLRefScanner [24] is a prototype tool that applies ML techniques to detect refactoring commits in ML Python projects. MLRefScanner extracts textual, process, and code features to train classifiers to detect commits. They conducted extensive experiments on 199 ML Python projects and verified the performance of MLRefScanner in various testing scenarios, and achieves a precision of 94%, a recall of 82%, and an AUC of 89%. However, MLRefScanner relies on high-quality commit messages and is unable to detect specific types of refactorings.

Existing refactoring detection tools like RefactoringMiner, RefDiff, and Ref-Finder offer effective mechanisms for identifying code changes, but they face limitations in Python-specific code changes. PyRef provides high precision but support only a limited set of refactorings and struggle with Python's flexible code structure and dynamic nature. RefDiff, though multi-language, relies on code similarity, making it less effective for complex or mixed changes in real-world commits. These limitations highlight the need for a more flexible, action-based approach that better captures nuanced refactorings, especially in dynamic languages like Python, where existing methods fall short.

8 CONCLUSION

This paper introduces ActRef, a novel refactoring mining algorithm based on code change actions, aiming to enhance the understanding of Python code refactoring. In contrast to existing methodologies, ActRef reveals the diversity and complexity of refactoring operations by analyzing specific actions within the code change process, thus offering a novel analytical perspective to software engineering. Experimental results indicate that ActRef outperforms current advanced refactoring detection tools in both effectiveness and efficiency, excelling in identifying complex code change patterns and distinguishing among various refactoring types. ActRef also demonstrates superior accuracy and detail in refactoring identification compared to large language models, highlighting the value of structured action-based methods in consistently capturing code changes with precision.

In conclusion, ActRef introduces a new perspective on understanding code refactoring, while opening up pathways for future research. Our ongoing work is to explore ActRef's efficacy in multi-language settings and advance the automation of action analysis through deep learning to address increasingly complex code evolution needs.

REFERENCES

[1] [n. d.]. *Replication package.* https://figshare.com/s/984c7a39266137e29c37

[2] 2024. *Imagaug.* https://github.com/aleju/imgaug

[3] Eman Abdullah Alomar. 2019. Towards Better Understanding Developer Perception of Refactoring. *IEEE* (2019).

[4] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. 2019. On the impact of refactoring on the relationship between quality attributes and design metrics. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM).* IEEE, 1–11.

[5] Eman Abdullah AlOmar, Tianjia Wang, Vaibhavi Raut, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. 2022. Refactoring for reuse: an empirical study. *Innovations in Systems and Software Engineering* (2022), 1–31.

[6] Maurício Aniche, Christoph Treude, Andy Zaidman, Arie Van Deursen, and Marco Aurélio Gerosa. 2016. SATT: Tailoring code metric thresholds for different software architectures. In *2016 IEEE 16th international working conference on source code analysis and manipulation (SCAM).* IEEE, 41–50.

[7] Hassan Atwi, Bin Lin, Nikolaos Tsantalis, Yutaro Kashiwa, Yasutaka Kamei, Naoyasu Ubayashi, Gabriele Bavota, and Michele Lanza. 2021. PyRef: refactoring detection in Python projects. In *2021 IEEE 21st international working conference on source code analysis and manipulation (SCAM).* IEEE, 136–141.

[8] Abdulrahman Ahmed Bobakr Baqais and Mohammad Alshayeb. 2020. Automatic software refactoring: a systematic literature review. *Software Quality Journal* 28, 2 (2020), 459–502.

[9] Zhiyuan Chen, Hai-Feng Guo, and Myoungkyu Song. 2018. Improving regression test efficiency with an awareness of refactoring changes. *Information and Software Technology* 103 (2018), 174–187.

[10] DeepSeek. 2025. *DeepSeek-R1.* https://www.deepseek.com/

[11] Kayla DePalma, Izabel Miminoshvili, Chiara Henselder, Kate Moss, and Eman Abdullah AlOmar. 2024. Exploring ChatGPT's code refactoring capabilities: An empirical study. *Expert Systems with Applications* 249 (2024), 123602. https://doi.org/10.1016/j.eswa.2024.123602

[12] Malinda Dilhara, Ameya Ketkar, Nikhith Sannidhi, and Danny Dig. 2022. Discovering repetitive code changes in python ml systems. In *Proceedings of the 44th International Conference on Software Engineering.* 736–748.

[13] Jean-Rémy Falleri and Matias Martinez. 2024. Fine-grained, accurate and scalable source differencing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering.* 1–12.

[14] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering.* 313–324.

[15] Martin Fowler. 2018. *Refactoring.* Addison-Wesley Professional.

[16] Xi Ge, Quinton L. DuBose, and Emerson Murphy-Hill. 2012. Reconciling manual and automatic refactoring. In *2012 34th International Conference on Software Engineering (ICSE).* 211–221. https://doi.org/10.1109/ICSE.2012.6227192

[17] Martina Iammarino, Fiorella Zampetti, Lerina Aversano, and Massimiliano Di Penta. 2019. Self-Admitted Technical Debt Removal and Refactoring Actions: Co-Occurrence or More?. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME).* 186–190. https://doi.org/10.1109/ICSME.2019.00029

[18] Martina Iammarino, Fiorella Zampetti, Lerina Aversano, and Massimiliano Di Penta. 2019. Self-admitted technical debt removal and refactoring actions: Co-occurrence or more?. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 186–190.

[19] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE transactions on software engineering* 28, 7 (2002), 654–670.

[20] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. 2010. Ref-finder: a refactoring reconstruction tool based on logic query templates. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering.* 371–372.

[21] Yun Lin, Xin Peng, Yuanfang Cai, Danny Dig, Diwen Zheng, and Wenyun Zhao. 2016. Interactive and guided architectural refactoring with search-based recommendation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* 535–546.

[22] Bo Liu, Hui Liu, Nan Niu, Yuxia Zhang, Guangjie Li, and Yanjie Jiang. 2023. Automated Software Entity Matching Between Successive Versions. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 1615–1627.

[23] Zheng Ma, Yuexiu Gao, Lei Lyu, and Chen Lyu. 2022. MMF3: neural code summarization based on multi-modal fine-grained feature fusion. In *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement.* 171–182.

[24] Shayan Noei, Heng Li, and Ying Zou. 2025. Detecting Refactoring Commits in Machine Learning Python Projects: A Machine Learning-Based Approach. *ACM Transactions on Software Engineering and Methodology* 34, 3 (2025), 1–25.
[25] OpenAI. 2025. *ChatGPT-4*. https://openai.com/index/chatgpt
[26] Juri Opitz and Sebastian Burst. 2019. Macro f1 and macro f1. *arXiv preprint arXiv:1911.03347* (2019).
[27] Napol Rachatasumrit and Miryung Kim. 2012. An empirical investigation into the impact of refactoring on regression testing. In *2012 28th ieee international conference on software maintenance (icsm)*. IEEE, 357–366.
[28] Bo Shen, Wei Zhang, Haiyan Zhao, Guangtai Liang, Zhi Jin, and Qianxiang Wang. 2019. Intellimerge: A refactoring-aware software merging technique. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–28.
[29] Christopher M Siefert, Timothy A Smith, and Elliott M Ridgway. 2021. *Evaluation of Programming Language-Aware Diffs for Improving Developer Productivity*. Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).
[30] Danilo Silva, Joao Paulo da Silva, Gustavo Santos, Ricardo Terra, and Marco Tulio Valente. 2020. Refdiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering* 47, 12 (2020), 2786–2802.
[31] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*. 858–870.
[32] Danilo Silva and Marco Tulio Valente. 2017. Refdiff: detecting refactorings in version histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 269–279.
[33] Tiobe. 2025. *Tiobe2025*. https://www.tiobe.com/tiobe-index/
[34] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2020. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* 48, 3 (2020), 930–950.
[35] Nikolaos Tsantalis, Matin Mansouri, Laleh M Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th international conference on software engineering*. 483–494.
[36] Kaiyuan Wang, Chenguang Zhu, Ahmet Celik, Jongwook Kim, Don Batory, and Milos Gligoric. 2018. Towards refactoring-aware regression test selection. In *Proceedings of the 40th International Conference on Software Engineering*. 233–244.
[37] Peter Weißgerber and Stephan Diehl. 2006. Identifying refactorings from source-code changes. In *21st IEEE/ACM international conference on automated software engineering (ASE'06)*. IEEE, 231–240.
[38] Zhenchang Xing and Eleni Stroulia. 2005. UMLDiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 54–65.
[39] Zhenchang Xing and Eleni Stroulia. 2006. Refactoring Detection based on UMLDiff Change-Facts Queries. In *2006 13th Working Conference on Reverse Engineering*. 263–274. https://doi.org/10.1109/wcre.2006.48
[40] Zhenchang Xing and Eleni Stroulia. 2008. The JDEvAn tool suite in support of object-oriented evolutionary development. In *Companion of the 30th international conference on Software engineering*. 951–952.
[41] Yilin Yang, Tianxing He, Yang Feng, Shaoying Liu, and Baowen Xu. 2022. Mining Python fix patterns via analyzing fine-grained source code changes. *Empirical Software Engineering* 27, 2 (2022), 48.