# C$^3$: Code Clone-based Identification of Duplicated Components

Yanming Yang
Zhejiang University, China
yanmingyang@zju.edu.cn

Ying Zou
Queen's University, Canada
zouy@queensu.ca

Xing Hu*
Zhejiang University, China
xinghu@zju.edu.cn

David Lo
Singapore Management University, Singapore
davidlo@smu.edu.sg

Chao Ni
Zhejiang University, China
chaoni@zju.edu.cn

John Grundy
Monash University, Australia
john.grundy@monash.edu

Xin Xia
Huawei, China
xin.xia@acm.org

## ABSTRACT

*Reinventing the wheel* is a detrimental programming practice in software development that frequently results in the introduction of duplicated components. This practice not only leads to increased maintenance and labor costs but also poses a higher risk of propagating bugs throughout the system. Despite numerous issues introduced by duplicated components in software, the identification of component-level clones remains a significant challenge that existing studies struggle to effectively tackle. Specifically, existing methods face two primary limitations that are challenging to overcome: 1) Measuring the similarity between different components presents a challenge due to the significant size differences among them; 2) Identifying functional clones is a complex task as determining the primary functionality of components proves to be difficult.

To overcome the aforementioned challenges, we present a novel approach named **C$^3$** (Component-level Code Clone detector) to effectively identify both textual and functional cloned components. In addition, to enhance the efficiency of eliminating cloned components, we develop an assessment method based on six component-level clone features, which assists developers in prioritizing the cloned components based on the refactoring necessity.

To validate the effectiveness of C$^3$, we employ a large-scale industrial product developed by Huawei, a prominent global ICT company, as our dataset and apply C$^3$ to this dataset to identify the cloned components. Our experimental results demonstrate that C$^3$ is capable of accurately detecting cloned components, achieving impressive performance in terms of precision (0.93), recall (0.91), and F1-score (0.9). Besides, we conduct a comprehensive user study to further validate the effectiveness and practicality of our assessment method and the proposed clone features in assessing the refactoring necessity of different cloned components. Our study establishes solid alignment between assessment outcomes and participant responses, indicating the accurate prioritization of clone components with a high refactoring necessity through our method. This finding further confirms the usefulness of the six "golden features" in our assessment.

*Corresponding author: Xing Hu.

*ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA*
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0327-0/23/12...$15.00
https://doi.org/10.1145/3611643.3613883

## CCS CONCEPTS

• **Software and its engineering → Maintaining software**.

## KEYWORDS

Component-level Clone Detection, Component-level Clone Metric, Community Detection Algorithm

**ACM Reference Format:**
Yanming Yang, Ying Zou, Xing Hu*, David Lo, Chao Ni, John Grundy, and Xin Xia. 2023. C$^3$: Code Clone-based Identification of Duplicated Components. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3611643.3613883

## 1 INTRODUCTION

*Reinventing the wheel* refers to the practice of developing a solution or functionality from scratch that already exists and is well-established in the software development community [11]. While *reinventing the wheel* may be necessary in certain situations to address software licensing incompatibilities or technical limitations in third-party modules [11], it often emerges as a poor programming practice during development and has become a prevalent issue in the industry. This practice often arises due to developers' limited programming skills or lack of awareness, resulting in unnecessary duplication of effort despite existing solutions or technologies already meeting the requirements or solving the problem at hand [12]. It demonstrates a disregard for the knowledge and experience accumulated by others, leading to inefficiencies, wasted resources, and development delays [36]. Even in some cases, developers may remain unaware that they have reinvented the wheel even after completing the task. Therefore, it is of utmost importance to address these issues by detecting the practice of *reinventing the wheel* and identifying instances of *duplicated wheels* within software systems. These *duplicated wheels* can be found in various aspects of software development, including algorithms, libraries, and frameworks, representing complete functionalities within the software. Unlike code clones discussed in existing studies [49], these *duplicated wheels* often manifest as components in software that encompass a wider scope of code, spanning multiple source files [8]. Consequently, identifying duplicated components presents a substantial challenge.

In recent decades, numerous studies [15, 28, 40, 41, 43, 53, 57, 61, 62] have proposed various methods to detect duplicated code fragments in software. However, these code clone detectors can only identify clones at the line, method, and file levels, and they have limited capability to identify duplicated software components that span multiple source files. This limitation arises due to the two primary challenges in identifying duplicated components, which are particularly difficult to address:

1) *Existing studies face challenges in measuring the similarity between software components due to the significant volume of source code involved.* Existing methods are limited to calculating the similarity of code fragments at a granularity no higher than the file level. Consequently, when a group number of similar code fragments are dispersed across different source files within a pair of components, existing methods fail to recognize the similarity between the two components, or even the similarity between the source files themselves. Thus, existing methods are difficult in their applicability to identify duplicated components.

2) *Existing studies struggle to accurately determine the primary functionality of a component, resulting in their inability to detect components with similar functionalities.* Apart from cloned components that display textual similarities, there exist functionally similar components. However, current methods lack the ability to generalize the primary functionality of a component, resulting in their inability to detect functionally cloned components.

To address these challenges, we develop a novel approach called $C^3$ (**C**omponent-level **C**ode **C**lone detector), which enables the identification of both textual and functional cloned components within the software. Specifically, we begin by gathering components from the software and detecting cloned files among them. Subsequently, we devise a method to assess the similarity between components based on the count of cloned files within components. We construct a graph structure to describe the relationships of similarity among components, allowing us to transform the task of identifying textual and functional clones into detecting nodes with close relationships and similar structures within the graphs. Finally, we employ a graph-based clustering algorithm that utilizes the connectivity patterns of nodes to identify both textual and functional cloned components effectively. Furthermore, to support developers in effectively addressing cloned components and improving code quality, reliability, and compatibility, we present an assessment method that incorporates six component-level clone features to evaluate the priority of each cloned component by assessing its necessity for refactoring.

To demonstrate the effectiveness of $C^3$, we utilize a large-scale industrial product developed by Huawei, a leading ICT company, as our dataset. This dataset consists of 2,099 components. We apply $C^3$ to detect cloned components within this dataset. The evaluation results demonstrate that $C^3$ effectively identifies component clones, achieving a precision of 0.93, a recall of 0.91, and an F1-score of 0.9. Furthermore, we conduct a user study to validate the correctness of our assessment method and the utility of the proposed six component-level clone features. The study result demonstrates that our assessment method can accurately determine which cloned components should be given higher priority for refactoring based on six valuable component-level clone features. It achieves over 90% in all three evaluation metrics on average, highlighting its accuracy and reliability. In summary, our study makes the following contributions:

(1) To the best of our knowledge, we are the first to develop a practical tool, $C^3$, to detect both textual and functional cloned components within software systems.
(2) We develop an assessment method that creates six component-level clone features to effectively measure the refactoring necessity of cloned components.
(3) We utilize a well-established industrial product developed by a globally renowned ICT company as our dataset to thoroughly evaluate the effectiveness and practicality of $C^3$. The experimental results conclusively demonstrate that $C^3$ can accurately detect cloned components within the software, attaining exceptional performance in terms of precision, recall, and F1-score.
(4) We conduct a user study to evaluate the effectiveness of our assessment method and the practicality of the six clone features. The study results demonstrate outstanding performance across all metrics, providing compelling evidence for the utility of the features and confirming the accuracy of the assessment method.

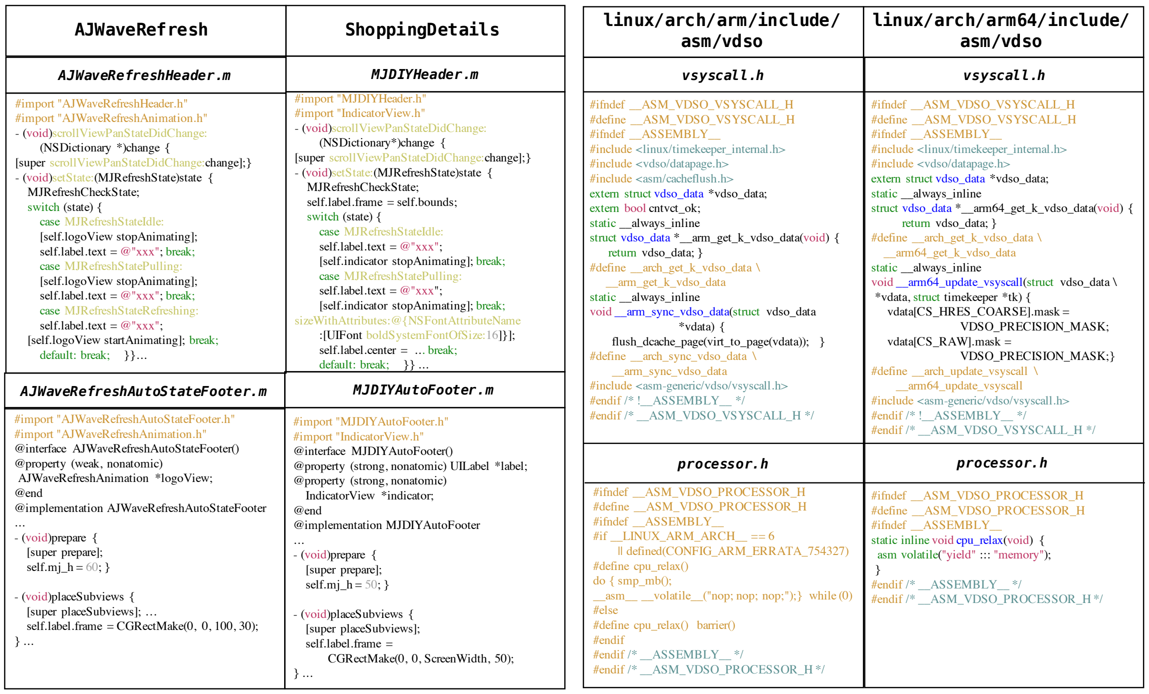
## 2 MOTIVATION

In this section, we initially present two common industrial scenarios leading to cross-project and within-project cloned components. Next, we highlight two real-world cloned component pairs detected by $C^3$ across different projects, enhancing readers' comprehension.

### 2.1 Industrial Application Scenarios

*2.1.1 Scenario One: Cross-project Cloned Components.* Jack and Nina are software developers employed in separate departments within the same company. In Department A, Jack is tasked with developing a commenting module for a music application, while Nina, in Department B, is responsible for implementing a similar commenting module for a social application. Since Jack and Nina work in different departments, they are unaware of each other's existence despite having similar development assignments. Following the company's coding conventions, they both use the same programming language and framework to complete their respective commenting modules. As a result, after completing their tasks, both Department A and Department B have their own commenting modules integrated into their respective applications, and the source code of these two components exhibits a high degree of similarity. The presence of these two redundant components increases maintenance expenses.
**Solution.** They can be refactored into a shared component, allowing for broader utilization, consolidation, and reuse across various departments within the company.

*2.1.2 Scenario Two: Within-project Cloned Components.* Bob, a recently hired software developer, is assigned to a coding task at his new company. His objective is to add a new feature to an existing component, Component C, in the latest version. As Bob delves into the development process, he realizes that this component is quite large and encompasses intricate dependencies. This implies that directly modifying Component C may pose potential risks to the



AJWaveRefresh

AJWaveRefreshHeader.m

```
#import "AJWaveRefreshHeader.h"
#import "AJWaveRefreshAnimation.h"
- (void)scrollViewPanStateDidChange:
    (NSDictionary *)change {
[super scrollViewPanStateDidChange:change];}
- (void)setState:(MJRefreshState)state {
  MJRefreshCheckState;
  switch (state) {
    case MJRefreshStateIdle:
    [self.logoView stopAnimating];
    self.label.text = @"xxx"; break;
    case MJRefreshStatePulling:
    [self.logoView stopAnimating];
    self.label.text = @"xxx"; break;
    case MJRefreshStateRefreshing:
    self.label.text = @"xxx";
  [self.logoView startAnimating]; break;
    default: break;   }}...
```

AJWaveRefreshAutoStateFooter.m

```
#import "AJWaveRefreshAutoStateFooter.h"
#import "AJWaveRefreshAnimation.h"
@interface AJWaveRefreshAutoStateFooter()
@property (weak, nonatomic)
 AJWaveRefreshAnimation *logoView;
@end
@implementation AJWaveRefreshAutoStateFooter
...
- (void)prepare {
  [super prepare];
  self.mj_h = 60; }

- (void)placeSubviews {
  [super placeSubviews]; ...
  self.label.frame = CGRectMake(0, 0, 100, 30);
} ...
```

ShoppingDetails

MJDIYHeader.m

```
#import "MJDIYHeader.h"
#import "IndicatorView.h"
- (void)scrollViewPanStateDidChange:
    (NSDictionary*)change {
[super scrollViewPanStateDidChange:change];}
- (void)setState:(MJRefreshState)state {
  MJRefreshCheckState;
  self.label.frame = self.bounds;
  switch (state) {
    case MJRefreshStateIdle:
    self.label.text = @"xxx";
    [self.indicator stopAnimating]; break;
    case MJRefreshStatePulling:
    self.label.text = @"xxx";
    [self.indicator stopAnimating]; break;
sizeWithAttributes:@{NSFontAttributeName
    :[UIFont boldSystemFontOfSize:16]}];
    self.label.center = ... break;
    default: break;   }} ...
```

MJDIYAutoFooter.m

```
#import "MJDIYAutoFooter.h"
#import "IndicatorView.h"
@interface MJDIYAutoFooter()
@property (strong, nonatomic) UILabel *label;
@property (strong, nonatomic)
  IndicatorView *indicator;
@end
@implementation MJDIYAutoFooter
...
- (void)prepare {
  [super prepare];
  self.mj_h = 50; }

- (void)placeSubviews {
  [super placeSubviews];
  self.label.frame =
      CGRectMake(0, 0, ScreenWidth, 50);
} ...
```

linux/arch/arm/include/asm/vdso

vsyscall.h

```
#ifndef __ASM_VDSO_VSYSCALL_H
#define __ASM_VDSO_VSYSCALL_H
#ifndef __ASSEMBLY__
#include <linux/timekeeper_internal.h>
#include <vdso/datapage.h>
#include <asm/cacheflush.h>
extern struct vdso_data *vdso_data;
extern bool cntvct_ok;
static __always_inline
struct vdso_data *__arm_get_k_vdso_data(void) {
   return vdso_data; }
#define __arch_get_k_vdso_data \
   __arm_get_k_vdso_data
static __always_inline
void __arm_sync_vdso_data(struct vdso_data
                  *vdata) {
    flush_dcache_page(virt_to_page(vdata));   }
#define __arch_sync_vdso_data \
    __arm_sync_vdso_data
#include <asm-generic/vdso/vsyscall.h>
#endif /* !__ASSEMBLY__ */
#endif /* __ASM_VDSO_VSYSCALL_H */
```

processor.h

```
#ifndef __ASM_VDSO_PROCESSOR_H
#define __ASM_VDSO_PROCESSOR_H
#ifndef __ASSEMBLY__
#if __LINUX_ARM_ARCH__ == 6
    || defined(CONFIG_ARM_ERRATA_754327)
#define cpu_relax()
do { smp_mb();
__asm__ __volatile__("nop; nop; nop;");}  while (0)
#else
#define cpu_relax()  barrier()
#endif
#endif /* __ASSEMBLY__ */
#endif /* __ASM_VDSO_PROCESSOR_H */
```

linux/arch/arm64/include/asm/vdso

vsyscall.h

```
#ifndef __ASM_VDSO_VSYSCALL_H
#define __ASM_VDSO_VSYSCALL_H
#ifndef __ASSEMBLY__
#include <linux/timekeeper_internal.h>
#include <vdso/datapage.h>
extern struct vdso_data *vdso_data;
static __always_inline
struct vdso_data *__arm64_get_k_vdso_data(void) {
    return vdso_data; }
#define __arch_get_k_vdso_data \
  __arm64_get_k_vdso_data
static __always_inline
void __arm64_update_vsyscall(struct vdso_data \
 *vdata, struct timekeeper *tk) {
   vdata[CS_HRES_COARSE].mask =
                 VDSO_PRECISION_MASK;
   vdata[CS_RAW].mask =
                 VDSO_PRECISION_MASK;}
#define __arch_update_vsyscall \
    __arm64_update_vsyscall
#include <asm-generic/vdso/vsyscall.h>
#endif /* !__ASSEMBLY__ */
#endif /* __ASM_VDSO_VSYSCALL_H */
```

processor.h

```
#ifndef __ASM_VDSO_PROCESSOR_H
#define __ASM_VDSO_PROCESSOR_H
#ifndef __ASSEMBLY__
static inline void cpu_relax(void) {
 asm volatile("yield" ::: "memory");
}
#endif /* __ASSEMBLY__ */
#endif /* __ASM_VDSO_PROCESSOR_H */
```

**Figure 1: Two pairs of real-world cloned components.**

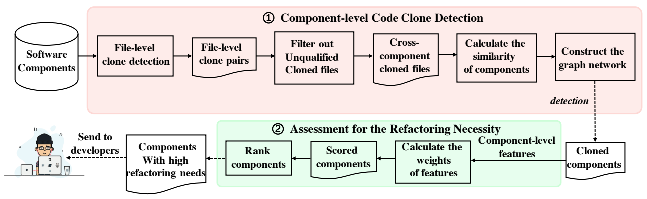

① Component-level Code Clone Detection
Software Components
File-level clone detection
File-level clone pairs
Filter out Unqualified Cloned files
Cross-component cloned files
Calculate the similarity of components
Construct the graph network
detection
② Assessment for the Refactoring Necessity
Cloned components
Component-level features
Calculate the weights of features
Scored components
Rank components
Components With high refactoring needs
Send to developers

**Figure 2: The overall framework of our approach.**

proper functioning of certain business logic. Given his limited familiarity with the business logic associated with both the components that depend on Component C and the components that Component C relies on, Bob decides to mitigate potential issues by creating a new component, Component D. This allows him to narrow down the scope of impact and avoid introducing unforeseen complications. Bob proceeds by copying and pasting the necessary source code from Component C and implementing the new feature within Component D. Upon completing his coding task, the system now consists of two similar components: Components C and D. However, Component D is only invoked when running the new feature, and the implementation of such duplicated components consumes a significant amount of developers' time and effort.

**Solution.** To eliminate the duplicated component, it is imperative to integrate Component D into Component C.

## 2.2 Two Pairs of Real-world Cloned Components

The first pair includes two cloned components, namely *AJWaveRefresh [1]* and *ShoppingDetails [2]*, which both have a commonly used functionality for implementing transition animations. Each component comprises eight source files, and half of the source files in both components exhibit significant similarities to each other. In Fig. 1, we showcase two sets of similar source files out of a total of four pairs of similarities. The two components are a pair of cross-project component-level clones, belonging to different systems. The second pair of cloned components originate from a renowned open-source software system, Linux [3], making it a within-project clone pair. They comprise five and six source files, respectively, with four of the files in each component demonstrating high similarities to each other. Similarly, Fig. 1 showcases two pairs of similar files out of the four pairs present in this clone pair.

**Summary.** From these two motivating examples, it is evident that even components with distinct path information, component names, file names, or residing in different systems can exhibit substantial similarity in terms of both textual content and functionalities.

## 3 APPROACH

The overall workflow of our approach is illustrated in Fig. 2, consisting of two primary phases: **❶ component-level code clone detection ($C^3$)** and **❷ assessing the refactoring necessity of cloned components**. In this section, we provide a detailed explanation of each phase of our approach.

## 3.1 Component-level Code Clone Detection

$C^3$ performs component-level clone detection through three primary steps: 1) identifying file-level clones among components; 2) calculating the similarity between components; and 3) detecting component-level code clones.

*3.1.1 Identifying File-level Clones among Components.* Given the substantial variation in code volume among components, fine-grained detection methods like line-level and function-level can be impractical. Therefore, we opt for file-level granularity to effectively characterize the similarity between components. Specifically, our initial step involves identifying file-level clones from software

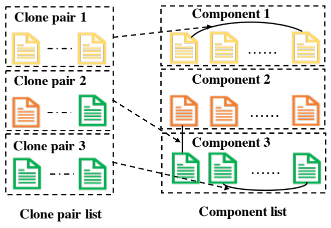


**Figure 3: Two categories of file-level clone pairs in components.**

components, which serve as the basis for calculating the similarity between different components. To ensure the effectiveness and utility of C$^3$ in the industry, the used clone detector requires to meet the following requirements: 1) *It should be capable of effectively identifying file-level clones from software.* 2) *It should be capable of being applied to software of varying scales, demonstrating high scalability.* 3) *Its detection time should be minimized for optimal efficiency.* 4) *It should be widely recognized as a well-established and commonly used tool.* Hence, after pre-processing the raw files according to the steps in existing work [51], we select the qualified tool, SourcererCC [29, 51], to identify clone pairs from components.

*3.1.2 Calculating the Similarity between Components.* The file-level clone pairs can be classified into two distinct categories: 1) *Within-component Clone Pairs*, which consist of two cloned files belonging to the same component. 2) *Cross-component Clone Pairs*, which include two cloned files originating from different components. As shown in Fig. 3, the left section displays three pairs of cloned files, while the right section depicts the corresponding components in which these pairs are detected. Source files within each component are depicted using a consistent color scheme. We observe that the first and last pairs of files represent *Within-component Clone Pairs*, as they share the same color, indicating that they belong to the same component. On the other hand, the source files in the second clone pair have different colors, suggesting that they belong to different components. Therefore, this clone pair falls under the category of *Cross-component Clone Pairs*.

During our endeavor to capture the similarity among different components, we observe from the three clone pair examples that *Within-component Clone Pairs* only contribute to evaluating the similarity within a component, while *Cross-component Clone Pairs* assist in calculating the similarity among different components. Based on this observation, we exclude the relationships of *within-component clone pairs* and focus solely on the relationships of *cross-component clone pairs* among components. We then leverage the relationships among cross-component clone pairs to quantify the similarity between two components. The specific equation is as follows:

$$Sim_{IJ} = \frac{CF_{IJ}}{F_I} \tag{3.1}$$

Where $Sim_{IJ}$ represents the ratio of the number of *Component I*'s source files cloned from *Component J* to the total number of files in *Component I*, serving as a measure of the similarity of *Component I* to *Component J*. $CF_{IJ}$ denotes the count of source files in *Component I* that exhibit similarity to the files in *Component J*, and $F_I$ represents the total number of source files in *Component I*.

Note that due to differences in the number of source files involved in each component, Equation 3.1 always produces two distinct values, namely $Sim_{IJ}$ and $Sim_{JI}$, which describe the similarity between two components. The two values represent the similarity of *Component I* to *Component J* (i.e., $Sim_{IJ}$ ) and the similarity of *Component J* to *Component I* (i.e., $Sim_{JI}$), respectively. To identify as many cloned components as possible, we consider the larger value as the measure of similarity between two components. This is because, for example, if *Component I* consists of 62 source files and *Component J* consists of 12 source files, even though all the files in *Component J* are cloned from files in *Component I*, the similarity of *Component I* to *Component J* ($Sim_{12}$) is relatively low at around 0.19. On the other hand, the similarity of *Component J* to *Component I* ($Sim_{21} = 1$) provides a larger value, which better captures the similarity between these two components to a certain extent. Hence, we take $Sim_{21}$ as the similarity between *Component I* and *Component J*.

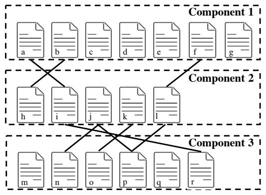


**Figure 4: A example for calculating the similarity of components.**

To provide a concrete illustration, we utilize Fig. 4 as an example to calculate the similarity between various components. Specifically, in Fig. 4, we observe three components, namely *Component 1*, *Component 2*, and *Component 3*, consisting of seven, five, and six source files, respectively. *Component 1* and *Component 2* exhibit three pairs of cloned files, while *Component 2* and *Component 3* show four pairs of cloned files. According to Equation 3.1, the similarity of *Component 1* to *Component 2* is $CR_{12} = \frac{3}{7}$, while the similarity of *Component 2* to *Component 1* is $CR_{21} = \frac{3}{5}$. Therefore, the final similarity between the two components is determined by the larger value, which in this case is $\frac{3}{5}$. Similarly, the similarity between *Component 2* and *Component 3* is $\frac{4}{5}$.

*3.1.3 Detecting Component-level Code Clones.* Equation 3.1 can only describe the textual similarity between components. To overcome the second limitation discussed in Section 1, we construct a graph structure based on the textual similarity between components to capture the functional similarity between components. Specifically, each node in the graph represents a component, and a weighted edge connecting two components signifies their level of similarity. The weight of the edge corresponds to their textual similarity value. Note that to distinguish from textual similarity, in our study, functional clones are defined as components that exhibit similar primary functionalities but with low textual similarity.

To further clarify the motivation behind utilizing the graph structure for identifying functional clones, let us consider an illustrative example. It becomes apparent from Fig. 5 that both *Component u* and *Component v* demonstrate textual similarity with the same set of components, namely Components *a, b, c, d, and e*, despite exhibiting no similarity to each other. Upon thorough manual examination, we discover that *Component u* and *Component v* are functional clones of each other. Furthermore, we observe a recurring pattern where components, such as *Component u* and *Component v*, which exhibit low textual similarities but are similar to the same set of other components, often turn out to be functional clones. After careful analysis, this observation can be easily explained. When multiple components

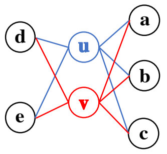


**Figure 5: The structural relationship of functional clones.**

exhibit similarity to the same set of components even with low textual similarity, it suggests that they are commonly utilized in similar application scenarios and possess similar functionalities.

Therefore, building upon this finding, we transform the task of identifying functional clones into detecting nodes that exhibit similar structures with the same set of nodes, while showing no relationships with each other. Considering that the transformed task can be effectively addressed with the assistance of knowledge from the field of the *Community Network*, we adopt the *community detection algorithm*, which is a graph-based clustering algorithm [26], specifically designed to identify clusters containing highly weighted similar nodes (textual clones) and communities consisting of nodes (functional clones) with high structural similarity from the graph. Hence, by leveraging this algorithm, C$^3$ can identify both textual and functional clones from components. Furthermore, employing the *community detection algorithm* offers the advantage of not requiring the pre-determination of the number of clusters.

During the construction of the graph structure, we establish a weighted edge between two components when they exhibit a certain degree of textual similarity. However, we observe that the performance of C$^3$ can vary depending on the establishment of weighted edges based on different degrees of textual similarity. If the threshold for the similarity between components is set too low, numerous pairs of components with low similarity are connected in the graph, potentially altering the overall structure. This change in structure can lead the community detection algorithm to mistakenly identify component pairs that are not similar as clones, resulting in unrelated components being grouped into clone clusters. On the other hand, if we only connect component pairs with high similarity in the graph, there is a risk of overlooking numerous real cloned components. These missed components may include functional clones and other cloned components that do not exhibit high textual similarity with others. To ensure the accurate identification of as many cloned components as possible, we introduce a parameter $k$ as the threshold for the minimum similarity between components. When the similarity between a pair of components exceeds this threshold, the two components can be connected in the graph through a weighted edge. For example, when $k = 0.5$, it means that the connected nodes in the graph must have a textual similarity of at least 0.5. The textual similarity values between these nodes are then assigned as weights to their respective edges in the graph. Finally, C$^3$ utilizes a community detection algorithm to identify both textual and functional clones from the graph based on the structure established by weighted edges.

## 3.2 Assessing the Refactoring Necessity of Cloned Components

To enhance developers' productivity in addressing detrimental cloned components, we propose a method for assessing the refactoring necessity of cloned components. Specifically, we introduce six clone features at the component level that serve as criteria for evaluating and determining the refactoring priority of cloned components. Next, based on the findings of a user survey, we assign weights to the six features. These weighted features are then utilized to assess the refactoring necessity of different cloned components. We provide a detailed description of the aforementioned three steps as follows.

### 3.2.1 Studied Features.

In this section, we present six studied features that effectively capture the impact of cloned components on software quality and assess their prioritization for refactoring. These six features are summarized as follows:

***F1: Average Clone Ratio (ACR)*** refers to the overall similarity of a component to others. Intuitively, developers should prioritize refactoring components whose source files have substantial clones in other components. Based on this intuition, we propose this feature which can be quantified using the following equation:

$$ACR_P = \frac{\sum_{n=0}^{m} Sim_{PC_n}}{m} \tag{3.2}$$

Here, $ACR_P$ represents the ACR of *Component P*, $m$ denotes the number of components that contain cloned files of *Component P*, and $C_n$ represents the nth component that contains similar source files to *Component P*. $Sim_{PC_n}$ denotes the similarity between *Component P* and the nth component. The value of this feature is ranged between 0 and 1. In particular, when ACR equals 0, it means that there are no other components whose source files are similar to the files in the current component. On the other hand, when ACR equals 1, it indicates that all files in the current component have clones in other components. We provide an example to facilitate better understanding. In Figure 4, when calculating the ACR of *Component 2*, denoted as $ACR_2$, we need to consider all components that contain similar source files to *Component 2*, including *Component 1* and *Component 3*. Based on Equation 3.2, $ACR_2$ thus equals $\frac{\frac{3}{5}+\frac{4}{5}}{2}$.

***F2: Number of Calls (NCL)*** refers to the count of calls made to a cloned component. It serves as a measure of the component's importance to the software to a certain extent. Components with a high value of NCL indicate that they are frequently used and invoked within the software. These components hold vital functions and take precedence in developers' refactoring efforts.

***F3: Average File Size (AFS)*** refers to the average file size of a cloned component. A component is typically considered a code unit with an independent functionality. Previous research [42] has revealed that larger and more complex code units are more likely to exhibit poor quality and intricate dependencies. Consequently, a component with a higher AFS may necessitate maintenance and refactoring.

***F4: Component Size (CS)*** refers to the number of source files within a cloned component. It exhibits a similar trend to AFS, whereby larger component sizes pose greater challenges for developers in terms of maintenance, consequently heightening the potential risks to software quality.

***F5: Clone Group Size (CGS)*** refers to the number of components within a clone group. A larger clone group indicates that the primary functionality of cloned components in that group is widely used and frequently reused within the software. Hence, components within such clone groups should be assigned high priority for refactoring.

***F6: Clones of Components (CC)*** refers to the number of copies that exist for a cloned component in the software. When a component has a substantial number of clones within the software system, it indicates potential design issues. Therefore, it is crucial for developers to prioritize refactoring such components to improve the overall software quality and maintainability.

#### 3.2.2 *Weight Assignment for Studied Features.*

After identifying these clone features, we conduct a user survey to quantitatively assess the significance of these features in evaluating the refactoring necessity of cloned components. We follow Kitchenham and Pfleeger's guidelines for personal opinion surveys [21, 64]. The process in this section can be classified into three steps: 1) survey design and 2) survey distribution and 3) survey analysis. We describe each step in detail as follows:

🔔**Survey Design.** Our survey consists of two parts. The first part collects demographic information from respondents to filter out individuals who may not have a strong understanding of our survey. For this purpose, we ask the question "*What best describes your main responsibilities in your projects?*" and provide options, including *1) developer, 2) software tester, 3) software maintainer, 4) user, and 5) other*. Respondents are required to select the option that best describes their job roles in their projects. We only collect and analyze responses given by developers and maintainers since they are more likely to have extensive experience in using and developing software components, compared to other roles with limited involvement in component development activities.

In the second part of the survey, we present six component-level features and ask respondents to select the features they believe could serve as metrics for assessing the refactoring necessity of a cloned component. We intentionally do not impose a limit on the number of features they can choose, allowing respondents to select all the relevant features they find applicable. For example, some respondents may choose F1 and F2, indicating their belief that F1 (Average Clone Ratio) and F2 (Number of Calls) are two critical factors for evaluating the necessity of refactoring cloned components. After collecting the responses, we calculate the choice rate for each feature, which represents the proportion of times the feature is selected compared to the total number of responses. These choice rates are then utilized as weights for the corresponding clone features, reflecting the importance of each feature.

🔔**Survey Distribution.** To encourage meaningful responses, we sent email invitations to 120 experienced software practitioners from Huawei, who have over five years of industry experience, inviting them to participate in our survey. Given that Huawei follows a component-level development method [8] in their software systems, we anticipate that their software practitioners would possess a deeper understanding of software components. We receive a total of 97 responses, resulting in a response rate of 80.8%. After excluding responses from individuals whose job roles do not fall under the categories of developers or maintainers, we are left with 81 valid responses. Among these, 16 respondents are developers of the system under study, possessing in-depth knowledge of the system, while the remaining participants are experienced software practitioners.

🔔**Survey Analysis.** After receiving the survey responses, we analyze the collected data following the guidelines [20]. The analysis results reveal that F1 (Average Clone Ratio), F2 (Number of Calls), and F6 (Number of Clones) are the three key features for assessing the prioritization for refactoring cloned components, as they are selected by over 55% of the respondents. Particularly, more than 70% of the respondents consider F2 to be a crucial feature. The second most important features are F1 and F6, with 48 respondents recognizing their significance, accounting for 59.3% of the total. Furthermore, F3 (Average File Size) has a selection rate of approximately 22.2%, and F4 (Component Size) has a selection rate of approximately 25.9%. For each feature, the choice rate is regarded as its weight, indicating its importance in assessing the necessity of refactoring cloned components. Therefore, the weights assigned to the six features are as follows: F1 (0.593), F2 (0.704), F3 (0.222), F4 (0.259), F5 (0.111), and F6 (0.593).

#### 3.2.3 *Determining the Prioritization of Refactoring Cloned Components.*

To assess the refactoring necessity of cloned components, we utilize the six weighted features to calculate the priority score of each component using the following equation:

$$Score(CC) = \sum_{i=1}^{6} W_i \times \frac{F_i - Min(F_i)}{Max(F_i) - Min(F_i)} \tag{3.3}$$

where $Score(CC)$ denotes the score of a cloned component, $F_i$ represents the ith feature value of a cloned component, and $W_i$ denotes the weight assigned to that feature. $Max(F_i)$ represents the maximum value among all cloned components' ith feature values, while $Min(F_i)$ represents the minimum value. A cloned component with a high score signifies that it should be prioritized for refactoring, as it has the potential to significantly impact software quality.

## 4 EVALUATION

In this section, we begin by introducing the industrial system utilized in our study, followed by a detailed description of the dataset preparation process. Subsequently, we provide an overview of the experimental settings.

### 4.1 Industrial System

In our study, we **utilize a vast and widely adopted commercial router platform as our case study dataset**. Employed by millions globally, it encompasses the source code of the latest routing system version, evolving from four previous iterations. With billions of lines of code, it spans over 470 file types, including six programming languages: C (51,830 files), C++ (1,084 files), Python (764 files), Java (73 files), Lua (22,277 files), and Bash (11,202 files). Each language has a distinct role, and we prioritize C due to its prominence within the platform's statistics.

### 4.2 Dataset Preparation

In this section, we commence by outlining the process of collecting components from the studied system. We then present the procedure for constructing a ground-truth dataset for evaluation purposes.

#### 4.2.1 *Software Component Collection.*

Components in various software systems can manifest in diverse forms. In our studied system, components are packaged using the CMake tool [4]. CMake is an open-source and cross-platform tool that assists developers in packaging software products and managing the software compilation process by generating native configuration files. These configuration files store essential information about software components, such as their names and associated source files. Thus, we extract components from the studied system based on component information recorded in these configuration files.

#### 4.2.2 *Construction of Ground-Truth Dataset for Evaluation.*

To ensure the accuracy and validity of our evaluation experiments, we meticulously construct a ground-truth dataset by manually identifying and labeling cloned components. Given the significant cost of this process, we devise three steps to identify cloned components, with the goal of minimizing labor costs while ensuring the quality of the provided labels. Besides, to minimize the possibility of human error, we engage the expertise of two highly experienced developers with more than five years of industrial experience in C language development to carry out the labeling process. Specifically, based on their experience, they independently follow the three steps we design to identify cloned components, respectively. Subsequently, they engage in discussions to address any discrepancies and reach a consensus, ensuring the integration of their respective findings into the final version. To evaluate their inter-rater agreement, we employ Cohen's Kappa [39], a commonly used metric in research [59, 60]. The analysis reveals a robust value of 0.86, indicating a strong level of agreement between the two experts. The whole labeling process spans over a period of three months, during which they diligently identify 211 clone groups comprising a total of 1,066 components. Note that manual examination is involved in each step of the labeling process to ensure accuracy and correctness. In the following sections, we elaborate on the three steps in detail.

**Step 1: Identifying cloned components through package names.** Upon careful examination, we observe that numerous cloned components, especially within-project cloned components, exhibit similarities in their package names. For instance, as exemplified in the second motivation example depicted in Fig. 1, a pair of cloned components [9, 10] belonging to the "*vdso*" package in the Linux system, displaying identical package names. Drawing from this discovery, to streamline the labeling process, two developers first identify component pairs with identical package names, resulting in a total of 8,548 component pairs. Subsequently, a meticulous manual examination is carried out to determine whether each pair of components is cloned based on components' comments and source code. If substantial similarities are identified, the components are assigned identical labels and organized into corresponding clone groups according to their shared functionalities.

**Step 2: Identifying cloned components through file names.** Aside from package names, source files with the same name often exhibit a high degree of similarity, as exemplified by the second pair of cloned components in the motivation example. Therefore, after completing Step One, the two developers focus on identifying component pairs with similar file names from the remaining components. Specifically, they select component pairs where more than half of the source file names in both components exhibit similarities greater than 0.8 to each other. This yields 13,613 component pairs meeting these criteria, which are then manually labeled based on functionality and source code.

**Step 3: Identifying cloned components from the remaining ones.** After completing the aforementioned two steps, over 1,100 components are unlabeled. They manually examine these remaining components and assign them to respective clone groups based on their functionalities. As a result, in this step, they identify a total of 152 pairs of functionally cloned components.

## 4.3 Experimental Settings

We implement C$^3$ using Python, and our experiments are conducted on Ubuntu v20.04.1 64-bit OS with an RTX3090-24GB GPU, ensuring reliable and replicable results. The replication package provides additional information on model settings for easy access.

## 5 RESULTS

To gain a comprehensive understanding of the performance of C$^3$ and our assessment method, we analyze our evaluation results by addressing three research questions (RQs):

(1) RQ1: How effective is C$^3$ under different parameter settings?
(2) RQ2: Which community detection algorithm is optimal for C$^3$?
(3) RQ3: How effective is our assessment method?

### 5.1 RQ1: How effective is C$^3$ under different parameter settings?

*5.1.1 Motivation.* Our study is the first to introduce a novel clone detector, C$^3$, designed to automatically identify cloned components in software systems. In this RQ, we aim to 1) validate the effectiveness of C$^3$ and 2) determine the optimal for the parameter $k$.

*5.1.2 Method.* To determine the optimal setting for $k$, we evaluate the performance of C$^3$ under various parameter configurations. In this evaluation, we vary the parameter $k$ from 0 to 1 in steps of 0.1. Additionally, in this RQ, C$^3$ employs the Louvain algorithm [19] as the community detection algorithm for identifying cloned components. Note that since there are no previous studies proposing similar tools for the identification of cloned components, no baseline methods are available for comparison with the performance of C$^3$.

*5.1.3 Metrics.* We utilize three commonly used evaluation metrics to validate C$^3$'s performance, including, *Precision*, *Recall*, and *F1-score*. Precision quantifies the percentage of accurately identified cloned components out of all the components detected by C$^3$. On the other hand, Recall measures the proportion of C$^3$'s detected components that account for real cloned components in the dataset. In addition, we evaluate the *running time* (*Time*) of C$^3$ during the clone detection process.

*5.1.4 Results.* Table 1 summarizes C$^3$ performance for varying $k$, highlighting the best results. From these results, the following observations arise:

1) **C$^3$ can effectively detect cloned components, and 0.6 is the optimal setting for the parameter $k$.** As shown in Table 1, C$^3$

**Table 1: The performance of C$^3$ under different $k$.**

| Parameter(k) | Precision | Recall | F1-score | Time(s) |
|---|---|---|---|---|
| **k = 0** | 0.58 | 0.90 | 0.61 | **4.97** |
| **k = 0.1** | 0.66 | 0.91 | 0.68 | 3.74 |
| **k = 0.2** | 0.69 | 0.90 | 0.72 | 3.38 |
| **k = 0.3** | 0.73 | 0.91 | 0.75 | 2.66 |
| **k = 0.4** | 0.80 | **0.92** | 0.81 | 2.12 |
| **k = 0.5** | 0.86 | **0.92** | 0.85 | 2.13 |
| ***k = 0.6*** | **0.93** | 0.91 | **0.90** | 2.56 |
| **k = 0.7** | 0.93 | 0.87 | 0.86 | 1.50 |
| **k = 0.8** | 0.94 | 0.79 | 0.82 | 1.34 |
| **k = 0.9** | 0.93 | 0.77 | 0.80 | 1.05 |
| **k = 1.0** | 0.92 | 0.73 | 0.76 | **0.80** |
| ***Avg.*** | *0.82* | *0.86* | *0.78* | *2.39* |

**Table 2: The performance of the $C^3$ using different algorithms.**

| | Girvan–Newman Algorithm | | | | Fast-Greedy Modularity-Maximization Algorithm | | | | Louvain Algorithm | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Parameter(K)** | **Precision** | **Recall** | **F1-score** | **Time(s)** | **Precision** | **Recall** | **F1-score** | **Time(s)** | **Precision** | **Recall** | **F1-score** | **Time(s)** |
| **K = 0** | 0.54 | 0.89 | 0.57 | 3.33 | 0.58 | 0.90 | 0.61 | 1.30 | 0.58 | 0.90 | 0.61 | 0.52 |
| **K = 0.1** | 0.63 | 0.90 | 0.66 | 3.09 | 0.65 | 0.90 | 0.68 | 0.67 | 0.66 | 0.91 | 0.68 | 0.24 |
| **K = 0.2** | 0.69 | 0.89 | 0.71 | 3.81 | 0.69 | 0.90 | 0.72 | 1.06 | 0.69 | 0.90 | 0.72 | 0.27 |
| **K = 0.3** | 0.73 | 0.91 | 0.75 | 2.86 | 0.73 | 0.91 | 0.75 | 0.99 | 0.73 | 0.91 | 0.75 | 0.21 |
| **K = 0.4** | 0.80 | 0.91 | 0.81 | 4.29 | 0.80 | 0.92 | 0.81 | 0.55 | 0.80 | 0.92 | 0.81 | 0.20 |
| **K = 0.5** | 0.85 | 0.92 | 0.85 | 3.33 | 0.86 | 0.92 | 0.85 | 0.65 | 0.86 | 0.92 | 0.85 | 0.18 |
| **K = 0.6** | 0.93 | 0.91 | 0.90 | 4.05 | 0.93 | 0.91 | 0.90 | 0.37 | 0.93 | 0.91 | 0.90 | 0.19 |
| **K = 0.7** | 0.93 | 0.87 | 0.86 | 5.00 | 0.93 | 0.87 | 0.86 | 0.38 | 0.93 | 0.87 | 0.86 | 0.19 |
| **K = 0.8** | 0.93 | 0.80 | 0.82 | 3.33 | 0.94 | 0.80 | 0.82 | 0.32 | 0.94 | 0.79 | 0.82 | 0.10 |
| **K = 0.9** | 0.93 | 0.77 | 0.80 | 3.09 | 0.93 | 0.77 | 0.80 | 0.16 | 0.93 | 0.77 | 0.80 | 0.06 |
| **K = 1.0** | 0.92 | 0.72 | 0.76 | 3.81 | 0.92 | 0.73 | 0.76 | 0.15 | 0.92 | 0.73 | 0.76 | 0.07 |
| ***Avg.*** | *0.81* | *0.86* | *0.77* | *3.63* | *0.81* | *0.87* | *0.78* | *0.59* | *0.81* | *0.87* | *0.78* | *0.20* |

exhibits remarkable performance, achieving a precision of 0.93, a recall of 0.91, and an F1-score of 0.90 at $k = 0.6$. On average, $C^3$ consistently achieves high performance with precision and recall scores exceeding 0.8. Even at the poorest parameter setting of $k = 0$, $C^3$ still demonstrates a high recall of 0.90 and an F1-score exceeding 0.6. These results validate the effectiveness of $C^3$ in identifying cloned components.

2) **From a time-cost perspective, $C^3$ detects component-level clones with high efficiency.** As depicted in Table 1, we can observe a clear decrease in the running time of $C^3$ as the value of $k$ increases. $C^3$ requires 4.97 seconds to obtain detection results when $k = 0$, which represents the highest time cost among different settings. Conversely, the lowest time cost is only 0.8 seconds when $k = 1$. Additionally, when the optimal value of $k$ (i.e., $k = 0.6$) is used, $C^3$ requires only 2.56 seconds to provide the detection results. Based on the above findings, it is evident that $C^3$ is a highly efficient tool for component-level clone detection, as it is capable of delivering detection results within mere few seconds.

3) **The precision and recall performances demonstrate divergent trends with changes in the parameter $k$.** As presented in Table 1, the recall performance gradually declines from 0.9 to 0.73 as the value of $k$ increases. This trend can be attributed to the formation of a richer structure among components when $k$ has a lower value, which leads to a higher recall rate. However, it should be noted that some edges in this complex structure are created based on a low similarity between component pairs. As $k$ increases, numerous component pairs remain unconnected in the graph, causing the used detection algorithm to overlook many clones, resulting in a decrease in recall. Different from the recall, the precision exhibits an upward trend as $k$ varies from 0 to 1. This is because that a higher value of $k$ ensures that the components connected in the graph exhibit a higher similarity to others. Consequently, the components identified by $C^3$ are more likely to be real clones, leading to higher precision values.

✍ **RQ1** ► In summary, $C^3$ demonstrates a strong ability to accurately identify cloned components, achieving an impressive F1-score of 0.90. Moreover, it efficiently generates groups of cloned components with minimal time cost, taking no more than five seconds. $C^3$ achieves its best performance when the parameter is set to the optimal value of 0.6. ◄

## 5.2 RQ2: Which community detection algorithm is optimal for $C^3$?

*5.2.1 Motivation.* We employ the community detection algorithm to identify cloned components from the graph structure. To determine the most suitable algorithm for $C^3$, in this RQ, we conduct a comparison experiment to evaluate the performance of $C^3$ when using different community detection algorithms.

*5.2.2 Method.* In this RQ, we utilize three widely used community detection algorithms, namely the *Louvain algorithm* [5], *Girvan–Newman algorithm* [6], and *Fast-Greedy Modularity Maximization algorithm* [7], to identify component-level clone groups from the constructed graph structure, respectively. We compare their performance under different $k$ settings to determine the most effective algorithm for $C^3$.

*5.2.3 Metrics.* Similar to RQ1, we employ the same set of metrics to evaluate the performance of $C^3$ when using different community detection algorithms, including *precision*, *recall*, *F1-score*, and *running time (Time)*. Note that in this context, the running time refers specifically to the time required for the algorithm's execution phase.

*5.2.4 Results.* Our experimental results are illustrated in Table 2. According to the results, we observe the following findings:

1) **The three algorithms achieve comparable performance in the precision, recall, and F1-score.** As shown in Table 2, on average, both the Girvan–Newman algorithm and the Fast-Greedy Modularity-Maximization algorithm achieve precision, recall, and F1-score values of 0.81, 0.87, and 0.78, respectively. While, the Girvan–Newman algorithm shows a slightly lower recall performance of 0.86 compared to the other two algorithms, with a decrease of 0.1. Furthermore, all three algorithms demonstrate their best performance when the parameter $k$ is set to 0.6, achieving 0.90 in F1-score.

2) **The Girvan-Newman algorithm costs the longest running time, while the Louvain algorithm incurs the least amount of time, making it the most suitable algorithm for $C^3$.** In Table 2, it can be observed that the Girvan-Newman algorithm has the longest running time, ranging from 3s to 5s, to obtain the results. On average, its time cost exceeds that of the second algorithm by more than six times, and it is ten times higher than that of the last algorithm. In contrast, the Louvain algorithm only requires an average of 0.2s to obtain the clone results. Even when $k = 0$, the Louvain algorithm only takes 0.52s, which is still lower than the time cost of other algorithms when $k = 0$. Hence, the Louvain algorithm exhibits the lowest time cost among these three algorithms. Considering that

the three algorithms exhibit comparable performance and the Louvain algorithm has the lowest running time, the Louvain algorithm emerges as the most optimal choice for $C^3$.

3) **The Girvan-Newman algorithm demonstrates poor performance in both classification accuracy and time cost at low $k$ values.** Although the three algorithms achieve similar overall classification performance, we observe that the Girvan-Newman algorithm performs worse than the other two algorithms when the value of $k$ is low. Specifically, compared to the other two algorithms, the Girvan-Newman algorithm exhibits a decrease of 0.4 in precision and F1-score when $k = 0$. Furthermore, when $k = 0.1$ and $k = 0.2$, the precision and recall performance only reach 0.65 and 0.89, respectively, which is 0.2 and 0.1 lower than the average performance of the other two algorithms. In terms of time cost, the Girvan-Newman algorithm exhibits the longest execution time to obtain the clone results. Therefore, the Girvan-Newman algorithm is the least suitable choice for $C^3$.

✍ **RQ2** ► In summary, taking into account both classification accuracy and time cost, the Louvain algorithm stands out as the optimal choice for $C^3$ in detecting cloned components. Conversely, the Girvan-Newman algorithm is deemed the least suitable option for $C^3$. ◄

## 5.3 RQ3: How effective is our assessment method?

*5.3.1 Motivation.* In this RQ, our objective is to verify the effectiveness and usefulness of the assessment method and the six proposed clone features in measuring the refactoring necessity of cloned components.

*5.3.2 Method.* To achieve our goal, we conduct a user study to validate the correctness of the assessment method and the usefulness of the six clone features. To ensure the accuracy of the evaluation and mitigate human error, we randomly choose 100 cloned components. Based on these selected components, we create ten distinct surveys, each containing information about ten cloned components. This information includes the names of the components, as well as the values of the six features for the involved components. Note that the cloned components in different surveys are different.

We invited 30 experienced developers, each with over five years of experience in C language development, to participate in the user study. Leveraging their expertise and knowledge, they assess the refactoring necessity of cloned components that the survey contains. To further minimize human error, three of the 30 respondents are grouped together and assigned the same survey. In our survey, they were required to rank the cloned components to describe the level of the refactoring necessity of the cloned components based on the feature values of the components as well as their own working experience. In addition, we encouraged respondents to suggest additional features that they believe can serve as assessment metrics but are not included in our survey.

We validate the accuracy of the ranking results obtained from our method by comparing them with the responses provided by the participants. Since each survey is assigned to three respondents simultaneously, we consider each individual response as the ground truth ranking result and then compare this with the ranking result generated by our assessment method to determine the correctness of our method. Therefore, the accuracy of our assessment method for each different survey is calculated by averaging the results obtained from the three surveys that are identical to it, providing an overall measure of our method's performance.

*5.3.3 Metrics.* Three well-known evaluation metrics (i.e., *MAP@N Recall@N*, and *NDCG@N*) in recommendation tasks [65] are used in this RQ. MAP calculates the average of the Average Precision (AP) for the performance of our assessment method. *Recall@N* (Recall at N) represents the proportion of correctly identified cloned components among the top-N ranking results. *NDCG@N* (Normalized Discounted Cumulative Gain at N) evaluates the effectiveness of our method based on the relevance and ranking of cloned components. It considers the positions of relevant components in the ranked list and assigns higher scores to cloned components that are both relevant and ranked higher. The *NDCG@N* value is normalized to a range of 0 to 1, with 1 indicating the highest level of relevance and ranking accuracy. In our study, $N$ is set to 3, 5, and 7, respectively.

*5.3.4 Results.* Our user study results are presented in Table 3. Based on these results, we can draw the following observations:

1) **Our assessment method effectively prioritizes cloned components with a high refactoring necessity.** Our assessment method achieves high performance with 97.4% and 92.8% in terms of MAP@N and AR@N, respectively, validating the effectiveness of our method. Specifically, the majority of evaluation results exceed 90% in all metrics. Furthermore, 7 out of 10 surveys achieve a perfect score of 100% in both MAP@3 and AR@3, indicating a strong agreement between our ranking results and the responses from participants. Compared to MAP@N and AR@N, the NDCG@N metric specifically emphasizes the correctness of the ranking order in the results. Our method demonstrates excellent performance in NDCG@N, achieving an average NDCG@N score of 98.6%. This high score indicates the accuracy and correctness of the ranking order provided by our method.

2) **The six clone features prove to be valuable in assessing the refactoring necessity of cloned components.** Given the high performance of our method across all three metrics, it provides strong evidence supporting the value of the six clone features in assessing the refactoring necessity of cloned components. This is evident because if the proposed features were ineffective for assessment, the ranking results provided by our method would significantly deviate from the respondents' replies. Therefore, the consistent and accurate performance of our method serves as an indication of the usefulness of the six features.

**Other potential features.** Out of all the responses received, only two respondents suggest the inclusion of new features in our assessment

**Table 3: The performance of our assessment method.**

| **Survey** | **MAP@N(%)** | | | **Recall@N(%)** | | | **NDCG@N(%)** | | |
|---|---|---|---|---|---|---|---|---|---|
| | **3** | **5** | **7** | **3** | **5** | **7** | **3** | **5** | **7** |
| S1 | 100 | 94.9 | 100 | 100 | 86.7 | 100 | 100 | 98.8 | 99.7 |
| S2 | 100 | 95.9 | 100 | 100 | 80 | 95.2 | 99.8 | 98.3 | 99.7 |
| S3 | 100 | 89.5 | 90.8 | 100 | 73.3 | 90.4 | 100 | 95.6 | 96.0 |
| S4 | 100 | 100 | 97.7 | 100 | 100 | 85.7 | 100 | 100 | 97.5 |
| S5 | 97.2 | 95.1 | 98.6 | 88.9 | 80 | 95.2 | 98.5 | 97.9 | 98.4 |
| S6 | 100 | 100 | 95.7 | 100 | 100 | 81.0 | 100 | 99.8 | 97.4 |
| S7 | 92.8 | 100 | 100 | 77.8 | 100 | 100 | 97.2 | 99.4 | 99.4 |
| S8 | 100 | 97.8 | 98.9 | 100 | 86.7 | 95.2 | 100 | 97.8 | 99.3 |
| S9 | 100 | 100 | 100 | 100 | 100 | 100 | 97.9 | 98.2 | 96.7 |
| S10 | 95.6 | 96.7 | 85.7 | 88.9 | 93.3 | 85.7 | 99.2 | 98.8 | 96.9 |
| **Avg.** | 98.6 | 97.0 | 96.7 | 95.6 | 90 | 92.8 | 99.3 | 98.5 | 98.1 |
| **MAvg.** | 97.4 | | | 92.8 | | | 98.6 | | |

method. One respondent proposes considering the dependencies of a cloned component as a metric during the assessment. After careful deliberation, we acknowledge the usefulness of this metric in assessing the necessity of components for refactoring. A cloned component with complex dependencies in software, being called by many other code units or calling many other code units, signifies its significance to the software. However, we note that this metric bears a strong resemblance to one of the proposed metrics, namely F2: NCL (Number of Calls), which also captures the calling relationships among components. On the other hand, another respondent suggests incorporating the intellectual property rights of cloned components as an assessment metric. However, after thorough consideration, we determine that this feature does not provide substantial guidance in selecting cloned components with a high refactoring necessity. As a result, we decide not to include the suggested features in our component-level clone feature set. The results demonstrate that the selection of the six features is comprehensive overall.

✍ **RQ3** ► In summary, our assessment method effectively prioritizes cloned components with a high refactoring necessity by utilizing the six comprehensive and useful features. ◄

## 6 THREATS TO VALIDITY

**Internal validity:** When selecting the appropriate detector for identifying file-level clones, we carefully considered a variety of tools, such as RtvNN [57], Deckard [30], ASTNN [63], SCDetector [58], DeepSim [66], TreeCen [29], and even large language models (LLMs) [25, 27]. This rigorous selection process is aimed at ensuring the effectiveness and advancement of $C^3$. However, we observe that the majority of existing tools are primarily designed to detect function-level clones, while LLMs cannot exhibit satisfactory performance in identifying duplicated components due to limitations in training data. Hence, we select SourcererCC [51] as the most suitable option. Furthermore, we explore the use of other graph-based clustering algorithms to identify cloned components. However, we observe that most of these algorithms generate clusters based on factors such as distance, density, or hierarchical structures among nodes, rather than considering the structural relationships among nodes. As a result, these algorithms are not well-suited for application in our study. To ensure the accuracy of our code implementation, we conduct comprehensive testing of our approach's source code during the automatic evaluation phase.

**External validity:** Despite evaluating $C^3$ solely on an industrial-scale product, this product consists of billions of lines of code and $C^3$ achieves impressive performance on this dataset. This result suggests the potential for $C^3$'s generalizability and practicality to other codebases. To mitigate the potential threat of biased metrics, we consider two broad categories of performance metrics, each encompassing three distinct metrics.

**Construct validity:** The manual examination may introduce human errors, including evaluators' level of attentiveness and subjectivity. To address these concerns, we adopt two strategies. Firstly, we design multiple distinct surveys to minimize the impact of individual biases. Secondly, we hire highly experienced software practitioners who possess a minimum of five years of experience in developing software. Moreover, our dataset primarily consists of textual clones, with functional clones representing only a small portion. While this distribution may favor the performance of $C^3$, it aligns with the typical ratio of textual clones to semantic clones found in real-world software systems.

## 7 RELATED WORK

We summarise related studies focusing on code clone detection at different clone granularities, which can be classified into three categories [44, 47, 56]: statement/line-level, function/method-level, and file-level clone detection method. Specifically, some studies propose line-level code clone detectors [16–18, 23, 30–35, 37]. Johnson et al. [31] propose a clone detector, which is a pioneer in textual-based clone detection and leverages a fingerprinting technique to find out line-level code clones. NICAD [45, 46], as a text-based detector, can effectively identify line-level clones, and the full NICAD [22] also uses flexible code normalization and filtering techniques for removing the small differences between code fragments and thus identifies Type-3 clones. Most of the proposed clone detectors identify clones at the function granularity [14, 24, 38, 48, 50, 52, 55, 57, 58, 67]. Su et al. [55] propose an In-Vivo Clone Detection technique named HitoshiIO to detect functional cloned fragments (i.e., methods) or functions in arbitrary programs. Compared with HitoshiIO, Mathew et al. [38] present an improved function-level code clone detector called SLACC for identifying cross-language clones by using the dynamic analysis method. Wu et al. [58] propose another new functional clone detection approach named SCDetector, by combining the static program analysis technique and the knowledge of the social network. Only a few studies focused on identifying cloned files [13, 54, 57]. Singh et al. [54] present a hybrid approach, which combines text-based and metric-based analysis of programs, for the detection of structural cloned files, where structural cloned files denote the files which consist of lower-level smaller clones with similar code fragments. Akram et al. [13] propose a novel clone detector at file-level granularity by leveraging the index-based features extraction technique (IBFET).

## 8 CONCLUSIONS AND FUTURE WORK

In this paper, we propose $C^3$, a novel clone detector designed to identify cloned components in software systems, aiming to reduce development and maintenance costs. To optimize the elimination process of cloned components, we propose six component-level clone features and introduce an assessment method that assists developers in prioritizing components based on their refactoring necessity. To validate the effectiveness of $C^3$, we apply it to a large-scale industrial product and evaluate its performance in detecting cloned components. Our experimental results demonstrate that $C^3$ is highly effective in identifying cloned components from software, achieving an impressive F1-score of 0.9. Furthermore, we conduct a user study to validate the accuracy of our assessment method and assess the practicality of the six clone features. Our user study results confirm the accuracy of our method, highlighting the significant role played by the six proposed features in the assessment process. In the future, our aim is to expand $C^3$'s utility by bolstering its ability to identify cloned components across diverse programming languages.

## ACKNOWLEDGMENTS

This research is partially supported by the Fundamental Research Funds for the Central Universities (No. 226-2022-00064), the National Natural Science Foundation of China (No. 62141222 and No.

62202419), the Ningbo Natural Science Foundation (No. 2022J184), ARC Laureate Fellowship FL190100035, and the National Research Foundation, Singapore, under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

## REFERENCES

[1] 2023. https://github.com/alienjun/AJWaveRefresh/blob/master/AJWaveRefresh/AJWaveRefreshHeader.m
[2] 2023. https://github.com/GentleForYou/ShoppingDetailsDemo/blob/ae415c1456e35c297772be8b92980e5f67242aad/ShoppingDetailsDemo/MJDIYAutoFooter.m
[3] 2023. https://github.com/torvalds/linux/tree/master/arch/arm/include/asm/vdso
[4] 2023. https://cmake.org
[5] 2023. https://en.wikipedia.org/wiki/Louvain_method
[6] 2023. https://en.wikipedia.org/wiki/Girvan%E2%80%93Newman_algorithm
[7] 2023. https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.community.modularity_max.greedy_modularity_communities.html
[8] 2023. Component-based software engineering. https://en.wikipedia.org/wiki/Component-based_software_engineering
[9] 2023. Example vdso in Linux. https://github.com/torvalds/linux/tree/master/arch/arm64/include/asm/vdso
[10] 2023. Example vdso in Linux. https://github.com/torvalds/linux/tree/master/arch/arm/include/asm/vdso
[11] 2023. Reinventing the wheel. https://en.wikipedia.org/wiki/Reinventing_the_wheel
[12] 2023. what-counts-as-reinventing-the-wheel. https://softwareengineering.stackexchange.com/questions/55378/what-counts-as-reinventing-the-wheel
[13] Junaid Akram, Majid Mumtaz, and Ping Luo. 2020. IBFET: Index-based features extraction technique for scalable code clone detection at file level granularity. *Software: Practice and Experience* 50, 1 (2020), 22–46.
[14] Manar H Alalfi, James R Cordy, Thomas R Dean, Matthew Stephan, and Andrew Stevenson. 2012. Near-miss model clone detection for Simulink models. In *2012 6th International Workshop on Software Clones (IWSC)*. IEEE, 78–79.
[15] Hakam W Alomari and Matthew Stephan. 2020. srcClone: Detecting Code Clones via Decompositional Slicing. In *Proceedings of the 28th International Conference on Program Comprehension*. 274–284.
[16] Brenda S Baker. 1995. On finding duplication and near-duplication in large software systems. In *Proceedings of 2nd Working Conference on Reverse Engineering*. IEEE, 86–95.
[17] Hamid Abdul Basit and Stan Jarzabek. 2007. Efficient token based clone detection with flexible tokenization. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 513–516.
[18] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 368–377.
[19] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment* 2008, 10 (2008), P10008.
[20] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative research in psychology* 3, 2 (2006), 77–101.
[21] Robert Cloutier, Gerrit Muller, Dinesh Verma, Roshanak Nilchiani, Eirik Hole, and Mary Bone. 2010. The concept of reference architectures. *Systems Engineering* 13, 1 (2010), 14–27.
[22] James R Cordy and Chanchal K Roy. 2011. The NiCad clone detector. In *2011 IEEE 19th International Conference on Program Comprehension*. IEEE, 219–220.
[23] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. 1999. A language independent approach for detecting duplicated code. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. IEEE, 109–118.
[24] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. 2020. Functional code clone detection with syntax and semantics fusion learning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 516–527.
[25] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).
[26] Santo Fortunato. 2010. Community detection in graphs. *Physics reports* 486, 3-5 (2010), 75–174.
[27] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).
[28] Aakanshi Gupta and Bharti Suri. 2018. A survey on code clone, its behavior and applications. In *Networking Communication and Data Knowledge Engineering*. Springer, 27–39.
[29] Yutao Hu, Deqing Zou, Junru Peng, Yueming Wu, Junjie Shan, and Hai Jin. 2022. TreeCen: Building Tree Graph for Scalable Semantic Code Clone Detection. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.
[30] 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.
[31] J Howard Johnson. 1994. Substring Matching for Clone Detection and Change Tracking.. In *ICSM*, Vol. 94. 120–126.
[32] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.
[33] Raghavan Komondoor and Susan Horwitz. 2001. Using slicing to identify duplication in source code. In *International static analysis symposium*. Springer, 40–56.
[34] Jens Krinke. 2001. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*. IEEE, 301–309.
[35] Seunghak Lee and Iryoung Jeong. 2005. SDD: high performance code clone detection system for large scale source code. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 140–141.
[36] Yun Lin, Xin Peng, Zhenchang Xing, Diwen Zheng, and Wenyun Zhao. 2015. Clone-based and interactive recommendation for modifying pasted code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 520–531.
[37] Andrian Marcus and Jonathan I Maletic. 2001. Identification of high-level concept clones in source code. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. IEEE, 107–114.
[38] George Mathew, Chris Parnin, and Kathryn T Stolee. 2020. SLACC: simion-based language agnostic code clones. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 210–221.
[39] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica* 22, 3 (2012), 276–282.
[40] Kawser Wazed Nafi, Tonny Shekha Kar, Banani Roy, Chanchal K Roy, and Kevin A Schneider. 2019. CLCDSA: cross Language code clone detection using syntactical features and API documentation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1026–1037.
[41] Mathieu Nayrolles and Abdelwahab Hamou-Lhadj. 2018. CLEVER: combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 153–164.
[42] 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.
[43] Daniel Perez and Shigeru Chiba. 2019. Cross-language clone detection by learning over abstract syntax trees. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 518–528.
[44] 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.
[45] 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.
[46] Chanchal K Roy and James R Cordy. 2008. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *2008 16th iEEE international conference on program comprehension*. IEEE, 172–181.
[47] 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.
[48] Tobias Sager, Abraham Bernstein, Martin Pinzger, and Christoph Kiefer. 2006. Detecting similar Java classes using tree algorithms. In *Proceedings of the 2006 international workshop on Mining software repositories*. 65–71.
[49] Neha Saini, Sukhdip Singh, et al. 2018. Code clones: Detection and management. *Procedia computer science* 132 (2018), 718–727.
[50] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V Lopes. 2018. Oreo: Detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 354–365.

[51] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*. 1157–1168.

[52] Abdullah Sheneamer and Jugal Kalita. 2015. Code clone detection using coarse and fine-grained hybrid approaches. In *2015 IEEE seventh international conference on intelligent computing and information systems (ICICIS)*. IEEE, 472–480.

[53] Abdullah Sheneamer and Jugal Kalita. 2016. A survey of software clone detection techniques. *International Journal of Computer Applications* 137, 10 (2016), 1–21.

[54] Manu Singh and Vidushi Sharma. 2015. Detection of file level clone for high level cloning. *Procedia Computer Science* 57 (2015), 915–922.

[55] Fang-Hsiang Su, Jonathan Bell, Gail Kaiser, and Simha Sethumadhavan. 2016. Identifying functionally similar code in complex codebases. In *2016 ieee 24th international conference on program comprehension (icpc)*. IEEE, 1–10.

[56] Jeffrey Svajlenko and Chanchal K Roy. 2020. A Survey on the Evaluation of Clone Detection Performance and Benchmarking. *arXiv preprint arXiv:2006.15682* (2020).

[57] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 87–98.

[58] Yueming Wu, Deqing Zou, Shihan Dou, Siru Yang, Wei Yang, Feng Cheng, Hong Liang, and Hai Jin. 2020. SCDetector: Software Functional Clone Detection Based on Semantic Tokens Analysis. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 821–833.

[59] Yanming Yang, Xin Xia, David Lo, Tingting Bi, John Grundy, and Xiaohu Yang. 2022. Predictive models in software engineering: Challenges and opportunities. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–72.

[60] Yanming Yang, Xin Xia, David Lo, and John Grundy. 2022. A survey on deep learning for software engineering. *ACM Computing Surveys (CSUR)* 54, 10s (2022), 1–73.

[61] Iqra Yaqub and Khubaib Amjad Alam. 2020. Code Clone Detection: A Systematic Review. *KIET Journal of Computing and Information Sciences* 3, 1 (2020), 16–16.

[62] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. 2019. Neural detection of semantic code clones via tree-based convolution. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 70–80.

[63] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.

[64] Liang-Jie Zhang and Jia Zhang. 2010. SOA reference architecture. In *Web Services Research for Emerging Applications: Discoveries and Trends*. IGI Global, 1–15.

[65] Neng Zhang, Qiao Huang, Xin Xia, Ying Zou, David Lo, and Zhenchang Xing. 2020. Chatbot4qr: Interactive query refinement for technical question retrieval. *IEEE Transactions on Software Engineering* (2020).

[66] Gang Zhao and Jeff Huang. 2018. Deepsim: deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 141–151.

[67] Yue Zou, Bihuan Ban, Yinxing Xue, and Yun Xu. 2020. CCGraph: a PDG-based code clone detector with approximate graph matching. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 931–942.

Received 2023-05-18; accepted 2023-07-31