

ANTICOPYPASTER 3.0: Just-in-Time Clone Refactoring

EMAN ABDULLAH ALOMAR, JACOB ASHKENAS, ROBERT FELICIANO, MATTHEW ANGELAKOS, DIMITRIOS HARALAMPOPOULOS, and XING QIAN, Stevens

Institute of Technology, Hoboken, New Jersey, USA

MOHAMED WIEM MKAOUER, University of Michigan-Flint, Flint, Michigan, USA

ALI OUNI, ETS Montreal, University of Quebec, Montreal, Quebec, Canada

Refactoring is a crucial practice in software maintenance that aims at improving design and coding practices while addressing design flaws. The *Extract Method* refactoring is particularly popular for consolidating duplicate code fragments into a single method. Various studies have explored ways to recommend *Extract Method* refactoring opportunities using techniques such as program slicing, program dependency graph analysis, change history analysis, structural similarity, and feature extraction. Despite their effectiveness, these approaches often disrupt the developer's workflow, requiring them to pause their coding and assess the refactoring opportunities suggested throughout the project, without considering the specific development context. To enhance the adoption of *Extract Method* refactoring, our previous work proposed ANTICOPYPASTER 2.0 and investigated the effectiveness of detecting and extracting code clones without disrupting the developer's workflow. To address these limitations, we develop a new approach in this article that supports the detection of Type-1 and Type-2 clones using the Program Structure Interface (PSI) and includes a custom-built *Extract Method* refactoring tool. We implement our approach using an IntelliJ IDEA extension plugin. Additionally, we integrated name recommendation models, including IntelliJ's built-in recommender and CODE2VEC, to enhance the quality of method names and improve developer productivity. To evaluate the accuracy and usefulness of our approach, we conducted a qualitative study involving 13 developers. The results indicate that (1) developers appreciate the approach and are satisfied with various aspects of the plugin's functionality, (2) PSI effectively identifies clones by analyzing the structural and semantic aspects of the code, (3) IntelliJ's naming recommender often provides default generic names, while CODE2VEC produces descriptive and relevant names based on the code context, (4) the performance of ANTICOPYPASTER remains stable regardless of the file size and the number of clones present, (5) despite different detection and correction mechanisms, JDEODORANT and ANTICOPYPASTER were able to perform method extraction, and ANTICOPYPASTER features just-in-time detection and correction, and (6) our results show an improvement in code quality after performing *Extract Method* refactoring with both refactoring tools. We envision that our ANTICOPYPASTER solution can streamline the *Extract Method* refactoring process, enhancing both developer efficiency and code quality by seamlessly integrating Type-2 clone detection and name recommendation capabilities in the development workflow.

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**;

Authors' Contact Information: Eman Abdullah AlOmar (corresponding author), Stevens Institute of Technology, Hoboken, New Jersey, USA; e-mail: ealomar@stevens.edu; Jacob Ashkenas, Stevens Institute of Technology, Hoboken, New Jersey, USA; e-mail: jashkena@stevens.edu; Robert Feliciano, Stevens Institute of Technology, Hoboken, New Jersey, USA; e-mail: rfelici1@stevens.edu; Matthew Angelakos, Stevens Institute of Technology, Hoboken, New Jersey, USA; e-mail: mangelak@stevens.edu; Dimitrios Haralampopoulos, Stevens Institute of Technology, Hoboken, New Jersey, USA; e-mail: dharalam@stevens.edu; Xing Qian, Stevens Institute of Technology, Hoboken, New Jersey, USA; e-mail: xqian7@stevens.edu; Mohamed Wiem Mkaouer, University of Michigan-Flint, Flint, Michigan, USA; e-mail: mmkaouer@umich.edu; Ali Ouni, ETS Montreal, University of Quebec, Montreal, Quebec, Canada; e-mail: ali.ouni@etsmtl.ca.



This work is licensed under Creative Commons Attribution International 4.0.

© 2026 Copyright held by the owner/author(s).

ACM 1557-7392/2026/4-ART132

<https://doi.org/10.1145/3749100>

Additional Key Words and Phrases: refactoring, code duplication, clone refactoring, software quality

ACM Reference format:

Eman Abdullah AlOmar, Jacob Ashkenas, Robert Feliciano, Matthew Angelakos, Dimitrios Haralampopoulos, Xing Qian, Mohamed Wiem Mkaouer, and Ali Ouni. 2026. ANTI-COPYPASTER 3.0: Just-in-Time Clone Refactoring. *ACM Trans. Softw. Eng. Methodol.* 35, 5, Article 132 (April 2026), 52 pages.

<https://doi.org/10.1145/3749100>

1 Introduction

When writing code, software developers often reuse code by duplicating and (slightly) modifying code segments, leading to the creation of code clones [Roy and Cordy, 2008]. Although reusing code is fundamental to software engineering, code clones present unique issues by reducing program comprehension and ease of maintenance. Specifically, when developers modify a single code clone, they usually need to apply the same changes to other clones, thus increasing the likelihood of errors in the program [Tsantalis et al., 2015].

In this context, Li et al. [2006] have shown that up to 22% of OS errors were found in duplicate code. To manage code clones, a whole community was established to develop tools and techniques for their detection and correction [IWSC, 2022]. However, in a recent analysis of a dataset containing 2.6 million original projects on GitHub, which included more than 258 million files in languages such as Java, C++, Python, and JavaScript, Lopes et al. found a significant level of duplication, far exceeding expectations, which is alarming, especially when leveraging open source repositories for statistical analysis and machine learning [Lopes et al., 2017].

One practical way to remove clones is through refactoring [AlOmar et al., 2023b; Choi et al., 2011; Ehsan et al., 2023; Fowler, 2018; Higo et al., 2004; Mondal et al., 2014; Roy and Cordy, 2007]. Refactoring is the practice of improving software quality without altering its behavior [Fowler, 2018]. Developers intuitively refactor their code for multiple purposes such as improving program comprehension, eliminating duplicate code, reducing complexity, dealing with technical debt, and removing code smells [AlOmar et al., 2019; Mkaouer et al., 2015; Murphy-Hill and Black, 2008; Ouni et al., 2016; Silva et al., 2016]. Refactoring duplicate code involves relocating a code segment to form a new method and substituting every occurrence of that segment with a call to this new method. This specific refactoring is termed as *Extract Method*. Numerous studies have advocated for refactoring based on various motivations and foundations, such as improving code structure to identify and automate refactoring [Bavota et al., 2014; Kanemitsu et al., 2011], using feature extraction to predict or recommend refactoring [Aniche et al., 2020; Xu et al., 2017b; Yue et al., 2018], reducing code duplication to improve the success rate of refactoring [Alcocer et al., 2020; AlOmar, 2025; Higo et al., 2008; Hotta et al., 2010; Yoshida et al., 2019], and removing the *Long Method* code smell to explore its impact on class change and fault proneness [Khomh et al., 2012; Morales et al., 2017; Palomba et al., 2014; Palomba et al., 2018; Tiwari and Joshi, 2022; Yang et al., 2009].

Although these studies produce promising results, their adoption is hindered by the necessity to thoroughly scan the entire code base to recommend appropriate *Extract Method* refactorings. In essence, they require the project's complete source code as input and scrutinize it to identify potential *Extract Method* refactoring opportunities. However, developers might lack either the time or the expertise to implement such extensive refactoring across the entire program, making them reluctant to follow these suggestions. Additionally, this method necessitates that developers allocate extra time to review various refactoring proposals that are irrelevant to their current tasks, thereby decreasing the likelihood of these recommendations being implemented in practice. However,

developers also do not prefer to recommend clones to refactor without providing solutions [Ehsan et al., 2023]. Even though modern programming environments include *Extract Method* refactoring as an integrated feature, its usage by developers remains limited even in this context [Golubev et al., 2021]. Furthermore, code clone tools mainly focus on the refactoring of exact clones, known as Type-1 clones (i.e., identical code fragments except for variations in whitespace, comments, and layouts, and formatting [Roy et al., 2009]), as their refactoring is straightforward, unlike near clones, known as Type-2 clones (i.e., syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout, and comments [Roy et al., 2009]), that are more complex and present challenges due to these variations.

ANTICOPYPASTER aims to automatically provide *just-in-time Extract Method* refactoring recommendation as soon as duplicate code is created [AlOmar et al., 2022; AlOmar et al., 2023a; AlOmar et al., 2024]. Our tool takes various semantic and syntactic code metrics as input and makes a binary decision on whether the code fragment has to be extracted, either using (1) our built-in binary classifier [AlOmar et al., 2022; AlOmar et al., 2023a] or (2) a user-specified configuration [AlOmar et al., 2024]. Currently, ANTICOPYPASTER 1.0 and 2.0 have the following limitations:

Limitation #1: Clone detection. The tool detects exact duplicates and is confined to certain subtypes of Type-1 clone detection. The detection was accomplished through two methods: direct substrings matching and bag-of-words. Unfortunately, these methods struggled with most Type-2 clones, or even some Type-1 clones, if they differ by whitespaces or comments.

Limitation #2: Clone refactoring. The removal of exact duplicates relies on using IntelliJ’s built-in method extraction, which is only suitable for Type-1 clones.

Limitation #3: Method extraction settings. Version 1.0 of the tool utilized only the AI model, while version 2.0 utilized only the custom user extraction that involved clustering and advanced metrics. Thus, developers cannot choose which extraction method to use.

Limitation #4: Name recommendation. Developers need to name the extracted method, which might impact the quality of the method names due to lack of context, and might affect developer productivity and cause disruption.

Limitation #5: User-configurable settings. Version 1.0 does not allow developers to configure settings related to detection and extraction. Version 2.0 allows configurations only for the number of duplicates, timer settings, and cluster and advanced metrics settings.

This article builds on our previous work [AlOmar et al., 2022; AlOmar et al., 2023a; AlOmar et al., 2024] and extends it with the following novel contributions:

- We extend our tool by designing and implementing code clone detection of both Type-1 and Type-2 (Limitation #1).
- We design our own method extraction strategy to support the refactoring of more variants of code clones by adopting **Program Structure Interface (PSI)**-based detection and developing a custom method extractor capable of detecting and extracting all Type-1 clones and most Type-2 clones. This addresses the limitations of ANTICOPYPASTER 1.0 and 2.0 confined to certain subtypes of Type-1 clone detection (Limitation #2).
- We introduce a new feature for choosing the method extraction approach. Our solution enables users to select between AI model-based and custom extraction heuristics to determine whether a code fragment should be extracted. Previously, ANTICOPYPASTER 1.0 [AlOmar et al., 2022; AlOmar et al., 2023a] supported only the AI model, while ANTICOPYPASTER 2.0 [AlOmar et al., 2024] supported only the user extraction settings (Limitation #3).
- Our approach provides a recommended name for the extracted method using IntelliJ’s built-in name recommender or neural model CODE2VEC (Limitation #4).
- We extend the user settings to give the user more flexibility. Users can now configure extraction judgments, set the maximum number of parameters for an *Extract Method*, choose the name

recommendation model, specify the maximum number of name predictions, and adjust the sensitivity to determine whether a clone should be extracted (Limitation #5).

- We provide ANTiCoPyPASTER 3.0 as an open source tool available to the community [AlOmar et al., 2025]. We also provide a comprehensive replication package that contains the dataset, survey results, and scripts, as well as documentation on how to embed another AI model or name recommendation models into the plugin, allowing researchers and practitioners to customize the tool according to their needs and preferences.

Our study shows the following key findings: (1) the participants expressed high levels of satisfaction with the plugin, (2) our detection method effectively identifies clones by analyzing the structural and semantic aspects of the code, (3) IntelliJ's recommender often provides generic names, lacking context-specific details, while CODE2VEC produces descriptive and relevant names based on the code context, (4) the performance of ANTiCoPyPASTER remains stable regardless of file size and number of clones present, (5) the absence of a systematic method for automatically comparing *Extract Method* plugins poses significant challenges in scaling the sample size, and (6) our results show an improvement in code quality after performing *Extract Method* refactoring with both refactoring tools.

The remainder of this article is organized as follows: Section 2 reviews existing studies related to code clone and *Extract Method* refactoring. Section 3 outlines the ANTiCoPyPASTER new features. Section 4 discusses our evaluation, while research implication is discussed in Section 5. Section 6 captures the limitations of our work before concluding with Section 7.

2 Background and Related Work

2.1 Clone Detection

Code clone detection is an active research area in software engineering [Jiang et al., 2007; Lin et al., 2014; Mondal et al., 2020; Roy et al., 2009; Tairas and Gray, 2009; Tairas and Gray, 2012; Wu et al., 2022; Xing et al., 2011]. Tairas and Gray [2009, 2012] developed CEDAR, a Java Eclipse plugin that incorporates the results of the DECKARD clone detection tool. The tool then sent the clone detection results to the Eclipse refactoring engine to determine the clones that can be refactored. The study concentrated solely on Type-2 clones that exhibit expression differences whose parameterization does not introduce side-effects. Xing et al. [2011] presented a clone analysis tool called CLONEDIFFERENTIATOR. The tool automatically characterized the clones based on their semantic differences by differentiating the Program Dependence Graphs of clones. Lin et al. [2014] proposed an approach as an Eclipse plugin to automatically detect differences across multiple clone instances. Their result showed that the algorithm has a precision of more than 97.66% and a recall of more than 95.63% in three open source Java projects. Mondal et al. [2020] presented a comprehensive study on clone refactoring and tracking. The authors indicated that automatic refactoring cannot eliminate the need for manual effort to find refactoring opportunities and to test system behavior after application of refactoring. Roy et al. [2009] studied two types of similarity between code fragments: fragments can be similar based on the similarity of their program text (Type-1, Type-2, and Type-3) [Bellon et al., 2007] or their functionality (Type-4) [Gabel et al., 2008; Komondoor and Horwitz, 2001]. In the following, we clarify these clone types:

Type-1 [Bellon et al., 2007]: Identical code fragments except for variations in whitespace (Type-1a), comments (Type-1b), and layouts and formatting (Type-1c).

Type-2 [Bellon et al., 2007]: Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout, and comments. It can involve renaming identifiers and

literals in the code fragment (Type-2a), renaming identifiers (but not necessarily systematically) (Type-2b), renaming data types and literal values (Type-2c), and replacing some parameters with expressions (Type-2d).

- Type-3* [Bellon et al., 2007]: Copied fragments with further modifications such as changed, added, or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout, and comments. It can involve making small insertions within a line (Type-3a), small deletions within a line (Type-3b), inserting some new lines (Type-3c), deleting some lines (Type-3d), and making changes to some whole lines (Type-3e).
- Type-4* [Gabel et al., 2008; Komondoor and Horwitz, 2001]: Two or more code fragments that perform the same computation but are implemented by different syntactic variants. It can involve reordering the data-independent declarations (Type-4a), reordering data-independent statements (Type-4b), reordering data-dependent statements (Type-4c), and replacing a control statement with a different one (Type-4d).

Furthermore, the majority of code clones are known to have behavioral differences (i.e., Type-2 and Type-3 clones) [Tsantalis et al., 2017]. Therefore, Lambda expressions can be used to preserve program behavior. Java 8 introduced Lambda expressions to facilitate functional programming and enable behavior parameterization by allowing functions to be passed as arguments to methods [Tsantalis et al., 2017]. Listing 1 shows two duplicated methods `arrangeRF()` and `arrangeFR()` from the JFreeChart project (version 1.0.10). These methods differ semantically by the dimension they process (i.e., width vs. height). After extraction, Listing 2 shows that the duplicated methods are replaced by a common extracted method called `arrangeExtracted()`. To handle the differing expressions (`s.width` and `s.height`, etc.), the parameterization of the methods uses the Lambda expressions that are passed as an argument to the extracted method. Thus, these Lambdas are passed in to parameterize the parts that differ between `arrangeRF()` and `arrangeFR()`. This parameterization preserves the behavior and makes the test pass.

2.2 Extract Method Refactoring

Various studies related to software refactoring have been of importance to both practitioners and researchers. A considerable effort has been spent by the research community on identifying and suggesting *Extract Method* refactorings. The focus of these studies ranges from using program slicing techniques [Maruyama, 2001; Tsantalis and Chatzigeorgiou, 2009; Tsantalis and Chatzigeorgiou, 2011] and graph representations of code [Kanemitsu et al., 2011; Shahidi et al., 2022; Sharma, 2012; Tiwari and Joshi, 2022] to relying on scoring functions to find the most appropriate refactoring candidates [Haas and Hummel, 2016; Silva et al., 2014; Silva et al., 2015; Yang et al., 2009] and using machine learning techniques [Aniche et al., 2020; van der Leij et al., 2021; Xu et al., 2017b; Yue et al., 2018]. We summarize the key studies in Table 1.

Maruyama [2001] developed a semi-automated approach for suggesting refactorings, which decomposed the control flow graph using block-based program slicing. This approach was later adapted and implemented by Tsantalis and Chatzigeorgiou [2009] in the JDEODORANT tool that identified *Extract Method* refactoring opportunities using code slicing along with a set of rules to ensure behavior preservation after slice extraction. In a follow-up work, Tsantalis and Chatzigeorgiou [2011] proposed a set of additional behavior preservation rules that exclude refactoring opportunities related to slices, the extraction of which could possibly cause a change in the program behavior. Meng et al. [2015] developed RASE, a clone removal tool that can apply combinations of six refactorings. *Extract Method* is one of these refactorings used to extract common code guided by systematic edits. Tsantalis et al. [2015, 2017] investigated the utility of Lambda expressions for parameterizing behavioral differences in clones. Their results revealed that Lambda expressions allow the refactoring of a significant portion of clones that could not be refactored by any

```
// First clone pair
protected Size2D arrangeRF(BlockContainer
    container, Graphics2D g2,
    RectangleConstraint constraint) {
    Size2D s = arrangeNF(container, g2,
        constraint);
    if (constraint.getWidthRange().contains(s.
        width)) {
        return s; }
    else {
        RectangleConstraint c = constraint.
            toFixedWidth(
            constraint.getWidthRange().constrain(s.
                getWidth()));
        return arrangeFF(container, g2, c); }
}

// Second clone pair
protected Size2D arrangeFR(BlockContainer
    container, Graphics2D g2,
    RectangleConstraint constraint) {
    Size2D s = arrangeFN(container, g2,
        constraint);
    if (constraint.getHeightRange().contains(s.
        height)) {
        return s; }
    else {
        RectangleConstraint c = constraint.
            toFixedHeight(
            constraint.getHeightRange().constrain(s.
                getHeight()));
        return arrangeFF(container, g2, c); }
}
```

Listing 1. A pair of clones (Before method extraction).

```
protected Size2D arrangeRF(BlockContainer
    container, Graphics2D g2,
    RectangleConstraint constraint){
    return arrangeExtracted(container, g2,
        constraint, arrangeNF(container, g2,
            constraint), (Size2D s) -> s.width,
            constraint.getWidthRange(),
            (Size2D s) -> constraint.toFixedWidth(
            constraint.getWidthRange().
            constrain(s.getWidth())));}

protected Size2D arrangeFR(BlockContainer
    container, Graphics2D g2,
    RectangleConstraint constraint){
    return arrangeExtracted(container, g2,
        constraint, arrangeFN(container, g2,
            constraint), (Size2D s) -> s.height,
            constraint.getHeightRange(),
            (Size2D s) -> constraint.toFixedHeight(
            constraint.getHeightRange().
            constrain(s.getHeight())));}

// Extracted method
protected Size2D arrangeExtracted(
    BlockContainer container, Graphics2D
    g2, RectangleConstraint constraint,
    Size2D arg0, Function arg1, Range arg2
    , Function arg3) {
    Size2D s = arg0;
    if (arg2.contains((double) arg1.apply(s)))
    {
        return s;
    } else {
        RectangleConstraint c = arg3.apply(s);
        return arrangeFF(container, g2, c);
    }
}
```

Listing 2. Using Lambda expressions to enable the refactoring of clones (After method extraction).

Table 1. Related Work in Identifying the *Extract Method* Refactoring Opportunities [AlOmar et al., 2023a]

Study	Year	Tool	Design defect	Plugin?	Detection and correction	Live refactoring	Multi-instance clone
Maruyama [2001]	2001	Not mentioned	Not mentioned	No	✗	✗	✗
Murphy-Hill and Black [2008]	2008	Not mentioned	Not mentioned	No	✗	✗	✗
Tsantalis and Chatzigeorgiou [2009, 2011]	2009 and 2011	JDEODORANT	Long Method	Yes	✓	✗	✗
Yang et al. [2009]	2009	AUTOMeD	Long Method	No	✗	✗	✗
Kanemitsu et al. [2011]	2011	ReAF	Not mentioned	No	✗	✗	✗
Sharma [2012]	2012	N/A	Not mentioned	No	✗	✗	✗
Silva et al. [2014, 2015]	2014 and 2015	JEXTRACT	Not mentioned	Yes	✗	✗	✗
Meng et al. [2015]	2015	RASE	Code Clone	Yes	✓	✗	✗
Tsantalis et al. [2015, 2017]	2015 and 2017	JDEODORANT	Code Clone	Yes	✓	✗	✗
Charalampidou et al. [2016]	2016	SEMI	Long Method	No	✗	✗	✗
Xu et al. [2017b]	2017	GEMS	Not mentioned	Yes	✗	✗	✗
Yue et al. [2018]	2018	CREC	Code Clone	Yes	✗	✗	✗
Yoshida et al. [2019]	2019	Not mentioned	Code Clone	Yes	✗	✗	✗
Aniche et al. [2020]	2020	N/A	Not mentioned	No	✗	✗	✗
van der Leij et al. [2021]	2021	N/A	Not mentioned	No	✗	✗	✗
Shahidi et al. [2022]	2022	N/A	Long Method	No	✗	✗	✗
Tiwari and Joshi [2022]	2022	SEGMENTATION	Long Method	Yes	✗	✗	✗
AlOmar et al. [2022, 2023a, 2024]	2022 and 2023 and 2024	ANTI-COPYPASTER	Code Clone	Yes	✓	✓	✓
Pomian et al. [2024a, 2024b]	2024	EM-ASSIST	Long Method	Yes	✗	✗	✗

other means. In another study, Murphy-Hill and Black [2008] presented three features to improve the adoption and usage of the *Extract Method* refactoring, namely, selection assist, box view, and refactoring annotation. Their formative study shows that user satisfaction was significantly

increased with these features. Sharma [2012] proposed *Extract Method* candidates based on the data and the structure dependency graph. Their suggestions were obtained by eliminating the longest dependency edge in the graph.

Kanemitsu et al. [2011] presented a visualization method for identifying *Extract Method* refactorings and introduced an implementation of the proposed method called REAF. Another approach was proposed by Yang et al. [2009], the authors identified fragments to be extracted from long methods. Their approach is implemented as a prototype called AUTOMED. The evaluation results suggested that the approach may reduce the refactoring cost by 40%.

Silva et al. [2014] used a similarity-based approach to recommend automated *Extract Method* refactoring opportunities that hide structural dependencies that are rarely used by the remaining statements in the original method. Their evaluation on a sample of 81 *Extract Method* opportunities achieved precision and recall rates close to 50% when detecting refactoring instances. In another study, Silva et al. [2015] extended their work by designing an Eclipse plugin called JEXTRACT that automatically identified, ranked, and applied refactorings upon request. Inspired by the study of Silva et al. [2015], Haas and Hummel [2016] developed a scoring function aimed to decrease the complexity of code by considering the code's length and nesting depth. The evaluation against 10 experienced developers showed that they accepted 86% of the suggested refactorings. Another study by Charalampidou et al. [2016] shows the application of functional relevance to detect the *Long Method* code smell. The authors developed a tool called SEMI to automate the application of this approach to Java classes.

Xu et al. [2017b] proposed a plugin called GEMS that used both structural and functional features of code fragments from the real-world *Extract Method* refactorings to train a model to suggest refactorings. GEMS utilized the same extraction algorithm as JEXTRACT [Silva et al., 2015] to create code fragments for extraction. Yue et al. [2018] presented a tool called CREC that combined static analysis and the analysis of code history to suggest code clones that should be extracted in a separate method. Another experiment was conducted using a pro-active clone recommendation system. Yoshida et al. [2019] designed an Eclipse plugin that tracks user code modification and constructed a system to support clone refactoring. When the system detects an *Extract Method* refactoring being performed, it automatically searches for clones of the extracted fragment and suggests extracting them as well. However, this still requires a developer to think about whether a code clone is worth refactoring.

Aniche et al. [2020] used a machine learning approach that involves predicting refactorings using code, process, and ownership metrics. The resulting models predict 20 different refactorings at the levels of a class, method, and variable with an accuracy often higher than 90%. Another experiment that predicts refactorings was conducted using quality metrics. van der Leij et al. [2021] explored the recommendation of the *Extract Method* refactoring in ING. They observed that machine learning models can recommend *Extract Method* refactorings with high accuracy, and the user study reveals that ING experts tend to agree with most of the recommendations of the model.

Shahidi et al. [2022] automatically identified and refactored the *Long Method* code smells in Java code using advanced graph analysis techniques. Their proposed approach was evaluated on five different Java projects. The findings reveal the applicability of the proposed method in establishing the single responsibility principle with a 21% improvement. In another study, Tiwari and Joshi [2022] introduced SEGMENTATION as a graph-based technique to identify *Extract Method* refactoring with the aim of achieving higher performance with fewer refactoring suggestions. The authors compared their approach against two state-of-the-art approaches, i.e., JEXTRACT and SEMI, and showed an improvement over both of them. More recently, Pomian et al. [2024a] combined the insights of **Large Language Models (LLMs)** with the power of IDEs to perform *Extract Method* refactoring. The authors proposed an EM-ASSIST IntelliJ IDEA plugin to remove hallucinations

from the refactoring candidates suggested by LLMs. The results of the study show that EM-ASSIST outperforms previous state-of-the-art tools.

Overall, recommending *Extract Method* refactoring opportunities has been extensively studied [Aniche et al., 2020; Tsantalis and Chatzigeorgiou, 2011; van der Leij et al., 2021; Yoshida et al., 2019]. Although some of the proposed techniques utilized various code metrics as a new way for recommending *Extract Method* refactoring, to the best of our knowledge, no previous studies have proposed a just-in-time automated *Extract Method* refactoring tool to specifically eliminate code clone while using code metrics as features to predict whether a piece of code should be extracted. The *just-in-time* automatic aspect is crucial since it allows developers to remain in the context of the suggestion and thus make a timely decision. This alleviates the burden of reviewing a long list of refactoring opportunities, located in code fragments that are irrelevant to the development context.

To gain a more in-depth understanding of the challenge, increase awareness of code clones, and increase the adoption of *Extract Method* refactorings, in this article, we extend our ANTICOPYPASTER tool to mainly support different types of code clones and integrate context-aware method name recommendations. Our study complements the existing efforts that are carried out to recommend *extraction method* refactorings in general [Aniche et al., 2020; Murphy-Hill and Black, 2008; Sharma, 2012; Silva et al., 2014; van der Leij et al., 2021] or specifically to eliminate certain code smells [Charalampidou et al., 2016; Haas and Hummel, 2016; Shahidi et al., 2022; Tiwari and Joshi, 2022; Tsantalis and Chatzigeorgiou, 2011; Yoshida et al., 2019].

3 ANTiCOPYPASTER Approach

3.1 ANTiCOPYPASTER Key Features

In the following, we summarize the aspects of ANTICOPYPASTER mentioned in Table 1.

Just-in-Time Bundled Detection and Correction. Our tool emphasizes real-time assistance during active development, rather than outsourcing clone detection tools and retrospective clone refactoring.

Live Refactoring. Inspired by the concept of *Live Programming* [Tanimoto, 2013], we put the refactoring concept of liveness into a context. Although live refactoring improves usability and encourages adoption, its implementation requires addressing unique technical and research challenges. These include: efficient real-time refactoring recommendation algorithms that operate seamlessly without noticeable performance degradation, even in large codebases, and the integration with existing development workflows in a way that supports refactoring scenarios, which existing clone refactoring tools typically do not address.

Advancing Clone Refactoring for Multi-Instance Groups. ANTICOPYPASTER ability to handle clone groups with more than two instances is based on this live refactoring feature, enabling real-time support for complex refactoring scenarios. This extends beyond tool adoption by exploring algorithms and heuristics to manage and refactor these larger clone groups effectively in a developer-friendly manner.

3.2 Comparison between ANTiCOPYPASTER 1.0, 2.0, and 3.0

In our extension of ANTICOPYPASTER, we sought to significantly improve the usability and versatility of the plugin, which consists of a new algorithm to detect pasted redundant code and the integration of CODE2VEC [Alon et al., 2019], a neural model to predict the method names of the extracted fragment. Table 2 provides a high-level comparison and evolution of ANTICOPYPASTER that will be described in more detail below.

Table 2. Comparison of ANTICOPYPASTER Versions 1.0, 2.0, and 3.0 across Various Dimensions

Dimension	ANTICOPYPASTER 1.0	ANTICOPYPASTER 2.0	ANTICOPYPASTER 3.0
Clone detection	Exact duplicates	Exact duplicates	Type-1(a), Type-1(b), Type-1(c) Type-2(a), Type-2(b), Type-2(c), Type-2(d)
Clone refactoring	Bag-of-words and direct substring matching	Bag-of-words and direct substring matching	Program Structure Interface (PSI)
Name recommendation	N/A	N/A	IntelliJ built-in and CODE2VEC recommender
Multi-project support	No	Yes	Yes
Method extraction settings	AI model	Custom user extraction	AI model and custom user extraction
User-configurable settings	N/A	Set minimum methods, alert time, and adjust metrics	Set minimum methods, alert time, adjust metrics, max parameters, name predictions, and clone sensitivity

Original Functionality in ANTICOPYPASTER 2.0. Listed below are the original functionality of ANTICOPYPASTER that must be maintained.

- ANTICOPYPASTER is accessible to IntelliJ users as a plugin and integrated into their normal workflow.
- ANTICOPYPASTER is activated from the IntelliJ plugin pane and, while active, will wait to detect a paste action from the user. This uses an AI model to detect code clones.
 - Upon being notified of a paste, the tool alerts the user when a pasted code fragment of Type-1 can be extracted to avoid redundancy.
 - Once the user accepts the notification, they are prompted with:
 - A selectable dropdown pertaining to the visibility of the method (private/public).
 - A textbox to enter the method’s name.
 - A list of the method’s extracted parameters to include in the extraction.
 - A signature preview of the method header.
 - The number of duplicate code fragments the user can replace the extracted method with.
 - An option to continue or cancel the refactoring.
 - Once the user continues, the plugin will ask the user whether they want to: (1) Replace or skip each duplicate occurrence one by one, (2) Replace all occurrences at once, or (3) Cancel the refactoring operation.
- ANTICOPYPASTER allows users to edit the following settings that impact the frequency and timing of its extraction recommendation.
 - Minimum number of methods that contain detected duplicated code before alerting the user (non-functional).
 - Time in seconds after pasting code before the plugin alerts the user (non-functional).
 - Adjust metrics selection, including relevant keywords, coupling, complexity, and size, to fine-tune clone detection.
 - “*Keywords*”: Number of keywords the segment should contain before refactoring is suggested.
 - “*Coupling*”: Number of references made outside of the class before refactoring is suggested.

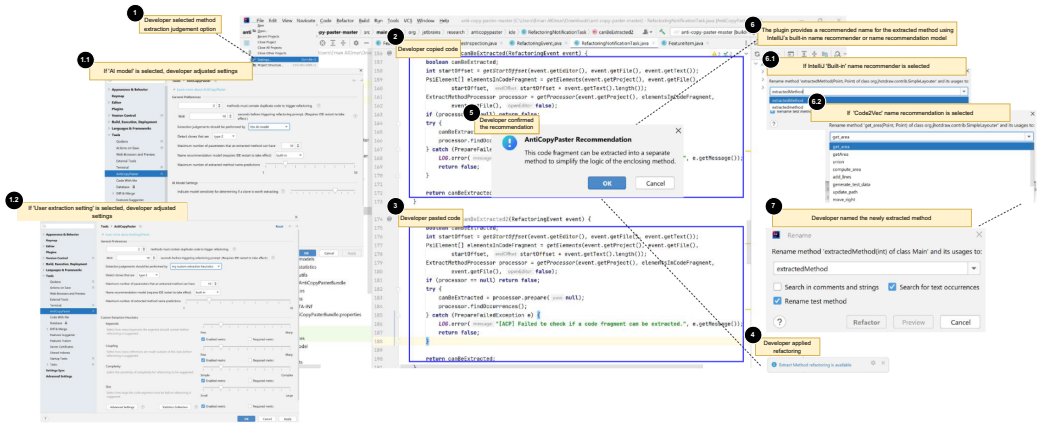


Fig. 1. ANTIcopyPASTER in action, showing the identified *Code Clone* and the recommended *Extract Method* refactoring opportunity.

- “Complexity”: Number representing the sensitivity of complexity for refactoring to be suggested.
- “Size”: **Lines of Code (LOC)** representing how large the code segment must be before refactoring is suggested.

New and Expanded Functionality in ANTIcopyPASTER 3.0. As an illustrative example, Figure 1 shows a step-by-step scenario to reproduce an *Extract Method* refactoring automatically. The user first opens our plugin’s settings ① and selects method extraction settings (i.e., AI model ①.1 or custom extraction heuristics ①.2), along with other settings related to code clone and name recommendation to flag any copy-pasted code. Then, whenever there is an event of copy-pasting code ②, ③, the plugin calculates the metrics associated with the duplicate code. If the metrics values are within the thresholds of the extraction heuristics, the refactoring recommendation is triggered through a pop-up notification ④. If the recommendation is accepted ⑤, the plugin invokes the IDE’s *Extract Method* refactoring with the appropriate parameters and prompts the user to input the name of the extracted method ⑥. Based on the user’s choice, the plugin shows the name of the recommended method from the IntelliJ built-in recommender ⑥.1 or CODE2VEC ⑥.2. The user can confirm the name of the recommended method or enter their own ⑦. Listed below are the objectives we pursued in our work on ANTIcopyPASTER:

- The plugin detects redundant pasted methods using PSI trees.
- The PSI is a concrete syntax tree that represents the entire program.
- See Sections 3.6, 3.7, and 3.8 below for more information on how clones are detected, judged, and extracted.
- The plugin provides a recommended name for the extracted method.
- Users will have the option to select from IntelliJ’s built-in name recommender or neural model CODE2VEC.
- Once the user is prompted with the refactoring window, the plugin will use the preferred model to predict and provide the user with a method name.
- The plugin passes the extracted method body to the Python program via a server that communicates between the Java and Python processes.
- CODE2VEC produces an output list of suggestions ranked by confidence level, which is then passed back to the Java side.

- Once the CODE2VEC output is received and stored on the Java side of the server, it sends the predictions as text to a secondary server that communicates between the extraction event and the modified `JavaNameSuggestionProvider` class file.
- The CODE2VEC suggestions are inserted into the suggestion provider, and finally, the rename refactoring event is called.
- Model suggestions are then selectable by the user, with the predicted name with the highest confidence acting as the default method name.
- The user may choose not to use any of the predicted method names and enter their own.

Overall, the updates to the existing code consist of replacing the functionality for detecting duplicate pasted code. The expanded functionality uses PSI trees to detect and extract a method heuristically, in addition to using the AI model of the previous version. Next, we elaborate on these features that improve the user experience with ANTICOPYPASTER.

3.3 PSI

To find clones, the **Abstract Syntax Tree (AST)** of the entire active file is analyzed via the JetBrains PSI using a pre-order tree traversal. As part of this traversal, two important events happen in terms of dataflow analysis:

- The scope is tracked. Each time the traversal descends into a new level of scope (e.g., for-loop body, if-statement body), a new frame is instantiated (i.e., scoped structures instantiated during code traversal). These frames are passed as parameters to the recursive traversal calls and are always created as new objects, so their additions do not affect the scope higher up in the tree.
- The variable declarations are tracked. Each time a declaration is encountered, it is added to the currently active scope. This means that variables declared in a more narrow scope (e.g., for-loop body) are not considered in a wider scope (containing function, for example).

Using the above mechanics, at any point in the code the in-scope variables of the clone are known. Therefore, to determine whether an expression needs to be a Lambda parameter, its set of used variables just needs to intersect with the current scope. If the intersection yields a non-empty list, scoped variables are used, and a Lambda parameter is needed. Otherwise, a normal parameter can be used. The one time this logic is overridden is if one clone uses clone-scoped variables but another does not for the same positional argument of the extracted function. In this case, since all the clones are merged to create the extracted function, the argument must be a Lambda argument to ensure that each removed duplicate can have a correctly generated method call to the extracted method.

Moreover, to parameterize two or more expressions that are different between clones, they should be evaluated to the same type, which is information available in the PSI. Since Java is statically typed, it is always possible to compute the type of an expression at compile time. The PSI takes advantage of this during its generation and presents types as node properties. The JetBrains PSI information that is used by ANTICOPYPASTER is the type of a node, the text of a node, the children of a node, and the parent of a node. A node is any element of the AST.

3.4 Condition

To determine whether the clone fragments can be extracted, we define a set of conditions that should be considered before the clone refactoring process. Given that Java programs are the focus, these

conditions are tailored to align with the characteristics and constraints of the Java programming language.

- *Only one live variable may be modified within the clone.* A “live variable” is a variable that is defined before the code clone and used after it. Of these, only one variable can be modified within the code clone. This is because Java methods cannot return multiple values (tuples), which complicates handling multiple modified variables in refactoring. Additionally, since Java is pass-by-value, modifying multiple live variables within a clone would require workarounds, such as creating custom objects, which adds complexity and risks errors.
- *No generic type extraction.* Code clones can only be extracted if all their types match precisely. If the code clones involve generic types (e.g., `List <T>`), the extraction cannot proceed.
- *Control flow must be identical.* The control flow within the duplicate fragments (e.g., loops, conditionals) must match exactly. Since our tool supports up to Type-2 clones, we can only parameterize expressions, not statements that are part of Type-3 clone detection.
- *Consistent return type.* If an extracted method returns a value other than void, all clones replaced by the method must return the same type. The lack of generic type support (see condition 3) means that the tool cannot handle cases where clones return different types. This ensures a consistent return type that simplifies the method extraction process.
- *Lambda expression inputs are limited to two variables.* For the extracted expressions, if they reference in-scope variables (i.e., variables declared within the clone fragment), the resulting Lambda expression can only take one or two inputs. This restriction ensures that clones referencing different in-scope variables are extracted since the correct variable can be used in the Lambda body. However, all (up to 2) needed ones are available for each generated method call. Allowing more than two inputs increases complexity and may reduce code clarity.
- *Variables for extracted expressions must be either live-in or in-scope.* Due to Java Lambda limitations, an extracted expression can only use live-in variables (i.e., variables defined before the clone) or in-scope variables (i.e., variables declared within the clone fragment). Java Lambdas use copies of their reference variables outside their scope; hence, mixing live-in and in-scope variables can break the data flow.

As a way of demonstrating PSI and the conditions, we showcase an example in Figure 2. The figure illustrates each major step of the PSI strategy, including clone detection, condition validation, and transformation, indicating that refactoring is valid and safe based on all six conditions.

3.5 ANTIcopyPASTER Tool Architecture

In this section, we describe the specific implementation of our plugin. The plugin consists of the following five main components:

3.5.1 Duplicate Detector. To detect clones, we use PSI clone detection, which is a concrete syntax tree that is used on the IntelliJ Platform to represent the code structure. This approach takes a given code fragment as input, then parses all methods inside the same file as described above. This approach can detect an exact or relative match. Since it is possible that a code fragment will be significantly edited soon after it is pasted, in order to avoid the immediate flagging of the pasted code as a clone and potentially interfering with the developer’s flow, we implement a delay and place the pasted code fragment in a queue. Then, two sanity checks are executed: we check whether the pasted fragment is Java code and whether it constitutes a correct syntactic statement. To do that, the plugin tries to build a PSI tree of the fragment. If a PSI tree can be built and represents a valid statement, and if duplicates still remain after the delay, the code fragment is passed to the Code Analyzer.

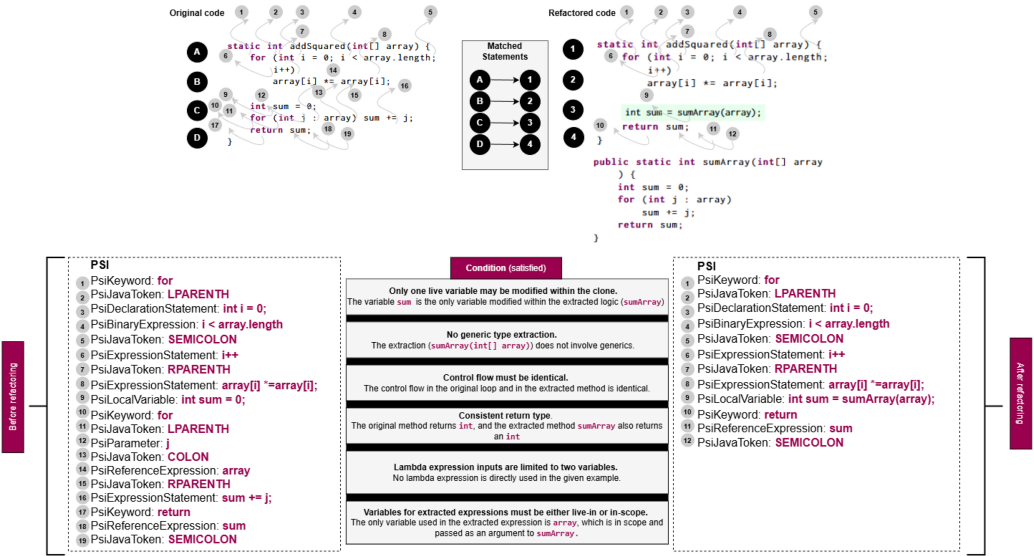


Fig. 2. Statement matching, PSI strategy, and conditions for an *Extract Method* refactoring using ANTIcopyPASTER.

3.5.2 Code Analyzer. This component takes the duplicate fragment as input and uses its PSI representation to calculate the 78 metrics discussed in Section 3.3. The code fragment, with its corresponding vector of metrics, constitutes the input to the Method Extractor.

3.5.3 Method Extractor. This component takes as input the vector of metrics and feeds it to the pre-trained model in order to make the binary decision of whether this code fragment is similar to the ones that have been previously refactored in real projects. If the classifier confirms the refactoring, then Refactoring Launcher is called.

3.5.4 Refactoring Launcher. This component starts with checking if the pasted code fragment could be extracted into a separate method without any compilation errors. If all checks pass, a notification is then enabled to appear in the bottom right corner of the editor, informing the developer that an *Extract Method* refactoring is recommended. If the user responds to the tip, the Refactoring Launcher passes the duplicate fragment as input to the `ExtractionTask` and initiates the preview window. The user has the choice to either confirm the refactoring, while renaming the newly extracted method, or cancel the entire process. If the user accepts the refactoring, Name Recommender is called.

3.5.5 Name Recommender. This component takes the extracted method as an input to recommend names using either the integrated models or the IntelliJ's built-in name recommender that will be discussed in detail in Section 3.9.

3.6 Clone Detection

After a paste event, the pasted text is captured in a `RefactoringEvent`, which is put in the `RefactoringNotificationTask.eventsQueue` to wait out the buffer period before being acted on. Once time is up, the `RefactoringNotificationTask` triggers clone detection through the `DuplicatesInspection` class. Previously, this detection was accomplished through two methods: direct substring matching and bag-of-words. Unfortunately, these methods struggled with most

Type-2 clones, or even some Type-1 clones if they contained different whitespace or comments. These drawbacks motivated the move to PSI-based detection. The PSI is IntelliJ's version of the AST. By traversing the AST of a code, it is possible to programmatically make decisions based on that code's specific content. Implementing our own tree traversal methods for the Java code ANTI-COPY-PASTER inspects therefore gave us much more control over what types of clones we could detect because we can dictate the exact code structures we are looking for.

DuplicatesInspection performs three main tasks:

- It finds the start and end points of the pasted clone fragment and extracts their associated `PsiStatements`.
- It executes the appropriate `CloneProcessor` based on the user's currently selected option in their settings page.
- Any clones that have more than one live-out variable (i.e., variable presented in the clone that are used after it) are removed from the results set since Java does not support tuples and we cannot extract these feasibly without creating additional record types.

For the `CloneProcessor` execution step, here is a more in-depth breakdown.

- All `CloneProcessors` implement the same base interface. This interface provides helper methods and one function that must be implemented: `getClonesOfType`. This function is called by the `DuplicatesInspection`.
- The general flow of a `CloneProcessor` is as follows:
 - `getClonesOfType` identifies all possible statements that could be the start of a clone and iterates through them.
 - Each potential clone has its PSI preorder tree traversed at the same time as the pasted fragment. This enables easy comparison between the two.
 - As each element is processed, if any information is encountered that could be useful later, it is stored in a `MatchState` (live-in variables (i.e., variables present in the clone that were declared before it), live-out variables (i.e., variables present in the clone that are used after it), variable types, elements that can be made into parameters, etc.).
 - This information varies per clone type and as such this step is where the differences are between different Clone Processors. For Type-1 clones, nothing is needed besides direct text matching. For Type-2 clones, elements that should be parameters have to be detected and stored in the `MatchState`.
 - Any potential clone that turns out to be an actual clone is added to a results list that is then returned by `getClonesOfType`.

3.6.1 Type-1 Clone. Algorithm 1 takes as input the pasted code fragment that needs to be checked for clones, detects exact copies, except for differences in whitespace, comments, and formatting (i.e., Type-1 clones) of a given code fragment *F* from a list of potential clones *M*. It initializes an empty list results to store the detected clones and their details (line 1). The algorithm then iterates through each potential clone candidate *match* in *M* (line 2). Two new instances of `MatchState`, named "fragMS" and "dupeMS," are created (lines 3–4). These objects will be used to store the state of the current matching process for the fragment and the potential duplicate, respectively. The function `isType1DuplicateAt` is called with the start and end positions of *F* and the current *match*, along with the match states "fragMS" and "dupeMS." This function checks if *match* is a Type-1 duplicate of *F*. If a duplicate is found, the function returns the "end" position of the match; otherwise, it returns "null" (line 5). If "end" is not "null," indicating that a Type-1 clone is found, the algorithm proceeds to gather details about the match and adds it to the "results" list. The details

Algorithm 1: Type-1 Clone Detection

Input: F , the pasted code fragment, M , the list of PSI Elements that could be clones
Output: $results$, list of clones with extraction details

```

1 results  $\leftarrow \emptyset$ 
2 foreach  $match \in M$  do
3   fragMS  $\leftarrow$  new MatchState
4   dupeMS  $\leftarrow$  new MatchState
5   end  $\leftarrow$  isType1DuplicateAt( $F.start$ ,  $F.end$ ,  $match$ , fragMS, dupeMS)
6   if end is not null then
7     results.add(
8       match.startStatement,
9       matchEnd,
10      liveOut(matchEnd, dupeMS.scope),
11      dupeMS.parameters,
12      dupeMS.aliasMap,
13      dupeMS.typeParams, dupeMS.liveIn,
14      dupeMS.extractable
15    )
16   end
17 end
18 return results
19 Function isType1DuplicateAt
20   Input:  $fragStart$ , the start of the pasted fragment,  $fragEnd$ , the last statement of the pasted
21     fragment,  $dupeStart$ , the start of the potential duplicate,  $fragMS$  and  $dupeMS$ , the match
22     states of the fragment and duplicate
23   Output: the last element in a duplicate code fragment
24   if  $fragStart$  is null then
25     return null
26   end
27   fragCurr  $\leftarrow$  fragStart
28   dupeCurr  $\leftarrow$  dupeStart
29   while  $fragCurr$  and  $dupeCurr$  not null do
30     if not exactMatch( $fragCurr$ ,  $dupeCurr$ ,  $fragMS$ ,  $dupeMS$ ) then
31       return null
32     end
33     if  $fragCurr == fragEnd$  then
34       break
35     end
36     fragCurr  $\leftarrow$  nextNonWhitespaceOrCommentSibling
37       ( $fragCurr$ )
38     dupeCurr  $\leftarrow$  nextNonWhitespaceOrCommentSibling
39       ( $dupeCurr$ )
40   end
41   return dupeCurr

```

added to “results” include: “match.startStatement”: The start statement of the match, “matchEnd”: The end position of the match, liveOut(“matchEnd,” dupeMS.scope”): Live out variables at the end of the match within the scope of “dupeMS,” “dupeMS.parameters”: Parameters involved in the match, “dupeMS.aliasMap”: Aliases used in the match, “dupeMS.typeParams”: Type parameters in the match, “dupeMS.liveIn”: Live in variables for the match, “dupeMS.extractable”: Information on

Algorithm 2: Exact Match—exactMatch

Input: f , an element in the pasted code fragment, d , an element in a potential duplicate fragment, fMS , matchState for the pasted fragment, dMS , match state for the potential duplicate

Output: whether or not the two fragments are an exact match (i.e. Type 1 clones)

```

1 if  $f$  is null or  $d$  is null then
2   | return false
3 end
4 childrenF  $\leftarrow$  nonWhitespaceOrCommentChildElements( $f$ )
5 childrenD  $\leftarrow$  nonWhitespaceOrCommentChildElements( $d$ )
6 if childrenF.size  $\neq$  childrenD.size then
7   | return false
8 else if childrenF is empty then
9   | return  $f.text == d.text$ 
10 childFMS = fMS.extend()
11 childDMS = dMS.extend()
12 updateScope( $f$ , fMS, childFMS)
13 updateScope( $d$ , dMS, childDMS)
14 foreach ( $childF$ ,  $childD$ )  $\in$  ( $childrenF$ ,  $childrenD$ ) do
15   | if not exactMatch( $childF$ ,  $childD$ ,  $childFMS$ ,  $childDMS$ ) then
16     | return false
17   | end
18 end
19 return true

```

whether the match can be extracted (lines 6–17). Finally, the algorithm returns the “results” list containing all the detected Type-1 clones and their details (line 18).

The function `isType1DuplicateAt` (line 19) aims to determine if a code fragment (starting at “fragStart” and ending at “fragEnd”) is a Type-1 duplicate of another code fragment starting at “dupeStart.” It takes as input (*fragStart*: The starting node of the pasted code fragment, *fragEnd*: The ending node of the pasted code fragment, *dupeStart*: The starting node of the potential duplicate code fragment, *fragMS*: The match state of the fragment, *dupeMS*: The match state of the duplicate, and outputs the last element in the duplicate code fragment if the fragments match). If “fragStart” is “null,” the function returns “null,” indicating that there is no fragment to compare (lines 20–22). It initializes “fragCurr” to the start of the fragment and “dupeCurr” to the start of the potential duplicate (lines 23–24). The loop continues as long as both “fragCurr” and “dupeCurr” are not “null” (line 25). If “fragCurr” and “dupeCurr” do not match exactly, return “null,” indicating that the fragments are not duplicates (lines 26–28). If “fragCurr” has reached “fragEnd,” break out of the loop, meaning that the entire fragment has been matched (lines 29–31). Then, move “fragCurr” and “dupeCurr” to their next siblings that are not whitespace or comments (lines 32–34). After exiting the loop, return “dupeCurr” (line 35). If the loop completes without returning “null,” “dupeCurr” will point to the last matched element of the duplicate fragment.

To determine whether two code fragments are exact matches, outlined in Algorithm 2, it takes into account the structure and content of the code fragments (f : An element in the pasted code fragment, d : An element in a potential duplicate fragment, fMS : Match state for the pasted fragment, and dMS : Match state for the potential duplicate fragment) and outputs a Boolean indicating whether the two fragments are an exact match. If either f or d is null, return “false” since a match cannot be determined if one of the fragments is missing (lines 1–3). Then, it retrieves the child elements and compares child sizes (lines 4–5). If the number of child elements in “childrenF” is not equal to the number of child elements in “childrenD,” return “false” because the structures are different

(lines 6–7). Next, it checks for an empty child; if both “childrenF” and “childrenD” are empty, compare the text of f and d . If they are equal, return “true”; otherwise, return “false” (lines 8–9). The next step involves creating new match states for child elements by extending “fMS” and “dMS,” resulting in “childFMS” and “childDMS” (lines 10–11). `updateScope` is called to adjust the scope of f and d with their respective match states (lines 12–13). Then iterates through pairs of child elements (“childF” from “childrenF” and “childD” from “childrenD”). For each pair, it calls `exactMatch` recursively with the child elements and their match states. If any pair does not match (`exactMatch` returns “false”), return “false” (lines 14–18). Finally, if all child elements match, return “true” (line 19).

3.6.2 Type-2 Clone. Algorithm 3 identifies Type-2 clones (i.e., code with possible renaming of variables, changes in literals, and different formatting) of a given code fragment F from a list of potential clones M . Following the steps listed in Algorithm 1, it initializes match states, verifies duplicates, gathers detailed information for each match, and returns a list of these matches along with their corresponding details. We elaborate below on the differences.

The function `isType2DuplicateAt` (line 19) determines if a code fragment (starting at “fragStart” and ending at “fragEnd”) is a Type-2 duplicate of another code fragment starting at “dupeStart.” It inputs (*fragStart*: The starting node of the pasted code fragment, *fragEnd*: The ending node of the pasted code fragment, *dupeStart*: The starting node of the potential duplicate code fragment, *fragMS*: The match state of the fragment, *dupeMS*: The match state of the duplicate, and outputs the last element in the duplicate code fragment if the fragments match). If “fragStart” is “null,” the function returns “null,” indicating that there is no fragment to compare (lines 20–22). It initializes “fragCurr” to the start of the fragment and “dupeCurr” to the start of the potential duplicate (lines 23–24). The loop continues as long as both “fragCurr” and “dupeCurr” are not null (line 25). If “fragCurr” and “dupeCurr” do not match using the `matchStack` function, return “null” (lines 26–28). `matchStack` is used on each element in the PSI tree of the pasted fragment and the potential duplicate fragment to check whether their `MatchState` fragments match semantically. It is used recursively on the children of the current element to ensure that it is a deep match. If “fragCurr” has reached “fragEnd,” break out of the loop, meaning that the entire fragment has been matched (lines 29–31). Then move “fragCurr” and “dupeCurr” to their next siblings that are not whitespace or comments (lines 32–34). After exiting the loop, return “dupeCurr” (line 35). If the loop completes without returning “null,” “dupeCurr” will point to the last matched element of the duplicate fragment.

Algorithm 4 is designed to determine whether two code fragments match in a way that indicates a Type-2 clone. The inputs required are identical to those needed by the Type-1 Algorithm 2 and outputs a Boolean indicating whether the two fragments’ stacks match. It first performs the null check, if either f or d is null, return “false” since a match cannot be determined if one of the fragments is missing (lines 1–3). Next, it determines whether f and d can be considered parameters by calling `canBeParam` on each with their respective match states (“fMS” and “dMS”). If both “canBeParamA” and “canBeParamB” are successful and the size of their “liveInDeps” is the same: it adds f and d as parameters to “fMS” and “dMS,” respectively, including their types, Lambda arguments, and live-in dependencies, and returns “true” since the fragments match as parameters (lines 4–12). It then retrieves children elements, “childrenF” and “childrenD” are obtained by calling `nonWhitespaceOrCommentChildElements` on f and d , respectively. This function extracts child elements of f and d (lines 13–14). The next step involves comparing children sizes if the number of child elements in “childrenF” is not equal to the number of child elements in “childrenD,” return “false” because the structures are different (lines 15–16). If both “childrenF” and “childrenD” are empty: get alias IDs for f and d using “getAliasId” (i.e., the function “getAliasId” is responsible for determining whether a given PSI element, specifically an identifier, has already been seen in

Algorithm 3: Type-2 Clone Detection**Input:** F , the pasted code fragment, M , the list of PSI Elements that could be clones**Output:** $results$, list of clones with extraction details

```

1 results  $\leftarrow \emptyset$ 
2 foreach  $match \in M$  do
3   fragMS  $\leftarrow$  new MatchState
4   dupeMS  $\leftarrow$  new MatchState
5   end  $\leftarrow$  isType2DuplicateAt( $F.start$ ,  $F.end$ ,  $match$ , fragMS, dupeMS)
6   if end is not null then
7     results.add(
8       match.startStatement,
9       matchEnd,
10      liveOut(matchEnd, dupeMS.scope),
11      dupeMS.parameters,
12      dupeMS.aliasMap,
13      dupeMS.typeParams, dupeMS.liveIn,
14      dupeMS.extractable
15    )
16   end
17 end
18 return results
19 Function isType2DuplicateAt
20   Input:  $fragStart$ , the start of the pasted fragment,  $fragEnd$ , the last statement of the pasted
21     fragment,  $dupeStart$ , the start of the potential duplicate,  $fragMS$  and  $dupeMS$ , the match
22     states of the fragment and duplicate
23   Output: the last element in a duplicate code fragment
24   if  $fragStart$  is null then
25     return null
26   end
27   fragCurr  $\leftarrow$  fragStart
28   dupeCurr  $\leftarrow$  dupeStart
29   while  $fragCurr$  and  $dupeCurr$  not null do
30     if not matchStack( $fragCurr$ ,  $dupeCurr$ ,  $fragMS$ ,  $dupeMS$ ) then
31       return null
32     end
33     if  $fragCurr == fragEnd$  then
34       break
35     end
36     fragCurr  $\leftarrow$  nextNonWhitespaceOrCommentSibling
37       ( $fragCurr$ )
38     dupeCurr  $\leftarrow$  nextNonWhitespaceOrCommentSibling
39       ( $dupeCurr$ )
40   end
41   return dupeCurr

```

the current scope and, if so, returning its associated alias ID). It returns “true” if the text of f is equal to the text of d , or if the alias IDs are the same and greater than zero (lines 17–20). Then, it creates clones of “fMS” and “dMS” to “childFMS” and “childDMS” (lines 21–22), followed by calling `updateScope` to adjust the scope of f and d with their respective match states (lines 23–24). If f and d are PSI-type elements: it adds their types to “fMS.typeParams” and “dMS.typeParams.” If the

Algorithm 4: Relative Match—matchStack

Input: f , an element in the pasted code fragment, d , an element in a potential duplicate fragment, fMS , matchState for the pasted fragment, dMS , match state for the potential duplicate

Output: whether or not the two fragments' stacks match (i.e. a Type-2 Clone)

```

1 if  $f$  is null or  $d$  is null then
2   | return false
3 end
4 canBeParamA  $\leftarrow$  canBeParam( $f$ ,  $fMS$ )
5 canBeParamB  $\leftarrow$  canBeParam( $d$ ,  $dMS$ )
6 if canBeParamA.success and canBeParamB.success then
7   | if canBeParamA.liveInDeps.size == canBeParamB.liveInDeps.size then
8     |    $fMS.addParameter(f, canBeParamA.type, canBeParamA.lambdaArgs,$ 
9       |    $canBeParamA.liveInDeps)$ 
10    |    $dMS.addParameter(f, canBeParamB.type, canBeParamB.lambdaArgs,$ 
11      |    $canBeParamB.liveInDeps)$ 
12    |   return true
13  | end
14 end
15 childrenF  $\leftarrow$  nonWhitespaceOrCommentChildElements( $f$ )
16 childrenD  $\leftarrow$  nonWhitespaceOrCommentChildElements( $d$ )
17 if childrenF.size != childrenD.size then
18   | return false
19 else if childrenF is empty then
20   | idF  $\leftarrow$  getAliasId( $f$ )
21   | idD  $\leftarrow$  getAliasId( $d$ )
22   | return ( $f.text == d.text$ ) or ( $idF == idD$  and  $idF > 0$ )
23 childFMS =  $fMS.clone$ 
24 childDMS =  $dMS.clone$ 
25 updateScope( $f$ ,  $fMS$ , childFMS)
26 updateScope( $d$ ,  $dMS$ , childDMS)
27 if  $f$  and  $d$  are PSI type elements then
28   |  $fMS.typeParams.add(f.type)$ 
29   |  $dMS.typeParams.add(d.type)$ 
30   | if  $f.type \neq d.type$  then
31     |  $dMS.setExtractable(false)$ 
32   | end
33   | return true
34 end
35 foreach ( $childF, childD$ )  $\in$  ( $childrenF, childrenD$ ) do
36   | if not matchStack( $childF, childD, childFMS, childDMS$ ) then
37     | return false
38   | end
39 end

```

types of f and d are not the same, mark “dMS” as not extractable and return “false.” If they are the same, proceed (lines 25–32). Finally, it iterates through pairs of child elements (“childF” from “childrenF” and “childD” from “childrenD”). For each pair, it calls matchStack recursively with the child elements and their cloned match states. If any pair does not match (matchStack returns “false”), return “false” (lines 33–37).

3.7 Fragment Judgment

Before clones can be extracted, the paste fragment is put through a “judgment” to determine if it is worth extracting. This is all a part of `RefactoringNotificationTask`. We currently have two judgment methods available:

- Custom extraction heuristics, allowed user to adjust metrics selection including relevant keywords, coupling, complexity, and size to fine-tune clone detection.
- An AI model, trained on thousands of method extractions from Apache projects [AlOmar et al., 2022; AlOmar et al., 2023a]. The model uses features from the pasted code such as the number of lines, keywords, symbols, and more. The full list of features can be found in `metrics/features/Feature.java`.

Every time a `RefactoringEvent` is processed, the currently selected judgment method is retrieved from the user’s configuration settings. This judgment method refers to the decision-making algorithm used to assess whether a refactoring opportunity should be suggested. It is applied to a feature vector, which is a structured representation of key characteristics extracted from the source code, that are derived from the textual and structural analysis of the code involved in the event. If the judgment is positive, the user is then shown a notification prompting extraction (if enough clones are found).

3.8 Method Extraction

Should the user accept the proposed refactoring advice, a new `ExtractionTask` is created. This task is responsible for generating the new extracted method, as well as replacing all the clones and the pasted fragment with appropriate method calls to the extracted method.

Previously, IntelliJ’s built-in extractor was used by the plugin. Unfortunately, the built-in extractor is inflexible; it cannot be extended, it works only with the limited types of clones it supports and it detects them itself; it cannot be provided with pre-detected clones. In order to upgrade the plugin with supporting more clone-types, it was necessary for us to fully replace this built-in extractor with our own designed extraction algorithm that inherently relies on our detection algorithms, then extract all Type-1 clones and most Type-2 clones.

To create the new method with its corresponding calls, metadata from the clone detection step is necessary. This metadata is passed from detection to extraction and includes information, such as each clone’s live-in variables (i.e., variables present in the clone that were declared before it), live-out variables (i.e., variables present in the clone that are used after it), and a list of fragments within the clone that should become parameters. All of this occurs in a few steps:

- The user is walked through prompts to select which clones they would like to extract.
- A default method name is generated (avoiding names already in use by the user’s code).
- The selected clones are then run through two minimizers: a parameter minimizer and a type-parameter minimizer (currently not used but left in for future expansion to Type-2c). These minimizers remove parameters that are the same across every clone, since that means they do not need to support variability and can be hard coded in the extracted method.
- The Lambda arguments have their arguments combined across clones. One clone may use one variable from within its own scope, while another clone may use a different variable; since the method signature must be the same for all clones because of the extraction refactoring, the entire point of this plugin, the Lambda needs to support an argument for each possible variable that might be used in the Lambda body. Thus, argument combination. This part is done using sets to store immutable unique elements and prevent duplicates from being present in the argument list.

- The return type of the extracted method is determined. Essentially, this involves identifying the first clone that has a live-out variable and then verifying that all other clones with a live-out variable have that variable as the same type as the first one. If all clones have no live-out variables, or just one of the same type, the clones can be extracted, and in the extracted method's final line, that variable will be returned. However, the extraction is canceled if two clones have different live-out variable types, as the extracted method cannot return two different data types.
- The appropriate modifiers are computed for the method (if it should be static).
- The extracted method is now created using all the information gathered in the previous steps. This is done by taking one of the clones and using them as a template. This template clone is then subjected to a syntax tree traversal. Parts of the method that change between clones are converted to parameters so that the appropriate method calls can be generated to preserve the functionality of each clone. As part of this process, Lambda invocations are constructed, the method signature is formulated, and import statements for function types are included for the Lambdas.
- Based on the user settings, the name of the extracted method can be predicted by our model or can just simply be default-called "extractedMethod" when using the IntelliJ built-in recommender. Our model can generate a set of possible method names, all based on the information inside the code in the method generated in the step above.
- The generated method is then added to the bottom of the class the user is currently working in. If the generated method requires Lambdas in its parameter list, which are called `Function<>` and `BiFunction<>` in Java, the necessary imports are added to the end of any preexisting imports.
- Method calls are generated for each clone, removing the clone itself from the document at the same time. This includes the generation of any Lambdas that are required.

Algorithm 5 takes as input the list of Type-1 or Type-2 detected clones, the Java file into which the new method will be added, and the method within which the code fragment was originally pasted. The algorithm updates the given Java file by adding a new method and replacing the clones with calls to this new method (lines 1–8). "normalizedLambdaArgs" is initialized as a new `ArrayList` to store normalized parameters for Lambda expressions (line 9). To remove duplicate parameters, the algorithm iterates over each parameter in the first clone's parameter list ("results[0].parameters"). For each parameter, it checks if this parameter exists in the parameters of any subsequent clones ("results[1:]"). If a parameter exists in other clones, it is removed from the old clone's parameter list and added to the live-in variables of the clone. For each clone in the results, duplicate parameters are removed, and normalized Lambda arguments are added to "normalizedLambdaArgs" (lines 10–13). To determine the return type, initially, "returnType" is set to null. The algorithm checks if all clones have the same type for their first live-out variable. If not, it returns immediately (indicating that the clones cannot be extracted into a single method due to type inconsistencies). If the types are consistent, "returnType" is set to the type of the first live-out variable of the first clone (lines 15–20). Then, "methodName" is generated using the model (this could be a custom function or the model) (line 21). Also, "isStatic" is set on the basis of whether the containing method is static (line 22). To build a new method, "newPSIMethod" is constructed using the first clone, the determined return type, normalized Lambda arguments, the predicted method name, and whether the method is static. The new method is then added to the Java file (lines 23–27). For each clone in "results," the original code fragment is replaced with a call to the newly created method, using the normalized Lambda arguments (lines 28–30). It is worth noting that with the replacement of

Algorithm 5: Method Extraction

Input: *results*, the results from Type 1 or 2 Clone Detection, *file*, the Java file the code fragment was pasted into, *containingMethod*, the method the code fragment was pasted into

```

1 foreach param  $\in$  results[0].parameters do
2   if  $\exists p \in \text{results}[1:].\text{parameters} : p == \text{param}$  then
3     foreach clone  $\in$  results do
4       removedP  $\leftarrow$  oldclone.remove(param)
5       clone.liveInVars.add(removedP.liveInDeps)
6     end
7   end
8 end
9 normalizedLambdaArgs  $\leftarrow$  new ArrayList
10 foreach clone  $\in$  results do
11   //remove any duplicate parameters
12   normalizedLambdaArgs.add
13   (Set(clone.parameters.lambdaArgs))
14 end
15 // All clones in results have at most one element in liveOutVars due to earlier
   filtering, so only the first element needs to be compared across all clones. Since
   all the clones are being merged into one extracted method, this conditional checks
   that they all return the same type, otherwise it aborts extraction as it isn't
   possible to merge two clones with two different return types since Java is statically
   typed.
16 returnType  $\leftarrow$  null
17 if  $\forall c_i \in \text{results}, \exists c_j \in \text{results} : c_i.\text{liveOutVars}[0].\text{type} \neq c_j.\text{liveOutVars}[0].\text{type}$  then
18   return
19 else
20   returnType  $\leftarrow$  results[0].liveOutVars[0].type
21 end
22 methodName  $\leftarrow$  namePredictionModel()
23 isStatic  $\leftarrow$  containingMethod.isStatic
24 newPSIMethod  $\leftarrow$  buildMethodText(
25   results[0], returnType, normalizedLambdaArgs,
26   methodName, isStatic
27 )
28 file.addMethod(newPSIMethod)
29 foreach clone  $\in$  results do
30   generateMethodCall(clone, normalizedLambdaArgs, methodName)
31 end

```

the built-in method extraction with our custom extraction, the default visibility for the extracted method is set to “private,” as our primary focus is on intrafile clone detection.

Example. Types 1a (additional whitespace), 1b (varying comments), and 1c (identical clone) were all detected the same exact way: from each potential start point in the PSI, the syntax was stepped through element by element ignoring all whitespace and comments (if they were encountered, the traversal would immediately skip them). If all non-skipped elements were identical, the clone was successfully identified as Type-1. An example of this can be seen in Listings 3 and 4. Listing 3 shows a sample pasted fragment of code in the main method, and three labeled different clones of the pasted sample above. After accepting the automatic refactoring, the resulting code is present in Listing 4. Additionally, if output is needed from the extracted method, a new variable is created automatically. Similarly, Listings 5 and 6 illustrate another pasted fragment with three more clones present above

```

public class TypeOne {
    // Includes Type-1a clone
    static int addSquared(int[] array) {
        for (int i = 0; i < array.length; i++)
            array[i] *= array[i];

        int sum = 0;
        for (int j : array) sum += j;
        return sum;
    }

    // Includes Type-1b clone
    static int addTripled(int[] array) {
        for (int i = 0; i < array.length; i++)
            array[i] *= 3;

        int sum = 0;
        // Add everything together here
        for (int j : array) sum += j;
        return sum;
    }

    // Includes Type-1c clone
    static int addDoubled(int[] array) {
        for (int i = 0; i < array.length; i++)
            array[i] *= 2;

        int sum = 0;
        for (int j : array)
            sum += j;
        return sum;
    }

    public static void main(String[] args)
    {
        int[] array = { 1, 2, 3, 4, 5 };
        //- The code below here was pasted
        -
        int sum = 0;
        for (int j : array)
            sum += j;
    }
}

```

Listing 3. Before method extraction (Type-1 clone).

```

public class TypeOne {
    // Includes Type-1a clone
    static int addSquared(int[] array) {
        for (int i = 0; i < array.length; i++)
            array[i] *= array[i];

        int sum = sumArray(array);
        return sum;
    }

    // Includes Type-1b clone
    static int addTripled(int[] array) {
        for (int i = 0; i < array.length; i++)
            array[i] *= 3;

        int sum = sumArray(array);
        return sum;
    }

    // Includes Type-1c clone
    static int addDoubled(int[] array) {
        for (int i = 0; i < array.length; i++)
            array[i] *= 2;

        int sum = sumArray(array);
        return sum;
    }

    public static void main(String[] args)
    {
        int[] array = { 1, 2, 3, 4, 5 };
        //- The code below here was pasted
        -
        sumArray(array);
    }

    public static int sumArray(int[] array)
    {
        int sum = 0;
        for (int j : array)
            sum += j;
        return sum;
    }
}

```

Listing 4. After method extraction (Type-1 clone).

it: Type-2a (renamed identifiers), 2b (varying identifiers), and 2d (varying expressions). Notice that where necessary, Lambdas are created to preserve functionality.

3.9 Name Recommendation

We reviewed the literature on method name prediction models (CODE2VEC [Alon et al., 2019], CODE2SEQ [Alon et al., 2018], DEEPNAME [Li et al., 2021], MNIRE [Nguyen et al., 2020], and COGNAC [Wang et al., 2021]) to determine their effectiveness and usability in the ANTIcopyPASTER plugin. However, not all of these models are equally feasible. For all these models, unless trained ones can be found, training is required. Also, almost all of these models have a similar issue of needing the

```

public class TypeTwo {
    static int add(int a, int b) {
        return a + b;
    };

    // Includes Type-2a clone
    static int addDoubled(int[] array) {
        for (int i = 0; i < array.length; i++)
            array[i] *= 2;

        int sum = 0;
        for (int k : array)
            sum += add(sum, k);
        return sum;
    }

    // Includes Type-2b clone
    static int addTripled(int[] array) {
        for (int i = 0; i < array.length; i++)
            array[i] *= 3;

        int sum = 0;
        for (int j : array)
            sum += add(j, sum);
        return sum;
    }

    // Includes Type-2d clone
    static int addSquared(int[] array) {
        int sum = 0;
        for (int j : array)
            sum += add(sum, j * j);
        return sum;
    }

    public static void main(String[] args)
    {
        int[] array = { 1, 2, 3, 4, 5 };
        //- The code below here was pasted
        -
        int sum = 0;
        for (int j : array)
            sum += add(sum, j);
    }
}

```

Listing 5. Before method extraction (Type-2 clone).

```

public class TypeTwo {
    // Includes Type-2a clone
    static int addDoubled(int[] array) {
        for (int i = 0; i < array.length; i++)
            array[i] *= 2;

        int sum = reduceArray((sumArg, k)
        -> sumArg, (sumArg, k) -> k, array);
        return sum;
    }

    // Includes Type-2b clone
    static int addTripled(int[] array) {
        for (int i = 0; i < array.length; i++)
            array[i] *= 3;

        int sum = reduceArray((sumArg, j)
        -> j, (sumArg, j) -> sumArg, array);
    }

    // Includes Type-2d clone
    static int addSquared(int[] array) {
        int sum = reduceArray((sumArg, j) ->
        sumArg, (sumArg, j) -> j*j, array);
        return sum;
    }

    public static void main(String[] args)
    {
        int[] array = { 1, 2, 3, 4, 5 };
        //- The code below here was pasted
        -
        reduceArray((sum, j) -> sum,
        (sum, j) -> j, array);
    }

    public static int reduceArray(
        BiFunction<Integer, Integer,
        Integer> p1, BiFunction<Integer,
        Integer, Integer> p2, int[] array)
    {
        int sum = 0;
        for (int k : array)
            sum += add(p1.apply(sum, k),
            p2.apply(sum, k));
        return sum;
    }
}

```

Listing 6. After method extraction (Type-2 clone).

plugin to be larger in size. Adding more models could increase the size of the plugin. Ultimately, we successfully implemented one of the above models, CODE2VEC, since it is the only model we are able to reproduce. Upon integration CODE2VEC, we have designed and developed an expandable architecture within the plugin to support developers with integrating their own method-name recommendation models. Future models can be integrated into the plugin workflow following the steps below for the CODE2VEC implementation.

For name predictions to function correctly, communication between the Java code in the plugin and the Python code in CODE2VEC is required, alongside the pre-trained name recommendation

model. This communication is configured using local **Transmission Control Protocol (TCP)** communication on port 8081. Additionally, to override and feed into `JavaNameSuggestion`, another port was required at 8082. For `CODE2VEC` to operate within the plugin, communication must be established at launch, and the model must be downloaded to ensure proper functionality. IntelliJ `plugin.xml` specifies this behavior with `postStartupActivity`, which was established as `AntiCopyPasterTelemetry` for this project. `AntiCopyPasterTelemetry` was modified if the `CODE2VEC` model is enabled:

- Start the `predHolder` thread, the server on port 8082, which will connect the extraction task client predictions to the overridden `JavaNameSuggestionProvider`.
- The file checks to see if the plugin files have the pretrained model within them. If they do not, this will trigger a notification. Once the download is finished, there will be another notification sent notifying the user.
- Start the ACP Server thread, which will be the server on port 8081. This thread will connect the `CODE2VEC` client and any extraction task client. If the pre-trained name recommendation model is not present, it will be downloaded to the `java14m_model` directory in `CODE2VEC` via the `downloadModel()` function. ACP Server.

As stated above, a server is needed to connect the `CODE2VEC` client to the extraction task client. This server must also start running the `CODE2VEC` model to make it available. ACP Server does this by following these steps:

- Check what OS the plugin is being run on.
- Depending on the OS given, it then checks to see if Python is installed in the default/common path. This is needed as starting the model requires Java's Process Builder, which uses the terminal. IntelliJ runs as a temporary user on the system, making the user running IntelliJ's environment variables inaccessible. Because of this, if Python's path is not a global environment variable, the temporary user will be unable to find it. It checks if Python is in any of those installation paths to circumvent this. If it finds it, it will continue; if not, it will stop the execution.
- Once the needed dependency is found, it will then start the server on port 8081.
- The plugin will find itself using the built-in IntelliJ `PluginId`. It then uses the found Python path and Plugin path to execute the command in Process Builder to start the model.
- The server then connects to the newly created `CODE2VEC` process and connects itself to its input and output stream.
- An initial suggestion must be provided to load the model itself. A placeholder function is passed through the Java Extractor, so it is in a usable form for the model. This uses all the default parameters of the Java Extractor that `CODE2VEC` used which is also used for the extraction task.
- The server sends this function to `CODE2VEC` where it will be processed and then sent back as a placeholder prediction, loading the model for future uses.
- The server will continue to run in a loop, accepting new clients, which will be extraction task clients. The server feeds the function into `CODE2VEC`, receives the prediction, and sends it back.

CODE2VEC Alterations. Many alterations to `CODE2VEC` were required to make it work outside of the terminal and run continuously. Figure 3 shows an overview of the architecture for `CODE2VEC` with the alterations. Below, we list these changes:

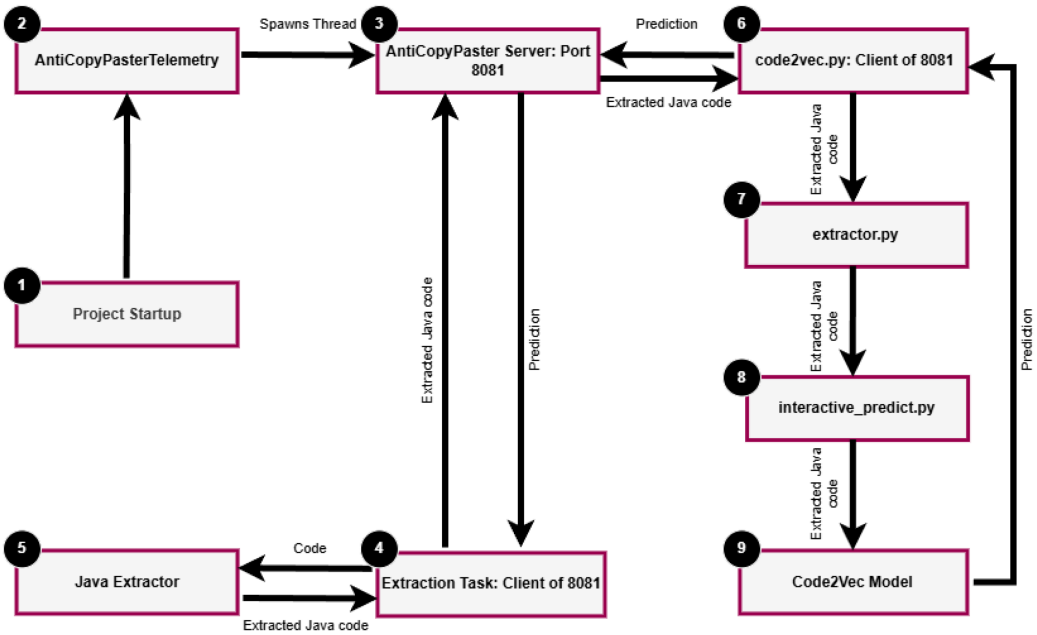


Fig. 3. Adaptation of CODE2VEC in the context of *Extract Method* refactoring.

- Unnecessary files related to C# and training the model were removed as they would not be necessary for the functionality of CODE2VEC.
- All logging statements were removed from the model. These were found to clog up the buffer for the TCP communication with the server, making it non-functional.
- code2vec.py was altered to support its initialization and prediction by removing all options unrelated to those two aspects. It was also changed to be a client of server 8081 in order to connect with the ACP Server, which would loop to accept new functions to make predictions continuously.
- interactive_predictor.py initially made predictions from a file called Input.Java. This approach did not work, so the data sent from the ACP Server were passed as a parameter to its prediction function. This function was configured to work exclusively from the terminal, so those parts were removed to work with our data flow. Attention values are also unnecessary for the actual predictions, so all code involved was removed.
- extractor.py was meant for calling the Java Extractor from the Input.java file and then processing that output via subprocess. This would cause issues, so instead, the extact_paths function now passes the output of the Java Extraction, which was received all the way in CODE2VEC.

Java Extractor Alterations. Java Extractor was originally a jar file contained in the CODE2VEC model that would run through the terminal. This would be the file responsible for extracting the Java code into a usable form for the model. Leaving it as is would not be possible, so it was refactored as part of its integration into the plugin.

- All files not related to the actual code for Java Extractor were removed. While keeping them will not impact the plugin's functionality, their removal was mainly to keep the plugin's smaller in size.

- The main method in the App class was altered to return an ArrayList <ProgramFeatures>, the data type for the Java Extractor output. Furthermore, it was changed to be called execute, which is seen in the other parts of the plugin as *App.execute*.
- For the ExtractFeaturesTask, the processFile function was changed from void to the same type as above. This is where the app execution would get the output. It originally just printed out the output.
- For the ExtractFeaturesTask, extractSingleFile was changed. Now, it simply gets the code from the String passed into the code argument and executes the function from that.

In addition to CODE2VEC, we also integrate IntelliJ's built-in name recommender into our plugin to allow users to choose between them. For CODE2VEC to work in the plugin, the user must have installed at least Python 3.8, TensorFlow 2.x, and NumPy. In addition, Python path in a common directory. Otherwise, it will only use the built-in.

3.10 User Extraction Setting Improvements

User Extraction Selector. A user can select two different options to judge whether a fragment of code should be extracted. The first option is an AI model that has been trained on thousands of extractions from Apache projects [AlOmar et al., 2022; AlOmar et al., 2023a]. The model uses features from the pasted code such as the number of lines, keywords, keywords per line, and many more. If this setting is selected, users can choose how sensitive the model should be. A more sensitive model will be more lenient and will notify the user of more clones, while a less sensitive model will be stricter when judging the features of the code fragment. If the user chooses custom extraction heuristics, they can select certain features by which the code fragment should be judged. These features include the number of keywords the fragment should contain, how complex the fragment of code is, how large the fragment of code is, and how many references to outside classes the code fragment should contain (also known as coupling, a measure of dependence between two modules).

Clone-Type Selector. The user can select which clone types to detect, either Type-1 or Type-2. Since Type-1 clones are a subset of Type-2 clones, being that they are the exact duplicates with nothing changed, if Type-2 is selected, the user will be notified about both. If the user only wants to be notified about exact duplicates, and not about Type-2 clones with differing operations and variable names, they can just select Type-1.

Name Recommendation Model Selector. The user can choose naming recommendation models using IntelliJ's built-in name recommender or neural model CODE2VEC and indicate the maximum number of extracted method name predictions.

4 Evaluation and Discussion

This section describes our empirical study aimed at evaluating the proposed approach and its main results. We formulate six research questions (RQs):

- RQ1. To what extent is ANTICOPYPASTER able to correctly recommend the *Extract Method* refactoring?
- RQ2. How do the detection accuracy and efficiency of Type-1 and Type-2 code clones compare between ANTICOPYPASTER 1.0 and 2.0, and ANTICOPYPASTER 3.0?
- RQ3. How does the accuracy of method name recommendations in IntelliJ IDEA's built-in system compare to those generated by CODE2VEC?
- RQ4. How does the ANTICOPYPASTER perform in terms of scalability when applied to large-scale codebases?
- RQ5. How effectively do JDEODORANT and ANTICOPYPASTER support developers in refactoring the clone?

RQ6. What is the impact of JDEODORANT and ANTICOPYPASTER refactored code on quality?

In the following, we elaborate on the results for each research question.

4.1 RQ1. To What Extent Is ANTICOPYPASTER Able to Correctly Recommend the *Extract Method* Refactoring?

4.1.1 Motivation. We pose this research question to study the accuracy and effectiveness of ANTICOPYPASTER in identifying *Extract Method* refactoring instances. This sheds light on its practical utility and reliability in real-world software development scenarios.

4.1.2 Approach. To assess the accuracy of ANTICOPYPASTER's method extraction, we performed an external validation involving 13 student developers from the Stevens Institute of Technology. We employ convenience sampling, where participants are selected based on their availability and willingness to contribute. Participants volunteered to participate in the experiment, and informed consent was obtained. Regarding the development experience of the participants, all participants have more than 1 year of coding experience. In addition, the participants have between 1 and 4 years of experience in industry or open source.

We conducted a pilot study with two participants to refine the experiment's instrumentation and ensure that the instructions were clear and unambiguous. A pilot survey is a common approach to testing tools, including the questions and survey structure. Following the pilot study, we interactively refined the protocol and survey questions. The participants in the pilot study are undergraduate students in Software Engineering. Among others, we learned that there is an issue with Type-2(d) method extraction; the tool cannot refactor clones involving replacing parameters with expressions. Therefore, we fixed the issue by parameterizing the expression before running our experiment.

As suggested by Kitchenham and Pfleeger [2008], we constructed the survey to use a 5-point ordered response scale ("Likert scale") questions, 2 open-ended questions, and 3 multiple choice questions with an optional "Other" category, allowing the respondents to share thoughts not mentioned in the list. The survey consisted of 22 questions. The first part of the survey includes questions about the demographics of the participants. Next, we ask about the usefulness, usability, and functionality of the proposed idea and the plugin. We provide participants with a video that demonstrates how to use the plugin and a link to download it.

Regarding task execution, we ask participants to practice the plugin by copy-pasting various pieces of Java code of their choice, such as Type-1a, Type-1b, Type-1c, Type-2a, Type-2b, and Type-2d. Since there are two options for method extraction: (1) an AI model and (2) custom extraction heuristics, we asked the participants to use both methods. Similarly, we asked them to try both options for name recommendation: (1) an IntelliJ built-in and (2) a CODE2VEC model.

Furthermore, we asked the participants to upload the Java files, screenshots of the clustering and advanced metric settings, and potentially share the log of the plugin events with us. This log tracks the usage of the tool. This information provides us with more detailed information about the usage of the plugin over time. The log does not track any personal information related to the user or the source code. The log shows events such as the action of copying a code fragment, the action of pasting a code fragment, and the acceptance of the refactoring recommendation. At the end of the survey, participants need to upload an XML file, which the plugin auto-generates, to share the log with us.

4.1.3 Results. We started the survey by asking participants to perform method extraction using the AI model or custom extraction heuristics. The results show that 51% of the developers used the AI model, while the other 49% relied on custom manual heuristics. Furthermore, when asked

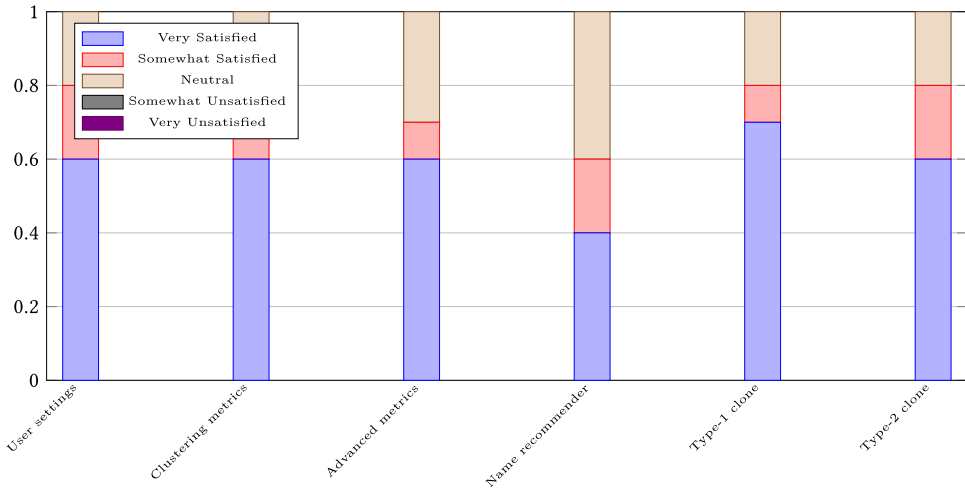


Fig. 4. Participants' satisfaction with various aspects of the ANTICOPYPASTER tool.

about the naming recommendations, 84% of the respondents used the built-in suggestions, and 16% opted for the model recommendation. Concerning the *Extract Method* refactoring notification, 81% indicated that the tool triggers the notification, while 19% reported that no notification was triggered. This can be explained by the fact that the copy-pasted fragments are not worth the extraction based on the AI model's decision.

All participants confirmed that they had executed the plugin. We asked them about their satisfaction with various aspects of the tool. Figure 4 presents an overview of their responses. Each bar represents various aspects of the tool, and the segments within each bar show the proportion of participants selecting a specific option. The total height of each bar is equal to 100%, representing the entire sample within that group. With respect to user-configurable settings, clustering metrics, advanced metric settings, and name recommendations, all respondents reported being satisfied with the tool. Some participants remained neutral. In terms of detecting Type-1 and Type-2 code clones, all respondents indicated that they were satisfied. There were a few participants who remained neutral. This might be explained by the fact that the model may not fully capture the semantic appropriateness or meaningfulness of the suggested names.

Regarding the frequency of the *Extract Method* pop-up notification for each type of code clone, Figure 5 illustrates that participants reported *always*, *usually*, and *sometimes* seeing the notification for Type-1a, Type-1b, and Type-1c clones. For Type-2 clones, participants indicated that they always, usually, and sometimes saw the notification for Type-2a, Type-2b, and Type-2d clones, respectively. A few participants reported *rarely* or *never* seeing the notification for these types of clones. Note that Type-2c is not included in this figure because we currently only support its detection. The extraction of Type-2c clones will be addressed in our future work.

Summary for RQ1. ANTICOPYPASTER correctly recommends *Extract Method* refactoring. In general, the participants expressed high levels of satisfaction with the plugin. They rated its various features and aspects very positively, highlighting its effectiveness and user-friendly design.

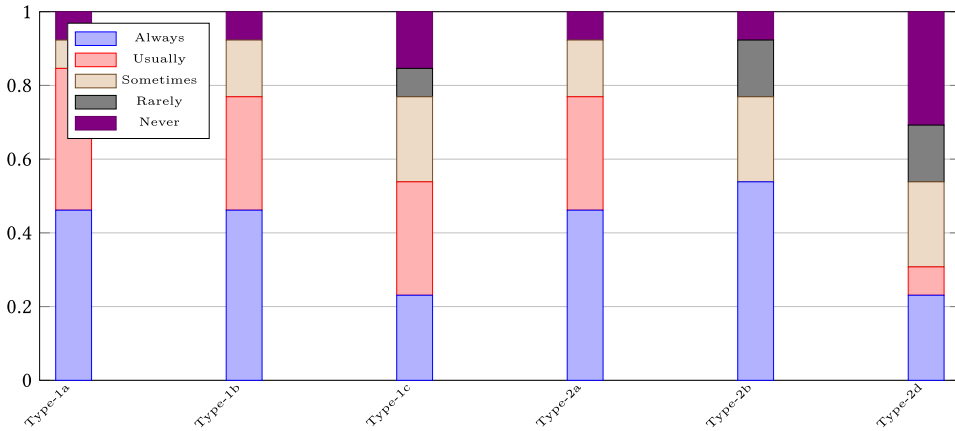


Fig. 5. Frequency of pop-up notifications seen by participants while using the ANTIcopyPASTER tool.

4.2 RQ2. How Do the Detection Accuracy and Efficiency of Type-1 and Type-2 Code Clones Compare between ANTIcopyPASTER 1.0 and 2.0, and ANTIcopyPASTER 3.0?

4.2.1 Motivation. Our second research question aims to compare the performance of the bag-of-words model and direct substring matching in ANTIcopyPASTER 1.0 and 2.0 with our newly designed detection strategy, using the PSI, in ANTIcopyPASTER 3.0. This comparison is vital to understanding the advancements in the effectiveness of code clone detection methods, highlighting the evolution and improvements in our plugin's capability.

4.2.2 Approach. To evaluate the accuracy of clone detection using the bag-of-words model and direct substring matching (used in ANTIcopyPASTER 1.0 and 2.0) and PSI (used in ANTIcopyPASTER 3.0), we follow these steps: (1) select 24 methods (6 subtypes of clones (4 examples each)) from different classes, (2) copy and paste the selected methods for each clone type (i.e., all subtypes of Type-1 and Type-2), and (3) report whether the plugin can detect code clone types. Since the aforementioned workflow is quite time-consuming, we decided to evaluate our approach using different Java methods on a project (i.e., JHotDraw), which is commonly used in *Extract Method*-related studies [Kaur, 2022; Silva et al., 2014; Tiwari and Joshi, 2022].

4.2.3 Results. Figure 6 illustrates the results of comparing clone detection techniques for the previous and extended versions of the plugin. We observe that using our newly introduced method accurately detects clones for both Type-1 and Type-2 clones. This method leverages the structural and semantic information of the code, enabling it to effectively identify even subtle variations in clones, such as renamed identifiers and modified literals, which are common in Type-2 clones. This observation is consistent with studies using AST to detect refactorings and assess the refactorability of software clones [Büch and Andrzejak, 2019; Tsantalis et al., 2015; Tsantalis et al., 2020; Wang et al., 2020].

However, bag-of-words and direct substring matching techniques show limitations in their reliability. Although they may detect some subtypes of clones, their effectiveness is highly dependent on the size of the code fragments and the ability of these models to recognize patterns. It should be noted that while ANTIcopyPASTER 1.0 is not designed to detect Type-2 clones, there are a few instances where it does. This is due to the 10-second timer, which is not user-configurable or controllable in version 1.0, making it challenging for users to make Type-2-related modifications before the notification appears. Bag-of-words, which treats code as a collection of tokens without

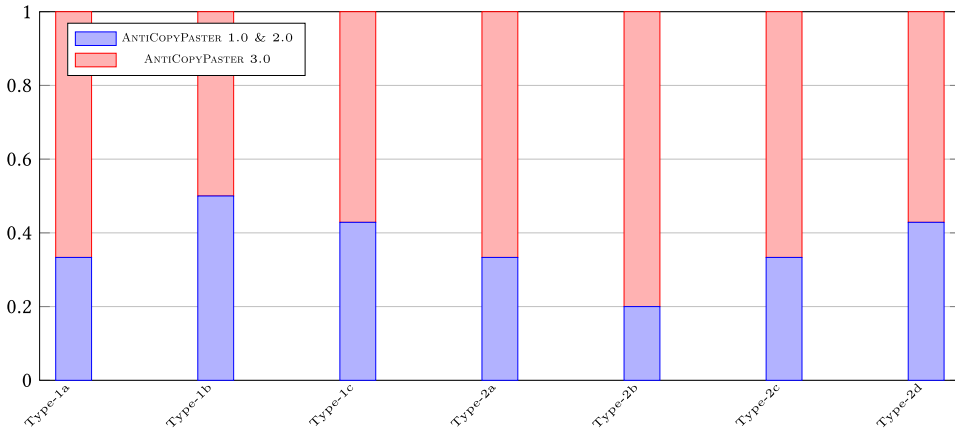


Fig. 6. Comparison between clone detection techniques.

considering their order or structure, can miss clones that involve significant structural changes. A recent study highlights the limitation of treating source code as bags-of-tokens, which renders the detection approach unable to handle changes in tokens caused by refactoring (e.g., parameterization of expressions in *Extract Method* refactoring) [Tsantalis et al., 2020]. Similarly, direct substring matching, which relies on exact or near-exact text matches, struggles with clones where even minor edits, like whitespace changes or comment additions, are present.

For example, consider a Type-2 clone in Figure 7, where the identifiers and literal names are changed. ANTIcopyPASTER can accurately detect this clone by analyzing the underlying AST and understanding the code’s structure and semantics. It detects the structural and semantic similarities (i.e., both methods have the same control flow (sequence of statements), and both methods perform similar operations (multiplication and subtraction)). PSI can map identifiers and recognize that “price” in `calculateDiscount` maps to “amount” in `computeReduction`, “discountRate” maps to “reductionRate,” “discountAmount” maps to “reductionValue,” and “finalPrice” maps to “totalAmount.”

In contrast, bag-of-words will capture the frequencies of tokens but will miss the context and relationships between them. When comparing token frequencies, it recognizes “double”: three occurrences (in both methods), “public”: one occurrence (in both methods), “return”: one occurrence (in both methods), “*”: one occurrence (in both methods), “-”: one occurrence (in both methods), and “=”: two occurrences (in both methods). Despite capturing these frequencies, bag-of-words fails to detect that “price” and “amount” are semantically equivalent and recognize that the logical structure of the operations (multiplication and subtraction) is preserved.

Similarly, direct substring matching might find common sequences such as “public double,” “double,” “*,” “-,” and “return.” However, it fails to identify that “price” corresponds to “amount,” identify that “discountRate” corresponds to “reductionRate,” and recognize the structural and logical equivalence between the two methods.

Summary for RQ2. Our detection method effectively identifies clones by analyzing the structural and semantic aspects of the code, making it reliable even when variables and literals are renamed. In contrast, bag-of-words and direct substring methods are less dependable, often missing clones due to their reliance on token patterns and exact text matches.

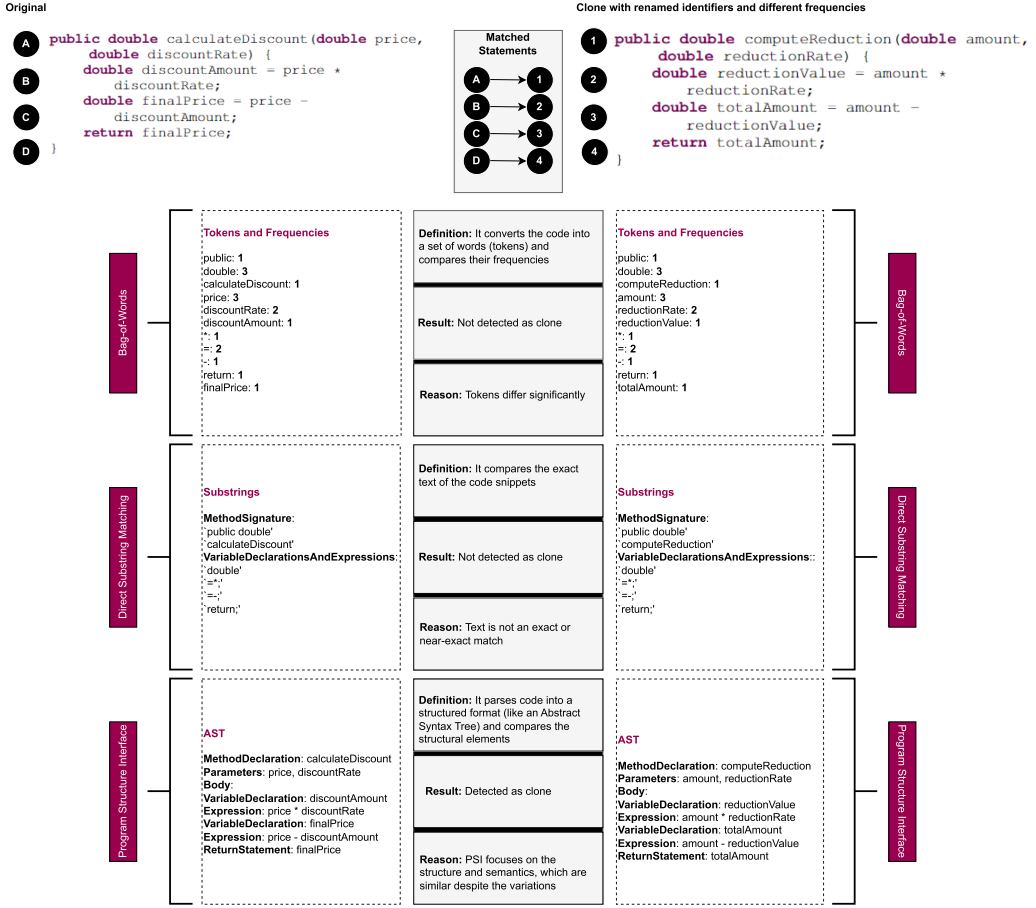


Fig. 7. Difference between code clone detection strategies (Type-2 example).

4.3 RQ3. How Does the Accuracy of Method Name Recommendations in IntelliJ IDEA's Built-in System Compare to Those Generated by CODE2VEC?

4.3.1 Motivation. Answering the third research question helps to evaluate the effectiveness of advanced machine learning-based recommendations provided by CODE2VEC against the traditional built-in recommender, providing insight into potential improvements in program comprehension.

4.3.2 Approach. To verify the accuracy between the recommendations of CODE2VEC and IntelliJ's built-in recommender, we evaluated the process as follows: (1) select 24 methods (6 subtypes of clones × 2 extraction settings × 2 name recommendation models) from different classes, (2) copy and paste the selected methods for each clone type (i.e., all subtypes of Type-1 and Type-2) using an AI model and custom extraction settings, (3) report the recommended names from both IntelliJ's built-in name recommendation and the CODE2VEC model, and (4) calculate the Jaccard similarity coefficient to determine how closely the recommended names match the original method names. Jaccard similarity coefficient is a measure of agreement between two sets and considers the overlap of words [Baeza-Yates et al., 1999] and is typically used for assessment in renaming studies [Nitin et al., 2021; Wang et al., 2017; Zhang et al., 2023]. Similarly to the previous research questions,

given the time-consuming nature of this workflow, we decided to evaluate our approach on a single project, JHotDraw.

Furthermore, to complement the Jaccard metric score with human judgment, we conducted a user study using the convenience sampling method with two student developers to assess whether the tool's recommendations align with developers' expectations. We evaluated the names generated by CODE2VEC as follows: (1) ask the participants to locate the 24 methods from the JHotDraw project, (2) understand the body of these methods, (3) check the names generated by CODE2VEC, and (4) report their feedback.

4.3.3 Results. Table 3 illustrates the recommendations of CODE2VEC for 24 methods. The Jaccard similarity measure indicates a low similarity match, highlighting discrepancies between both recommenders and the original method names. Due to space constraints, detailed values are included in our replication package. The difference between the two name recommenders can be explained by the fact that IntelliJ's built-in recommender consistently provides generic default names `extractedMethod` and `extractedmethod` regardless of the code context. These generic names lack specificity and do not convey the purpose or functionality of the methods, making them less useful to developers who rely on meaningful method names to understand and maintain code. In contrast, CODE2VEC suggests various recommendations that are more context-aware and capable of generating various names based on the code. For CODE2VEC, the prediction list is sorted from most to least confident, with the most confident suggestion at the top and the least confident suggestion at the bottom. This ranking indicates that the best recommendations are those CODE2VEC consider the most accurate based on its training data and algorithms. For example, consider the `DrawApplet` class and the `createAttributeChoices` method from Table 3. When processed by CODE2VEC, the following ranked names are suggested: `add_to2`, `addTo2`, `add_togetcombo_box`, `j_combo_box_action_performed`, `set_content_panel`, `parse_index`, and `update_gui`. These recommendations are ordered based on CODE2VEC's confidence, with `add_to2` being the most confident suggestion and `update_gui` being the least. This ranking helps developers quickly identify the most relevant and contextually appropriate method names.

It is worth noting that while Jaccard similarity provides a measure of the overlap of characters or words between the recommended and original names, it often yields low similarity scores for method name recommendations. This can be attributed to several factors. First, Jaccard similarity focuses on textual overlap without considering the semantic meaning of the words. As a result, even if two method names have similar functional intents but different word choices, the Jaccard similarity may not reflect this conceptual alignment. For example, a method named `addConstant` and another named `add` perform similar operations, but Jaccard similarity would score them low (i.e., 20%) due to different word usage. Furthermore, IntelliJ's built-in recommender provides generic names such as `extractedMethod`, which lack specific context and do not overlap well with more descriptive names generated by CODE2VEC, resulting in lower similarity scores. Therefore, other evaluation measures can be used to evaluate the quality of the name recommendations.

Figure 8 demonstrates the relevancy, clarity, accuracy, and quality of CODE2VEC's name suggestions as perceived by developers. For relevancy (i.e., how well the generated method names align with the functionality of the methods), 17.4% of the method names are rated highly relevant, 15.2% are rated very relevant, 23.9% are moderately relevant, another 26.1% are slightly relevant, but 17.4% are found to be irrelevant. As for clarity (i.e., how easily the generated names can be understood by developers), the majority of method names (47.8%) are rated moderately clear and other varied responses, with 17.4% extremely clear, 15.2% somewhat unclear, 13% very clear, but 6.5% are rated very unclear. Regarding accuracy (i.e., whether the generated names match the expected or ideal names for the methods), while 47.8% rated the names somewhat accurate, only

Table 3. Method Name Recommendations Generated by CODE2VEC

Class	Clone type	Extraction settings	Original method name	CODE2VEC method name (ranked from most to least confident)
DrawApplet	Type-1(a)	AI model	createAttributeChoices	add_to2/addTo2/add_to get_combo_box/j_combo_box_action_performed set_content_panel/parse_index/update_gui
FigureAttributeConstant	Type-1(a)	AI model	addConstant	set_constant/setConstant/set_default add/set_parent/set_variable set_permission/set/get_data_value set_checked/set_action
DrawApplet	Type-1(a)	Custom heuristics	init	init1/init_ui/start init/run/initialize create_components/set up/init_components create_ui/add_components
NothingApplet	Type-1(a)	Custom heuristics	createTools	set_images/setImages/create_images update_images/get_all_images/add_picture create_header_panel/get_cell_name/create_item make_actions/create_tool_bar
CompositeFigureCreationTool	Type-1(b)	AI model	mouseDown	create_figure/createFigure/get_connection_figure set_focus/case_item/is_type on_websocket_close/draw_line/get_background ret/retrieve_context
ChopDiamondConnector	Type-1(b)	AI model	chop	find_intersection/findIntersection/get_angle diff/angle_between/substract closest_point/drag/transfer set_selected/intersection
MiniMapDesktop	Type-1(b)	Custom heuristics	createLeftComponent	create_overlay/createOverlay/create_buffers create_attribute/resize/set_background remove_overlay/create_buffer/create_binary_frame set_red/set_radius
Animator	Type-1(b)	Custom heuristics	run	pulse/fetch_documents/read from collect_files/get_histogram/populate_data set_repositories/test_histogram/post_runnable internal_run
CTXWindowMenu	Type-1(c)	AI model	buildChildMenus	add_item/addItem/add_to sign_in/add_command/add_menu_item menu_selected/fire/update_item customize/on_notification
ColorMap	Type-1(c)	AI model	colorIndex	get_color/add_entry/compare get_color_value/update_value/put_color forward/find/get/get_index
BouncingDrawing	Type-1(c)	Custom heuristics	animationStep	on_location_changed/onLocationChanged/tick set_labels/remove_figure/remove_entity process_mouse_motion_event/update_position/update_fonts do_unregister/set_chart
PertDependency	Type-1(c)	Custom heuristics	handleConnect	adopt/visit/rotate/paint handle_result/create_relation/add_field notify_start/remove_figure/merge_attributes
AutoscrollHelper	Type-2(a)	AI model	getAutoscrollInsets	get_autoscroll_insets/getAutoscrollInsets/autoscroll represent_data/add_input_stream/prepare_queries reconciled/update tick/test_f
Geom	Type-2(a)	AI model	direction	transform_inverse/init_input scroll_to_visible/scrollToVisible/turn/move jump/get_vertex/contains/down find_dir/rot/is_bottom
JHDropTargetListener	Type-2(a)	Custom heuristics	dragOver	drop_action_changed/dropActionChanged/drag_enter drag_over/update_children/update_drag is_drag_ok/set_contact/is_equal text_changed/update_row
StandardLayouter	Type-2(a)	Custom heuristics	layout	provide_shape/provideShape/column_at_point get_popup_location/contains_point/get_initial_size is_dark/hit_test/is_over_selection can_dock/mouse_event
LineConnection	Type-2(b)	AI model	basicMoveBy	compare/translate/distance_sq consistent/distance_squared/mouse_dragged heuristic/add_entries/adjacent
TriangleFigure	Type-2(b)	AI model	rotate	compare/evalaute_int/consistent check_equal/check_match/check_contents rotate/rotate_x/assert_equals/get_direction
SplitPaneDesktop	Type-2(b)	Custom heuristics	removeFromDesktop	compare/consistent/get_value find_index/round/decode_value get_idx/change_title/get_enum_value/read_field
Helper	Type-2(b)	Custom heuristics	getDrawingView	compare/consistent/eval/round_up find_index/get_idx/add_entries distance_sq/update_map/add_constant
PolygonFigure	Type-2(d)	AI model	splitSegment	compare/translate/compare_attribute check_max/get_value_at/eval fmod/compare_nodes/dist
PatternPainter	Type-2(d)	AI model	drawPattern	paint_image/paintImage/scale_image image_update/compare/consistent check_image/paint/pack draw_centered/save_image

(Continued)

Table 3. Continued

Class	Clone type	Extraction settings	Original method name	CODE2VEC method name (ranked from most to least confident)
TriangleRotationHandle	Type-2(d)	Custom heuristics	getOrigin	update_view/updateView/is_syncing/do_repaint enable_debugging/update_name/data_changed/set_shape_type repaint_view/set_red/get_polygon
DiamondFigure	Type-2(d)	Custom heuristics	getPolygon	run/undo/down/compute extend/move_down/zoom_in/draw

19.6% found them accurate, indicating that relevance alone does not guarantee accuracy. In terms of the overall quality of the names, it had varied responses, with 37% somewhat dissatisfied, 21.7% very satisfied, 15.2% neutral, 13% somewhat satisfied, and 13% very dissatisfied. The relatively lower quality ratings suggest that the names may require more refinement to enhance readability, conciseness, or informativeness.

Figure 8 illustrates the developers’ perspectives on the extent to which CODE2VEC’s suggestions follow naming standards, and how well they meet developers’ expectations. The high ratings for adhering to the standard (89.1% rated somewhat or very well) suggest that the generated names generally follow naming conventions or practices. As for whether the recommendation aligns with expectations for meaningful and descriptive names, 26.1% of the methods met the expectations, 28.3% partially met them, and 45.7% did not meet the expectations.

About what considered to be the strengths of the CODE2VEC recommendations for the name of the refactored code, the participants indicate that the model can accurately identify the goal of the function even when the original function name was not very descriptive and works well with a simple function and general names. As for the suggested improvements for the CODE2VEC’s naming recommendations, the participants indicate the model needs to be improved by coming up with a good specific name and taking into account the comments around the function to understand the context better.

Summary for RQ3. Our analysis compared the method name recommendations generated by IntelliJ IDEA’s built-in default message and CODE2VEC. IntelliJ’s recommender often provides generic names like `extractedMethod`, lacking context-specific details, while CODE2VEC produces descriptive and relevant names based on the code context. This highlights the importance of integrating advanced name recommendation tools into development environments to improve developer productivity.

4.4 RQ4. How Does the ANTICOPYPASTER Perform in Terms of Scalability When Applied to Large-Scale Codebases?

4.4.1 Motivation. This research question aims to assess the performance of a tool in a large project and to determine whether it maintains efficiency when applied to significant codebases. We measured our tool’s performance by (1) scaling the size of the project, to challenge its computation on large codebases; and (2) scaling the number of clones, to challenge its ability to track and refactor many code blocks.

4.4.2 Approach. To evaluate the scalability of ANTICOPYPASTER, we select JFreeChart (version 1.0.10) as our experimental project. JFreeChart includes Java files of varying sizes, from a few hundred lines to several thousand, and Tsantalis et al. [2015, 2017] mentioned that the JFreeChart project contains a significant number of clone pairs. Hence, this project is well suited for testing how the tool performs across different codebase scales.

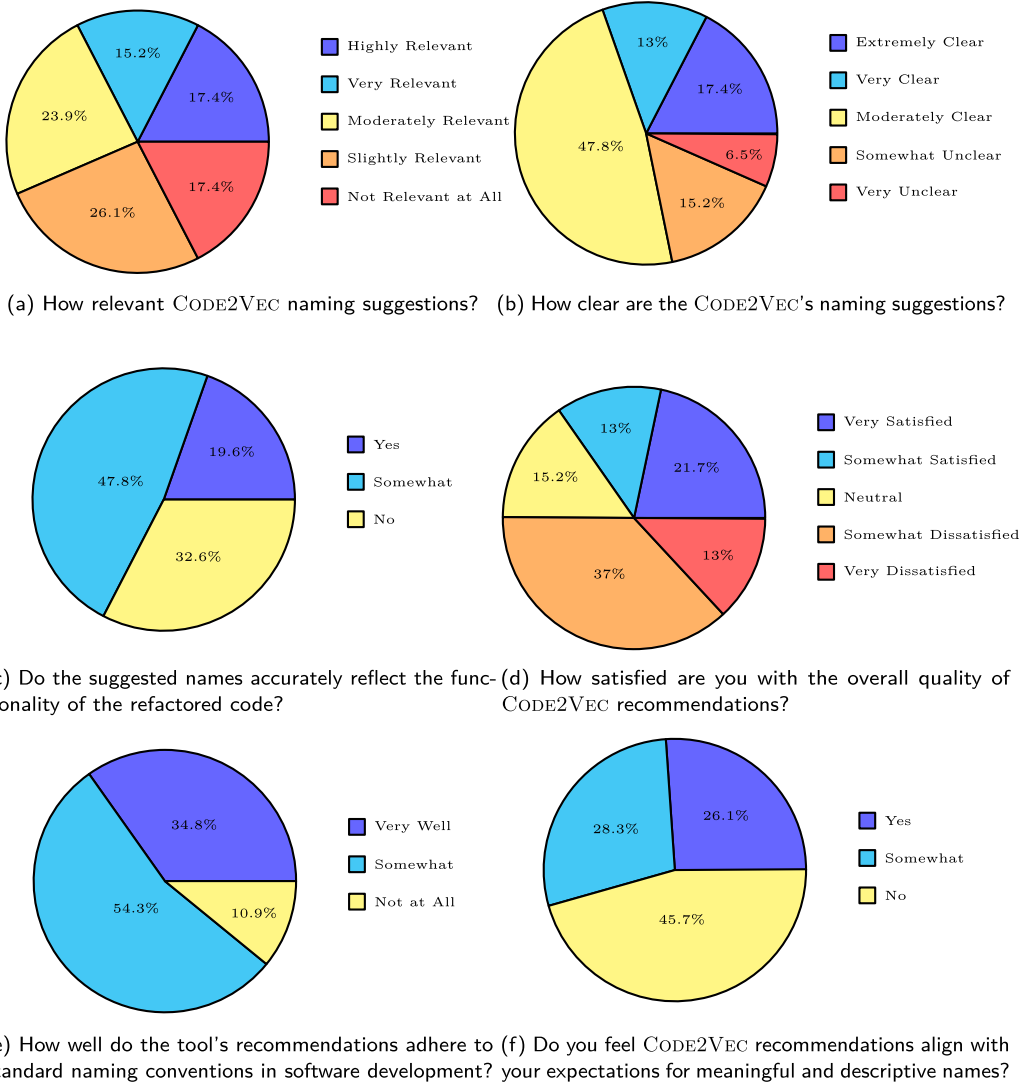
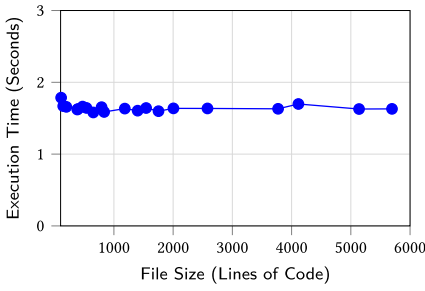


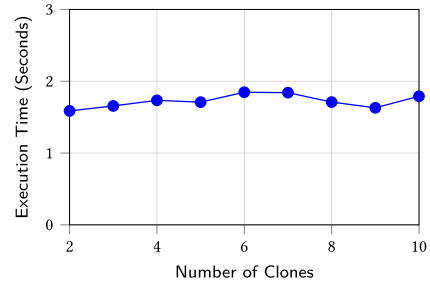
Fig. 8. Participants' attitude toward utilizing CODE2VEC for method name recommendation.

Since ANTICOPYPASTER shows notifications when users perform copy-paste actions, we utilized JAVA.QSL.TIMESTAMP library to record timestamps to measure the tool's response time. Specifically, we capture the time when the user pastes the code ("Paste time") and when the tool flags a notification ("Notify time"). The timestamps follow the following format: Year-Month-Day Hour:Minute:Second. Because the wait time never exceeds 1 minute, we calculate the delay using only the seconds field. Therefore, we consider two scenarios: (1) Does the size of file (number of codebase lines) affect the response time of ANTICOPYPASTER? and (2) Does the number of clones in the file affect the response time of ANTICOPYPASTER?

4.4.3 Results. To answer our research question, we performed clone refactoring experiments. Figure 9(a) shows the results for the first scenario (i.e., Does file size (number of codebase lines) affect



(a) Execution time of ANTICOPYPASTER across varying file sizes.



(b) Execution time of ANTICOPYPASTER across varying number of clones.

Fig. 9. Scalability evaluation of ANTICOPYPASTER with respect to (a) file size and (b) number of clones.

response time of ANTICOPYPASTER). The x -axis represents the number of lines in a file, while the y -axis shows the response time in seconds. As illustrated in the figure, even as the file size increases from 109 to 5,695 lines, the response time remains fairly consistent, which is around 1.6 seconds. The results suggest that the size of the codebase does not affect ANTICOPYPASTER response time, which primarily depends on the user-defined waiting time set in the tool's configuration.

Regarding the second scenario (i.e., Does the number of clones in the file affect the response time of ANTICOPYPASTER?), the x -axis of Figure 9(b) represents the number of clones in the file, and the y -axis shows the response time. This figure illustrates the relationship between the number of clones and the response time. As the number of clones increases, the response time remains relatively stable. Even when a file contains 10 clones, the response time is comparable to that of a file with only two clones. This suggests that the number of clones has a minimal impact on the response time of ANTICOPYPASTER.

Summary for RQ4. *The performance of ANTICOPYPASTER remains stable regardless of the file size and the number of clones present.*

4.5 RQ5. How Effectively Do JDEODORANT and ANTICOPYPASTER Support Developers in Refactoring the Clone?

4.5.1 Motivation. Our fifth research question aims to analyze the method extraction capabilities of JDEODORANT and ANTICOPYPASTER. We chose JDEODORANT [Tsantalis et al., 2015; Tsantalis et al., 2017], a popular publicly available state-of-the-art clone refactoring tool that supports the detection of Type-1 and Type-2 code clones in Java and is widely used in empirical studies.

4.5.2 Approach. The comparison between JDEODORANT and ANTICOPYPASTER, as two fully functioning plugins, is tricky, as JDEODORANT relies on third-party detectors to identify clones and assumes that developers are performing a refactoring session, in which they (1) run the detection, (2) choose the appropriate clones, and (3) fill out a template that serves as input to the refactoring engine. ANTICOPYPASTER's idea is to elevate such a process by recommending the extraction of the method as soon as the clone is introduced. Also, ANTICOPYPASTER can be tuned to detect any clone, and so it supports the refactoring of any clone a developer would choose to remove. Therefore, our comparison focuses on the correction mechanics, assuming that both plugins are given the same set of clones, with the goal of removing them.

Table 4. Refactoring Capabilities Performed by JDEODORANT and ANTICOPYPASTER

Class	Method	Clone type	Tools			Test cases passed?	
			DECKARD	JDEODORANT	ANTICOPYPASTER	JDEODORANT	ANTICOPYPASTER
DrawApplet	createAttributeChoices	Type-1(a)	Yes	Yes	Yes	Yes	Yes
FigureAttributeConstant	addConstant	Type-1(a)	Yes	Yes	Yes	Yes	Yes
DrawApplet	init	Type-1(a)	Yes	Yes	Yes	Yes	Yes
NothingApplet	createTools	Type-1(a)	Yes	No	Yes	Yes	Yes
CompositeFigureCreationTool	mouseDown	Type-1(b)	Yes	Yes	Yes	Yes	Yes
ChopDiamondConnector	chop	Type-1(b)	Yes	Yes	Yes	Yes	Yes
MiniMapDesktop	createLeftComponent	Type-1(b)	Yes	Yes	Yes	Yes	Yes
Animator	run	Type-1(b)	Yes	Yes	Yes	Yes	Yes
CTXWindowMenu	buildChildMenus	Type-1(c)	Yes	Yes	Yes	Yes	Yes
ColorMap	colorIndex	Type-1(c)	Yes	Yes	Yes	Yes	Yes
BouncingDrawing	animationStep	Type-1(c)	Yes	Yes	Yes	Yes	Yes
PertDependency	handleConnect	Type-1(c)	Yes	Yes	Yes	Yes	Yes
AutoscrollHelper	getAutoscrollInsets	Type-2(a)	Yes	Yes	Yes	Yes	Yes
Geom	direction	Type-2(a)	Yes	Yes	Yes	Yes	Yes
JHDropTargetListener	dragOver	Type-2(a)	Yes	Yes	Yes	Yes	Yes
StandardLayouter	layout	Type-2(a)	Yes	Yes	Yes	Yes	Yes
LineConnection	basicMoveBy	Type-2(b)	Yes	Yes	Yes	Yes	Yes
TriangleFigure	rotate	Type-2(b)	Yes	Yes	Yes	Yes	Yes
SplitPaneDesktop	removeFromDesktop	Type-2(b)	Yes	Yes	Yes	Yes	Yes
Helper	getDrawingView	Type-2(b)	Yes	Yes	Yes	Yes	Yes

For the benchmark, we selected refactorings from the dataset that has been used to validate the correctness of *Extract Method* refactoring [Charalampidou et al., 2016; Silva et al., 2014; Silva et al., 2015; Tiwari and Joshi, 2022; Xu et al., 2017a; Xu et al., 2017b]. Furthermore, we selected the refactorings of the clones detected by DECKARD [Jiang et al., 2007], a tree-based tool integrated with JDEODORANT that mostly reports complete AST statements as clones. This choice ensures JDEODORANT's optimal performance, as reported in the tool's paper [Tsantalis et al., 2017]. For the class methods selected for the analysis, we removed all the unsupported clones to ensure that we mimic the clones whose behavior fits both tools. Additionally, we decided to select Java methods from the JHotDraw project because it contains high testing coverage, allowing for testing the methods after the refactoring to verify the behavior preservation. It should be noted that some of the test cases in JHotDraw project do not cover the clone, so we use GPT-o4-mini-high to automatically generate test cases for the selected methods.

4.5.3 Results. Table 4 shows an analysis of the refactoring capabilities between ANTICOPYPASTER and JDEODORANT. We can observe that, despite the use of different detection and correction mechanisms, the tools were able to perform method extraction, except in a few cases that we discuss below:

For the detection of code clones, we have realized that DECKARD and ANTICOPYPASTER successfully detect most clones. According to a previous study [Roy et al., 2009], DECKARD does not support Type-2d detection, but is capable of detecting other subtypes of Type-1 and Type-2 clones. For the types that DECKARD cannot detect, it prevents us from running JDEODORANT on them. Upon our qualitative analysis, DECKARD struggles to detect Type-1c due to variations in the layout. Through repeated attempts, we found that the placement of changes within the layout impacts DECKARD's ability to work. Therefore, we exclude all cases that are considered inappropriate for our experiment.

Regarding the method extraction, both JDEODORANT and ANTICOPYPASTER were able to perform the extraction successfully while preserving the behavior of the code (i.e., safely extracted in a common method). This also implies compliance with pre- and post-conditions based on both tools despite the differences between these conditions in both tools.

Concerning Lambda expressions, given the different extraction mechanisms, ANTICOPYPASTER and JDEODORANT follow different approaches in the way they handle Lambda. For instance, for `rotate()` method in `TriangleFigure` class shown in Listing 7, it appears that the extraction

Table 5. Refactoring Capabilities Performed by AIDER

Class	Method	Clone type	Detection		Correction		Test cases passed?	
			GPT-3.5	GEMINI 2.5 PRO	GPT-3.5	GEMINI 2.5 PRO	GPT-3.5	GEMINI 2.5 PRO
DrawApplet	createAttributeChoices	Type-1(a)	No	Yes	N/A	N/A	N/A	N/A
FigureAttributeConstant	addConstant	Type-1(a)	No	Yes	N/A	Yes	N/A	Yes
DrawApplet	init	Type-1(a)	No	Yes	N/A	N/A	N/A	N/A
NothingApplet	createTools	Type-1(a)	No	Yes	N/A	N/A	N/A	N/A
CompositeFigureCreationTool	mouseDown	Type-1(b)	No	Yes	N/A	Yes	N/A	Yes
ChopDiamondConnector	chop	Type-1(b)	No	Yes	N/A	Yes	N/A	Yes
MiniMapDesktop	createLeftComponent	Type-1(b)	No	Yes	N/A	Yes	N/A	Yes
Animator	run	Type-1(b)	No	Yes	N/A	N/A	N/A	N/A
CTXWindowMenu	buildChildMenus	Type-1(c)	No	Yes	N/A	N/A	N/A	N/A
ColorMap	colorIndex	Type-1(c)	No	Yes	N/A	N/A	N/A	N/A
BouncingDrawing	animationStep	Type-1(c)	No	Yes	N/A	N/A	N/A	N/A
PertDependency	handleConnect	Type-1(c)	No	Yes	N/A	N/A	N/A	N/A
AutoscrollHelper	getAutoscrollInsets	Type-2(a)	No	Yes	N/A	N/A	N/A	N/A
Geom	direction	Type-2(a)	No	Yes	N/A	N/A	N/A	N/A
JHDropTargetListener	dragOver	Type-2(a)	No	Yes	N/A	N/A	N/A	N/A
StandardLayouter	layout	Type-2(a)	No	Yes	N/A	N/A	N/A	N/A
LineConnection	basicMoveBy	Type-2(b)	No	Yes	N/A	Yes	N/A	Yes
TriangleFigure	rotate	Type-2(b)	No	Yes	N/A	N/A	N/A	N/A
SplitPaneDesktop	removeFromDesktop	Type-2(b)	No	Yes	N/A	N/A	N/A	N/A
Helper	getDrawingView	Type-2(b)	No	Yes	N/A	N/A	N/A	N/A

“No” for detection indicates that the model is unable to detect, hence, it impacts the refactoring and test correctness labeled as “N/A.” “N/A” for correction indicates that there were either ineffective refactoring or incomplete refactoring, which also impacted the test correctness labeled as “N/A.”

of JDEODORANT in Listing 8 (i.e., `Function < Integer, Double > arg0, Function < Integer, Double > arg1`) is concise, while ANTICOPYPASTER in Listing 9 (i.e., `Java . util . function . BiFunction < Double , Double , Boolean > p1 , double angle , double [] rotations )`) prioritizes explicit logic.

Based on our qualitative analysis, several challenges arise when comparing these tools. The inherent differences in the operation of these tools make direct comparisons difficult. One major challenge is ensuring that the code fragments selected for the analysis do not trigger precondition violations in either tool. This is essential to accurately analyze their refactoring capabilities. Additionally, to create a comparable scenario, we need to carefully select pairs of methods as input and ensure that the tools run on a single file. Furthermore, we have to limit the comparison to no more than two clone instances. These constraints highlight the effort to create meaningful insights between tools with fundamentally different working mechanisms.

Furthermore, regarding the recent advances of refactoring support using LLMs, we utilized an AI agent named AIDER¹ to explore the potential of detecting and refactoring clones in addition to recommending names using the *Extract Method* refactoring. To do so, we begin by integrating AIDER into the workflow of ANTICOPYPASTER and constructing the prompts for detecting, extracting, and renaming the method. This approach is executed asynchronously using IntelliJ’s background thread infrastructure, ensuring that the analysis does not interrupt the user interface. Once AIDER completes its analysis, the tool output is scanned for indicators of potential code clones. If such clones are found, AIDER does not immediately apply any changes. Instead, it asks the user via a dialog box if they would like to proceed with the *Extract Method* refactoring operation. If the user agrees to proceed, AIDER enables a side-by-side comparison with the refactored result. If the user confirms the refactoring, the plugin replaces the original file content with the refactored code and recommends names for the extracted method. Otherwise, it leaves the file unchanged and informs the user that the refactoring was canceled.

After integrating the AI agent, we experimentally explore the ability of LLMs to detect and refactor clones using models from two common providers: OpenAI : GPT – 3.5 and Gemini : Gemini2.5pro, as well as propose names for the extracted method. We report the results in Table 5 and have made the following observations:

¹<https://github.com/Aider-AI/aider>.

- AIDER is generally capable of detecting and refactoring clones in small codebases. The quality and reliability of these tasks are highly dependent on the underlying model being used. For instance, we observe that GPT – 3.5 cannot detect clones, and therefore, it impacts the refactoring, whereas Gemini2.5pro successfully detects clones but struggles with refactoring. It is important to note the limitation when using AIDER for code clone detection and refactoring which relates to its heavy reliance on LLMs. LLMs have strict limits on the number of tokens they can process in a single request. When experimenting the tool with some files containing 5,000 LOC, Aider outputs a message “*Your estimated chat context of 39,071 tokens exceeds the 16,385 token limit for gpt-3.5-turbo.*” As a result, applying AIDER to large codebases might not be successful.
- AIDER can suggest names for the extracted methods. During the setup and integration with ANTICOPYPASTER, we observed that the performance can vary significantly between models. For instance, when using GPT – 3.5 to generate method names for an extracted method, the model does not always follow the instructions mentioned in the prompt as precisely as expected. In some cases, the model generates fewer names than requested by the user, even when the code snippet is sufficiently detailed. In other cases, it may ignore the specified format entirely, for example, including additional explanations or method bodies instead of method names, which were explicitly discouraged in the prompt. When the output does not conform to the expected structure, the parser fails to extract method names correctly, which can result in empty suggestions.
- As can be seen in Table 5, the code clone detection was successful for some types of clones (i.e., GPT – 3.5 cannot detect clones at all, but Gemini2.5Pro can). However, the refactoring performed by AIDER could be unsuccessful. According to the table, refactoring is successful in only five instances. The remaining cases were unsuccessful due to issues such as ineffective or incomplete refactoring, direct deletion of one of the clones without method extraction, or deletion of one of the clones after extraction rather than replacing it with a method call.

To sum up, while AIDER can provide some help in clone refactoring, it should not be relied on as the only refactoring tool. AIDER is perfect as a code companion, but not a replacement.

Summary for RQ5. *ANTICOPYPASTER encompasses several features, including several aspects such as just-in-time detection and correction, live refactoring, and advance clone refactoring for multi-instance groups. The absence of a systematic method for automatically comparing Extract Method plugins poses significant challenges in scaling the sample size. This difficulty arises due to various factors, including the detection and extraction mechanisms and restrictions, the nature of pre- and post-conditions, and the inherent workings of each tool. Furthermore, AIDER leverages LLMs to perform clone detection, refactoring, and renaming of extracted methods. However, the performance, quality, and reliability of these tasks are highly dependent on the underlying model being used.*

4.6 RQ6. What Is the Impact of JDEODORANT and ANTICOPYPASTER Refactored Code on Quality?

4.6.1 Motivation. Several studies attempted to assess the impact of refactoring on quality, allowing to collect, analyze, and get actionable data-driven insights about refactoring practices within software projects. In this study, our objective is to investigate the impact of *Extract Method* refactoring recommended by ANTICOPYPASTER and JDEODORANT on software quality.

```

public void rotate(double angle) {
    willChange();
    double dist = Double.MAX_VALUE;
    int best = 0;
    for (int i = 0; i < rotations.
        length; ++i) {
        double d = Math.abs(angle -
            rotations[i]);
        if (d < dist) {
            dist = d;
            best = i;
        }
    }
    fRotation = best;
    changed();
}

```

Listing 7. Using Lambda expressions to enable the refactoring of Type-2 clones (Before method extraction).

```

public void rotate_extracted(Function<
    Integer, Double> arg0, Function<
    Integer, Double> arg1) {
    willChange();
    double dist = Double.MAX_VALUE;
    int best = 0;
    for (int i = 0; i < rotations.
        length; ++i) {
        double d = Math.abs((double)
            arg0.apply(i) - (double)
            arg1.apply(i));
        if (d < dist) {
            dist = d;
            best = i;
        }
    }
    fRotation = best;
    changed();
    return;
}

public void rotate(double angle){
    rotate_extracted((Integer i) ->
        angle, (Integer i) ->
        rotations[i]);
}

public void rotate_Cloned(double angle
){
    rotate_extracted((Integer j) ->
        rotations[j], (Integer j) ->
        angle);
}

```

Listing 8. Using Lambda expressions to enable the refactoring of Type-2 clones (After method extraction by JDeodorant).

```

public void rotate_extracted(java.
    util.function.BiFunction<Double,
    Double, Boolean> p1, double angle
    , double[] rotations) {
    willChange();
    double dist = Double.MAX_VALUE;
    int best = 0;
    for (int i = 0; i < rotations.
        length; ++i) {
        double d = Math.abs(angle -
            rotations[i]);
        if (p1.apply(dist, d)) {
            dist = d;
            best = i;
        }
    }
    fRotation = best;
    changed();
}

public void rotate(double angle){
    rotate_extracted((dist, d) -> d <
        dist, angle, rotations);
}

public void rotate_Cloned(double angle
){
    rotate_extracted((test_d, d) -> d
        < test_d, angle, rotations);
}

```

Listing 9. Using Lambda expressions to enable the refactoring of Type-2 clones (After method extraction by AntiCopyPaster).

4.6.2 Approach. To collect a set of quality metrics, to be assessed in our study, we first survey the literature on existing and commonly acknowledged software quality attributes [Chidamber and Kemerer, 1994; Henry and Kafura, 1981; Lorenz and Kidd, 1994; McCabe, 1976]. Then, we checked if the metrics evaluate several object-oriented design aspects in order to map each internal quality attribute to the appropriate structural metric(s). More generally, we extract, from literature review, all

Table 6. Effect of Clone Refactoring on Structural Metrics

Quality attribute	Metric	JDEODORANT			ANTICOPYPASTER		
		Impact	p-value	Cliff's delta (δ)	Impact	p-value	Cliff's delta (δ)
Cohesion	LCOM	-ve	0.003	0.88 (Large)	-ve	0.01	0.87 (Large)
Coupling	CBO	-	no change observed	N/A	-ve	0.01	0.9 (Large)
	RFC	-ve	0.00006	0.92 (Large)	-ve	0.0001	0.91 (Large)
Complexity	CC	+ve	0.0004	0.89 (Large)	+ve	0.0004	0.89 (Large)
	WMC	+ve	0.0003	0.84 (Large)	+ve	0.0002	0.84 (Large)
Inheritance	DIT	-	no change observed	N/A	-	no change observed	N/A
	NOC	-	no change observed	N/A	-	no change observed	N/A
Size	LOC	+ve	0.0004	0.87 (Large)	+ve	0.0004	0.87 (Large)

(+ve) indicates positive impact; (-ve) indicates negative impact; (-) indicates metric remains unaffected; **bold** indicates statistical significance; *italic* indicates improvement.

the associations between metrics (e.g., CK suite [Chidamber and Kemerer, 1994], McCabe [McCabe, 1976], and Lorenz and Kidd's book [Lorenz and Kidd, 1994]) with internal quality attributes. The extraction process results in eight distinct structural metrics (i.e., LOC, **Cyclomatic Complexity (CC)**, **Weighted Method Count (WMC)**, **Response For Class (RFC)**, **Lack of Cohesion of Methods (LCOM)**, **Coupling between Objects (CBO)**, **Depth of Inheritance Tree (DIT)**, and **Number of Children (NOC)**). These metrics (1) are well known and defined in the literature, (2) can be evaluated on different code-level elements, and (3) can be calculated by existing static analysis tools. For this study, all metric values are automatically computed using the UNDERSTAND,² a popular static analysis framework. These data points will be used in the experiments to quantitatively measure the effect of changes in terms of structural metrics.

To assess the impact on quality, we consider the Java methods from JHotDraw used in RQ5. Then, we run the UNDERSTAND tool on these methods before (i.e., before eliminating clones) and after (i.e., after eliminating clones) the application of refactoring for both tools. As we calculate the metrics values of pre- and post-refactoring, we want to distinguish, for each metric, whether there is a variation on its pair of values, whether this variation indicates an improvement, and whether that variation is statistically significant. Therefore, we use the Wilcoxon test, a non-parametric test, to compare between the group of metric values before and after the refactoring, since these groups are dependent on one another. In each case, a decreased metric value is considered desirable (i.e., an improvement). Additionally, the variation between values of both sets is considered significant if its associated p-value is less than 0.05. Furthermore, we used the Cliff's delta (δ) effect size to estimate the magnitude of the differences. Regarding its interpretation, we follow the guidelines reported by Grissom and Kim [2005]: Negligible for $|\delta| < 0.147$, Small for $0.147 \leq |\delta| < 0.33$, Medium for $0.33 \leq |\delta| < 0.474$, and Large for $|\delta| \geq 0.474$.

4.6.3 Results. For each Java method from JHotDraw (Table 4), we compute its corresponding metric values before and after the refactoring. Table 6 shows the overall impact of *Extract Method* refactorings on quality. The boxplots in Figure 10 illustrate the distribution of each metric before and after refactoring for JDEODORANT and ANTICOPYPASTER. Next, we discuss the results for each metric.

LOC. The boxplot sketched in Figure 10(a) shows the pre- and post-refactoring results of LOC; we notice the improvement of the metric after the application of the refactoring. As can be seen in the boxplot, the mean decreased for JDEODORANT and ANTICOPYPASTER, from 126.26 to 115.53, and from 126.26 to 115.16, respectively. This result indicates that both tools improve the size after

²<https://scitools.com/>.

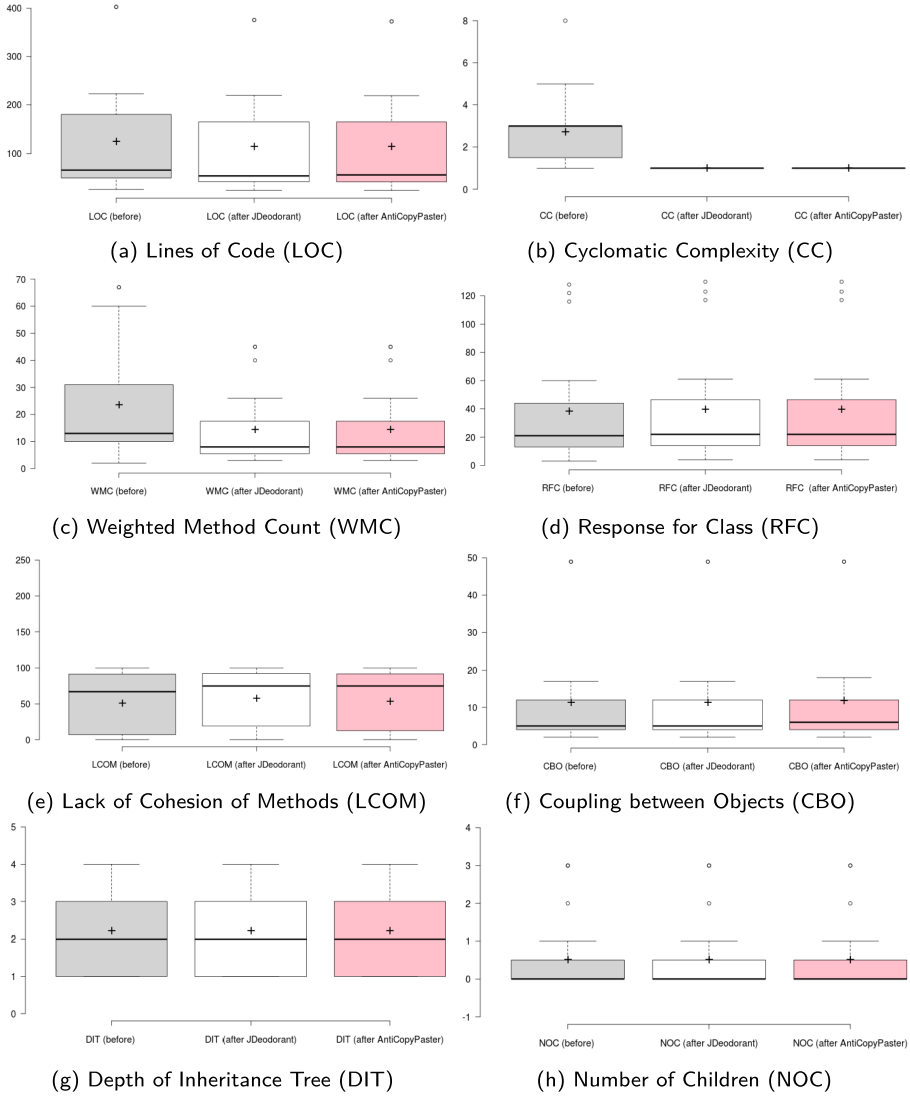


Fig. 10. Boxplots of metrics values of pre- and post-refactored files.

refactoring. Table 6 shows that the differences in LOC are statistically significant, and the magnitude of the differences is large.

CC. As for the complexity, we consider CC, shown in Figure 10(b), to investigate the code complexity reduction. As seen in the boxplot, we observe that the metric improves in the mean values from 2.74 to 1 for both JDEODORANT and ANTICOPYPASTER. This result indicates that both tools improve the complexity after refactoring. Table 6 shows that the differences in CC are statistically significant, and the magnitude of the differences is large.

WMC. The boxplot sketched in Figure 10(c) shows the pre- and post-refactoring results of WMC. We observe that this metric improves in the mean values from 23.68 to 14.58, and from 23.68 to 14.58, respectively, for JDEODORANT and ANTICOPYPASTER. This result indicates that both tools

improve the complexity after refactoring. Table 6 shows that the differences in WMC are statistically significant, and the magnitude of the differences is large.

RFC. We noticed a degradation of RFC metric, as shown in Figure 10(d) for JDEODORANT and ANTICOPYPASTER, respectively, from 38.42 to 39.74, and from 38.42 to 39.68. This result indicates that both tools worsen RFC after refactoring. Table 6 shows that the differences in RFC are statistically significant, and the magnitude of the differences is large.

LCOM. Figure 10(e) shows the results of the normalized LCOM, used in the literature to estimate the cohesion. A poor LCOM metric value implies generally that the classes should be split into 1 or more classes with better cohesion. Thus, if the value of this metric is low, it indicates a strong cohesiveness of the class. As can be seen from the boxplot in the figure, the mean increases from 51.37 to 58.21 and from 51.37 to 54.11 for JDEODORANT and ANTICOPYPASTER, respectively. This result indicates both tools worsen the cohesion after refactoring. Table 6 shows that the differences in LCOM are statistically significant, and the magnitude of the differences is large.

CBO. Figure 10(f) shows the results of the CBO, used in the literature to estimate the coupling. We observe from the figure that the coupling metric for JDEODORANT kept unchanged in the mean values. For ANTICOPYPASTER, we observed that the mean value slightly increased from 11.37 to 11.84. Upon the manual analysis, we observe that the slight increase in the mean can be attributed to the usage of Lambda expression, making CBO increases as it needs to call `java.util.function.Function`. Table 6 shows that the differences are tied, meaning that there was no measurable change on this metric for JDEODORANT. As for ANTICOPYPASTER, the differences in CBO are statistically significant and the magnitude of the differences is large.

DIT. The boxplot sketched in Figure 10(g) shows the pre- and post-refactoring results of DIT, used in the literature to estimate inheritance. The general depth of the inheritance tree was not affected for JDEODORANT and ANTICOPYPASTER. This result indicates that both tools do not affect inheritance after refactoring. Table 6 shows that the differences are tied, meaning that there was no measurable change on this metric.

NOC. Figure 10(h) illustrates the results of NOC, used in the literature to estimate inheritance. As can be seen, the overall number of immediate subclasses and superclasses was not affected for JDEODORANT and ANTICOPYPASTER. This result indicates that both tools do not affect inheritance after refactoring. Table 6 shows that the differences are tied, meaning that there was no measurable change on this metric.

Overall, we can see an improvement of quality after performing *Extract Method* refactoring by both tools. Our results are consistent with previous observations regarding size, complexity, and inheritance [Al Dallal and Abdin, 2017; Fernandes et al., 2020]. Previous studies have found that size and complexity typically improve after applying the *Extract Method* refactoring, while inheritance remains unaffected, and we observed similar trends. In contrast, our findings differed from earlier work in terms of cohesion and coupling. Specifically, cohesion worsened rather than improved following the refactoring for both tools, although the literature reports mixed effects (either improvement [Al Dallal and Abdin, 2017] or worsening [Fernandes et al., 2020]). Regarding coupling, the literature reports varied effects, some studies noted improvements [Fernandes et al., 2020], while others found no significant change [Al Dallal and Abdin, 2017], which helps explain the variability observed in our results for this metric.

Summary for RQ6. Overall, our results show an improvement in code quality after performing *Extract Method* refactoring with both tools, specifically for size and complexity. Inheritance remains unaffected, while cohesion and coupling do not show consistent improvement, reflecting the mixed effects previously reported in the literature.

5 Implication

The ANTICOPYPASTER tool has significant implications for the software development community, particularly in raising awareness of code clones and the importance of managing them through effective refactoring. By integrating these concepts, the tool not only highlights the prevalence of code clones but also provides practical strategies to address them. Here are the key research implications:

Fostering Collaboration among Research Communities. A recent keynote highlights significant code duplication within a large corpus of 2.6 million non-fork projects on GitHub, emphasizing the need for caution when using open source repositories for statistical analysis, especially with the advent of machine learning [IWS, 2022]. The ANTICOPYPASTER tool bridges the gap between different research communities, focusing on code cloning, refactoring, and program comprehension. The code clone community can provide insight into what constitutes good and bad clones, helping to refine detection and categorization methods. The refactoring community can suggest strategies for managing various types of code clones, providing guidelines and tools to improve the code structure and reduce redundancy. The program comprehension community can enhance the tool's naming recommendation capabilities by developing models or heuristics to generate high-quality method names, thereby improving the overall comprehensibility of the code.

Promoting Better Naming Conventions. The tool helps developers understand the importance of meaningful method names. Leveraging name recommendation models, such as those from the program comprehension community, generates intuitive and descriptive names for extracted methods. This improves the readability of the code.

Integrating Clone Refactoring Tool into CI/CD Pipelines. The clone refactoring toolset should be treated in the same way as CI/CD toolset and integrated into the tool-chain. Careful design of this integration is needed to ensure that it complements existing processes without disrupting developer productivity. For example, following the Jenkins pipeline, the clone refactoring tool could analyze pull requests for code clone patterns and suggest refactoring before merging. Additionally, this integration could be enhanced by displaying a dashboard that includes metrics such as code clone density, refactoring completion rates, and code complexity over time.

6 Limitations

In this section, we outline the known limitations of our tool and describe the features we plan to develop.

- *Only one variable can be modified by an extracted method.* Java does not have lightweight tuples and supports only returning one variable. As everything in Java is pass-by-value, we cannot update variables outside the scope of the extracted method except by returning them. Since we can only return one, if more than one variable is modified by a code fragment, it no longer becomes eligible for extraction. The example in Figure 11 depicts that our tool can extract a method for modifying either `sum` or `prod`, but not both, because it cannot return multiple values. Since Java methods can only return one value, the tool cannot handle refactoring both `sum` and `prod` into a single extracted method.
- *Lambda parameters can only have two inputs.* Lambda parameters can have a maximum of two inputs due to the limitations of the built-in Java-type system. To allow Lambdas with more arguments, we would have to generate interface types on the fly. This would further bloat user code and, therefore, was ruled against, as it is counterproductive to the goal of refactoring. The example in Figure 11 shows that our tool can handle two parameters (`sum` and `prod`), but cannot handle more than two parameters in a Lambda, such as include `i` as a third input.

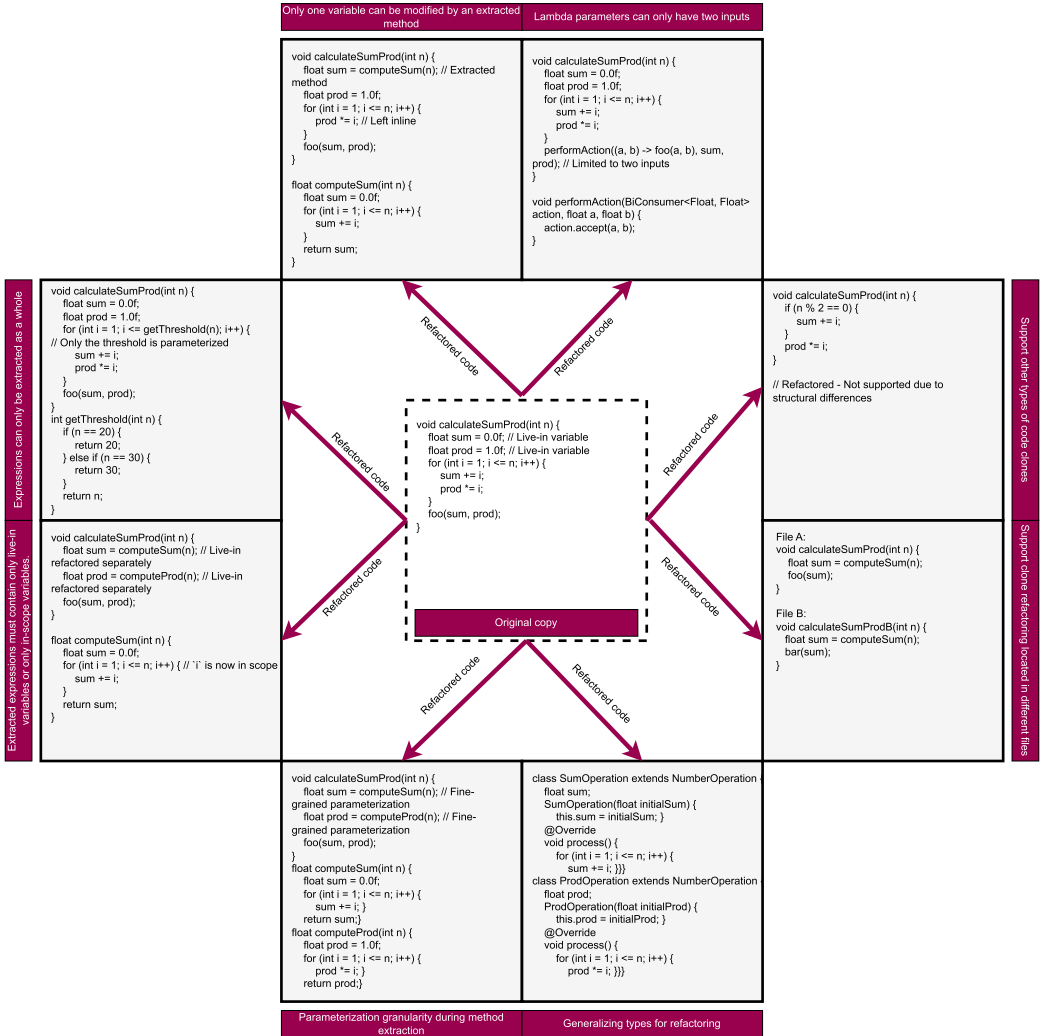


Fig. 11. Current tool limitation.

- *Expressions can only be extracted as a whole.* If an expression is similar in two fragments except for a single part (e.g., “ $i < 20$ ” vs. “ $i < 30$ ”, where only the number differs), we extract the entire expression as a parameter rather than focusing on the specific difference. This is because expression minimization would require what we believe to be a complex tree-ascent algorithm. This limitation can be overcome in a future version of the plugin. The example in Figure 11 demonstrated this limitation. If we assume the original code is copy-pasted with slight variations for “ $n = 20$ ” and “ $n = 30$ ” in two different fragments, our tool only extracts the entire loop condition ($i \leq 20$ or $i \leq 30$) as a parameter instead of focusing only on the number difference.
- *Extracted expressions must contain only live-in variables or only in-scope variables.* Extracted expressions must contain either exclusively live-in variables or exclusively in-scope variables. This is because we extract live-in expressions as the expression itself, but expressions with

in-scope variables are extracted as Lambdas. In the future, we plan to improve it to allow mixed-bags of variables (live-in and in-scope) by forcing parameter live-in dependencies to always be present in the parameter list. Then, the live-in variables can be passed as Lambda arguments, circumventing the Lambda limitations that permit only final variables to be used from outside contexts. The example in Figure 11 illustrates that our tool cannot refactor the loop because it mixes live-in variables (`sum` and `prod`) with in-scope variables (`i`).

- *Parameterization granularity during method extraction.* The current parameterization granularity is the largest (highest in the AST) non-function call expression with differences inside it that require parameterization. This implementation focuses on fine-grained parameterization, where each argument is treated independently to provide flexibility in handling diverse clones. However, this method does not allow users to select a coarser granularity (e.g., parameterizing the entire method call). To ensure better usability and code elegance, future improvements could include an improved algorithm that finds the smallest possible extractable expression. This would require a bottom-up smallest differing sub-tree scan, allowing them to control the granularity of parameterization by selecting between fine-grained and coarse-grained parameterization. The example in Figure 11 shows that our tool supports fine-grained parameterization but cannot extract the entire loop as a coarse-grained method.
- *Generalizing types for refactoring.* The tool currently has limitations in handling clones that could potentially be generalized to a common supertype. The applicability of refactoring for this type of clone is also not implemented, as extraction for this type is not carried out. In other words, our tool does not refactor and introduce a subtype (parent class) for the offspring. Our refactoring only refactors for a given type without looking up any hierarchies. At the current time, the detection of clones of Type-2c is the only ability this plugin has and that ability is internal only. It is implemented as an extension point to give us the necessary resources and a starting point to develop the necessary extraction logic. The example in Figure 11 demonstrates that the tool cannot refactor by introducing a parent class or common supertype for related types (offspring). It only refactors for the exact types used in the code and does not consider hierarchical relationships or abstraction opportunities.
- *Support clone refactoring located in different files.* At this time, the tool supports clone extraction within a single file. This design choice aligns with previous studies that found “*clones with a close relative location (i.e., same method, type, or file) tend to be more refactorable than clones in distant locations,*” and clones that are removed by refactoring are mostly located in the same file [Göde, 2010; Tsantalis et al., 2015]. Additionally, it was driven by our emphasis on minimizing runtime overhead, prioritizing immediate developer feedback, and streamlining the process of just-in-time refactoring by offering a fast, safe, and localized solution. In the future, we plan to support clones located in different files as well. The example in Figure 11 depicts that the tool cannot handle clones that reside in different files.
- *Support other types of code clones.* Currently, the detection is fully implemented for Type-2c clones. This includes tracking all metadata necessary for extraction. We also have a parameter minimization algorithm implemented that removes any detected types that are the same within all clones so that unnecessary extractions are not made. However, extraction for this clone type is not yet present. This is because generic methods are very hard to generate dynamically, yet are necessary to make this extraction type possible. All operations must be generic to work with generic types. Some of these operations include: the arithmetic operators must be replaced with generic functions; any functions called from within the clone must be changed to take generic types as well; and the generated method call might need additional code generated to handle packing and unpacking generics back into the regular code flow. Currently, all Type-2c clones are marked as nonextractable in the `TypeTwoCP` class as a record

field of `MatchState`; they are left in the detection list for convenience. In addition, we plan to detect and extract Type-3 and Type-4 clones and integrate other name recommendation models into our plugin. The example in Figure 11 illustrates that the tool does not support refactoring clones with structural or semantic differences.

7 Conclusion

This work presents a newer version of our code clone extraction. ANTICOPYPASTER 3.0 has some unique features the detection of clones of its previous versions: (1) it supports the detection of clones of both Type-1 and Type-2, along with a custom-built *Extract Method* refactoring, (2) it does not rely on direct substring matching and bag-of-words, (3) it provides a recommended name for the extracted method using IntelliJ's built-in name recommender or neural model CODE2VEC, and it allows the user to choose between AI model-based and custom extraction heuristics to determine if a code fragment should be extracted. To evaluate the accuracy of our tool, we refactored the 48 methods and all refactorings were carried out successfully. Furthermore, we assessed the usefulness of our tool with 13 student developers, and participants positively rated various aspects of the plugin. Moreover, our results show that PSI effectively identifies clones by analyzing the structural and semantic aspects of the code, and CODE2VEC produces descriptive names based on the context of the code. We foresee several promising avenues for future work, including the implementation of other types of code clones, the integration of additional name recommendation models, and the support for other programming languages.

Acknowledgments

The authors thank Dr. Nikolaos Tsantalis for his responsiveness and constructive feedback when performing the comparative analysis between JDEODORANT and ANTICOPYPASTER. We also thank Cavin Gada and Mark Falletta for exploring alternative name recommendation models and for assisting with tool testing.

Declaration of Generative AI and AI-Assisted Technologies in the Writing Process. The authors used the ChatGPT web interface to improve the readability in some parts of the manuscript. After using this tool, we reviewed and edited the content as needed, and took full responsibility for the publication's content. GPT was used to assist in the drafting of the textual description of the algorithm presented in this article, then refined by the authors; the algorithm itself was developed and designed independently by the authors.

References

- IWSC. 2022. Keynote. Retrieved from <https://iwsc2022.github.io/pages/keynote.html>
- Jehad Al Dallal and Anas Abdin. 2017. Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Transactions on Software Engineering* 44, 1 (2017), 44–69.
- Juan Pablo Sandoval Alcocer, Alejandra Siles Antezana, Gustavo Santos, and Alexandre Bergel. 2020. Improving the success rate of applying the extract method refactoring. *Science of Computer Programming* 195 (2020), 102475.
- Eman Abdullah AlOmar. 2025. An empirical study on the impact of code duplication-aware refactoring practices on quality metrics. *Information and Software Technology* 182 (2025), 1–21.
- Eman Abdullah AlOmar, Jacob Ashkenas, Robert Feliciano, Matthew Angelakos, Dimitrios Haralampopoulos, Xing Qian, Mohamed Wiem Mkaouer, and Ali Ouni. 2025. ReplicationPackage. Retrieved from <https://se4airesearch.github.io/AntiCopyPaster-Website/>
- Eman Abdullah AlOmar, Anton Ivanov, Zarina Kurbatova, Yaroslav Golubev, Mohamed Wiem Mkaouer, Ali Ouni, Timofey Bryksin, Le Nguyen, Amit Kini, and Aditya Thakur. 2022. AntiCopyPaster: Extracting code duplicates as soon as they are introduced in the IDE. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 1–4.
- Eman Abdullah AlOmar, Anton Ivanov, Zarina Kurbatova, Yaroslav Golubev, Mohamed Wiem Mkaouer, Ali Ouni, Timofey Bryksin, Le Nguyen, Amit Kini, and Aditya Thakur. 2023a. Just-in-time code duplicates extraction. *Information and Software Technology* 158 (2023), 107169.

- Eman Abdullah AlOmar, Benjamin Knobloch, Thomas Kain, Christopher Kalish, Mohamed Wiem Mkaouer, and Ali Ouni. 2024. AntiCopyPaster 2.0: Whitebox just-in-time code duplicates extraction. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, 84–88.
- Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. 2023b. Behind the intent of extract method refactoring: A systematic literature review. *IEEE Transactions on Software Engineering* 50 (2023), 668–694.
- 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.
- Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. arXiv:1808.01400. Retrieved from <https://arxiv.org/abs/1808.01400>
- Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.
- Mauricio Aniche, Erick Maziero, Rafael Durelli, and Vinicius H. S. Durelli. 2020. The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Transactions on Software Engineering* 48, 4 (2020), 1432–1450.
- Ricardo Baeza-Yates and Berthier Ribeiro-Neto. 1999. *Modern Information Retrieval*. Vol. 463. ACM Press New York.
- Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. 2014. Automating extract class refactoring: An improved method and its evaluation. *Empirical Software Engineering* 19, 6 (2014), 1617–1664.
- Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering* 33, 9 (2007), 577–591.
- Lutz Büch and Artur Andrzejak. 2019. Learning-based recursive aggregation of abstract syntax trees for code clone detection. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 95–104.
- Sofia Charalampidou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Antonios Gkortzis, and Paris Avgeriou. 2016. Identifying extract method refactoring opportunities based on functional relevance. *IEEE Transactions on Software Engineering* 43, 10 (2016), 954–974.
- Shyam R. Chidamber and Chris F. Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 20, 6 (1994), 476–493.
- Eunjong Choi, Norihiro Yoshida, Takashi Ishio, Katsuro Inoue, and Tateki Sano. 2011. Extracting code clones for refactoring using combinations of clone metrics. In *Proceedings of the 5th International Workshop on Software Clones*. ACM, 7–13.
- Osama Ehsan, Foutse Khomh, Ying Zou, and Dong Qiu. 2023. Ranking code clones to support maintenance activities. *Empirical Software Engineering* 28, 3 (2023), 70.
- Eduardo Fernandes, Alexander Chávez, Alessandro Garcia, Isabella Ferreira, Diego Cedrim, Leonardo Sousa, and Willian Oizumi. 2020. Refactoring effect on internal quality attributes: What haven't they told you yet? *Information and Software Technology* 126 (2020), 106347.
- Martin Fowler. 2018. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.
- Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable detection of semantic clones. In *Proceedings of the 30th International Conference on Software Engineering*, 321–330.
- Nils Göde. 2010. Clone removal: Fact or fiction? In *Proceedings of the 4th International Workshop on Software Clones*, 33–40.
- Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. 2021. One thousand and one stories: A large-scale survey of software refactoring. In *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1303–1313.
- Robert J. Grissom and John J. Kim. 2005. *Effect Sizes for Research: A Broad Practical Approach*. Mahwah, N.J.; London: Lawrence Erlbaum Associates. Formerly CIP.
- Roman Haas and Benjamin Hummel. 2016. Deriving extract method refactoring suggestions for long methods. In *International Conference on Software Quality*. Springer, 144–155.
- Sallie Henry and Dennis Kafura. 1981. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering* 5 (1981), 510–518.
- Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2004. Refactoring support based on code clone analysis. In *Proceedings of the 5th International Conference on Product Focused Software Process Improvement (PROFES '04)*. Springer, 220–233.
- Yoshiki Higo, Shinji Kusumoto, and Katsuro Inoue. 2008. A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. *Journal of Software Maintenance and Evolution: Research and Practice* 20, 6 (2008), 435–461.
- Keisuke Hotta, Yukiko Sano, Yoshiki Higo, and Shinji Kusumoto. 2010. Is duplicate code more frequently modified than non-duplicate code in software evolution? An empirical study on open source software. In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, 73–82.
- Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 96–105.

- Tomoko Kanemitsu, Yoshiaki Higo, and Shinji Kusumoto. 2011. A visualization method of program dependency graph for identifying extract method opportunity. In *Proceedings of the 4th Workshop on Refactoring Tools*, 8–14.
- Harpreet Kaur. 2022. Basic methodology to detect code clones. *NeuroQuantology* 20, 6 (2022), 101405.
- Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2012. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering* 17, 3 (2012), 243–275.
- Barbara A. Kitchenham and Shari L. Pfleeger. 2008. Personal opinion surveys. In *Guide to Advanced Empirical Software Engineering*. Springer, 63–92.
- Raghavan Komondoor and Susan Horwitz. 2001. Using slicing to identify duplication in source code. In *International Static Analysis Symposium*. Springer, 40–56.
- Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2006. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering* 32, 3 (2006), 176–192.
- Yi Li, Shaohua Wang, and Tien Nguyen. 2021. A context-based automated approach for method name consistency checking and suggestion. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 574–586.
- Yun Lin, Zhenchang Xing, Yinxing Xue, Yang Liu, Xin Peng, Jun Sun, and Wenyun Zhao. 2014. Detecting differences across multiple instances of code clones. In *Proceedings of the 36th International Conference on Software Engineering*, 164–174.
- Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajjani, and Vitek Jan. 2017. DéjàVu: A map of code duplicates on GitHub. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–28.
- Mark Lorenz and Jeff Kidd. 1994. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, Inc.
- Katsuhisa Maruyama. 2001. Automated method-extraction refactoring by using block-based slicing. In *Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context*, 31–40.
- Thomas J. McCabe. 1976. A complexity measure. *IEEE Transactions on Software Engineering* 4 (1976), 308–320.
- Na Meng, Lisa Hua, Miryung Kim, and Kathryn S. McKinley. 2015. Does automated refactoring obviate systematic editing?. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, 392–402.
- Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheue, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni. 2015. Many-objective software remodularization using NSGA-III. *ACM Transactions on Software Engineering and Methodology* 24, 3 (2015), 1–45.
- Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. 2014. Automatic identification of important clones for refactoring and tracking. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 11–20.
- Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. 2020. A survey on clone refactoring and tracking. *Journal of Systems and Software* 159 (2020), 110429.
- Rodrigo Morales, Zéphyrin Soh, Foutse Khomh, Giuliano Antoniol, and Francisco Chicano. 2017. On the use of developers' context for automatic refactoring of software anti-patterns. *Journal of Systems and Software* 128 (2017), 236–251.
- Emerson Murphy-Hill and Andrew P. Black. 2008. Breaking the barriers to successful refactoring: Observations and tools for extract method. In *Proceedings of the 30th International Conference on Software Engineering*, 421–430.
- Son Nguyen, Hung Phan, Trinh Le, and Tien N. Nguyen. 2020. Suggesting natural method names to check name consistencies. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 1372–1384.
- Vikram Nitin, Anthony Saieva, Baishakhi Ray, and Gail Kaiser. 2021. Direct: A transformer-based model for decompiled identifier renaming. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, 48–57.
- Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Kalyanmoy Deb. 2016. Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Transactions on Software Engineering and Methodology* 25, 3 (2016), 1–53.
- Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation. *Empirical Software Engineering* 23, 3 (2018), 1188–1221.
- Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. 2014. Do they really smell bad? A study on developers' perception of bad code smells. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 101–110.
- Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Timofey Bryksin, and Danny Dig. 2024a. Together we go further: LLMs and IDE static analysis for extract method refactoring. arXiv:2401.15298. Retrieved from <https://arxiv.org/abs/2401.15298>
- Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Andrey Sokolov, Timofey Bryksin, and Danny Dig. 2024b. EM-Assist: Safe automated ExtractMethod refactoring with LLMs. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 582–586.
- Chanchal Kumar Roy and James R. Cordy. 2007. A survey on software clone detection research. *Queen's School of Computing TR* 541, 115 (2007), 64–68.

- Chanchal K. Roy and James R. Cordy. 2008. An empirical study of function clones in open source software. In *2008 15th Working Conference on Reverse Engineering*. IEEE, 81–90.
- Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming* 74, 7 (2009), 470–495.
- Mahnoosh Shahidi, Mehrdad Ashtiani, and Morteza Zakeri-Nasrabadi. 2022. An automated extract method refactoring approach to correct the long method code smell. *Journal of Systems and Software* 187 (2022), 111221.
- Tushar Sharma. 2012. Identifying extract-method refactoring candidates automatically. In *Proceedings of the 5th Workshop on Refactoring Tools*, 50–53.
- Danilo Silva, Ricardo Terra, and Marco Tulio Valente. 2014. Recommending automated extract method refactorings. In *Proceedings of the 22nd International Conference on Program Comprehension*, 146–156.
- Danilo Silva, Ricardo Terra, and Marco Túlio Valente. 2015. JExtract: An eclipse plug-in for recommending automated Extract Method refactorings. arXiv:1506.06086. Retrieved from <https://arxiv.org/abs/1506.06086>
- Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? Confessions of GitHub contributors. In *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16)*. ACM, 858–870.
- Robert Tairas and Jeff Gray. 2009. Get to know your clones with CeDAR. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, 817–818.
- Robert Tairas and Jeff Gray. 2012. Increasing clone maintenance support by unifying clone detection and refactoring activities. *Information and Software Technology* 54, 12 (2012), 1297–1307.
- Steven L. Tanimoto. 2013. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*. IEEE, 31–34.
- Omkarendra Tiwari and Rushikesh Joshi. 2022. Identifying extract method rrefactorings. In *15th Innovations in Software Engineering Conference*, 1–11.
- Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2009. Identification of extract method refactoring opportunities. In *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, 119–128.
- Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2011. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software* 84, 10 (2011), 1757–1782.
- Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2020. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* 48, 3 (2020), 930–950.
- Nikolaos Tsantalis, Davood Mazinanian, and Giri Panamoottil Krishnan. 2015. Assessing the refactorability of software clones. *IEEE Transactions on Software Engineering* 41, 11 (2015), 1055–1090.
- Nikolaos Tsantalis, Davood Mazinanian, and Shahriar Rostami. 2017. Clone refactoring with lambda expressions. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 60–70.
- David van der Leij, Jasper Binda, Robbert van Dalen, Pieter Vallen, Yaping Luo, and Maurício Aniche. 2021. Data-driven extract method recommendations: A study at ING. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1337–1347.
- Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 261–271.
- Shangwen Wang, Ming Wen, Bo Lin, and Xiaoguang Mao. 2021. Lightweight global and local contexts guided method name recommendation with prior knowledge. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 741–753.
- Ying Wang, Hai Yu, Zhiliang Zhu, Wei Zhang, and Yuli Zhao. 2017. Automatic software refactoring via weighted clustering in method-level networks. *IEEE Transactions on Software Engineering* 44, 3 (2017), 202–236.
- Yueming Wu, Siyue Feng, Deqing Zou, and Hai Jin. 2022. Detecting semantic code clones by building AST-based Markov chains model. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 1–13.
- Zhenchang Xing, Yinxing Xue, and Stan Jarzabek. 2011. Clonedifferentiator: Analyzing clones by differentiation. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*. IEEE, 576–579.
- Sihan Xu, Chenkai Guo, Lei Liu, and Jing Xu. 2017a. A log-linear probabilistic model for prioritizing extract method refactorings. In *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*. IEEE, 2503–2507.
- Sihan Xu, Aishwarya Sivaraman, Siau-Cheng Khoo, and Jing Xu. 2017b. GEMS: An extract method refactoring recommender. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 24–34.
- Limei Yang, Hui Liu, and Zhendong Niu. 2009. Identifying fragments to be extracted from long methods. In *2009 16th Asia-Pacific Software Engineering Conference*. IEEE, 43–49.
- Norihiro Yoshida, Seiya Numata, Eunjong Choiz, and Katsuro Inoue. 2019. Proactive clone recommendation system for extract method refactoring. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWor)*. IEEE, 67–70.

- Ruru Yue, Zhe Gao, Na Meng, Yingfei Xiong, Xiaoyin Wang, and J. David Morgenthaler. 2018. Automatic clone recommendation for refactoring based on the present and the past. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 115–126.
- Jingxuan Zhang, Junpeng Luo, Jiahui Liang, Lina Gong, and Zhiqiu Huang. 2023. An accurate identifier renaming prediction and suggestion approach. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2023), 1–51.

Received 19 January 2025; revised 6 May 2025; accepted 6 July 2025