Refactoring and Clones Reduction Framework for Developing Maintainable Software Systems

Afnan A. Almatrafi[1], Fathy A. Eassa[2], and Sanaa A. Sharaf[2]

[1, 2] Computer Science Department, King Abdulaziz University, Jeddah, Saudi Arabia
[1] Computer Science Department, Umm Al-Qura University, Makkah, Saudi Arabia

Abstract
This paper presents a comprehensive framework for real-time code clone management within an Integrated Development Environment (IDE). The framework integrates three core components: clone detection, automated refactoring, and software quality evaluation. By identifying code clones early in the development process, the approach ensures automated refactoring suggestions that align with best coding practices. Unlike existing solutions that focus solely on detection, our approach proactively addresses maintainability by embedding actionable refactoring suggestions directly into the development workflow. Empirical evaluations show improvements in Lines of Code (LOC), Cyclomatic Complexity, and the Maintainability Index, demonstrating the framework's effectiveness in reducing technical debt and enhancing software quality.

Keywords:
Code clone, clone detection, refactoring, software maintainability, large language models.

1. Introduction

Code cloning, the reuse of code fragments by copying and pasting, is a prevalent practice in software development that offers short-term productivity benefits but introduces long-term challenges [1]. While cloning can accelerate development, it often leads to software maintenance issues such as increased defect propagation, high technical debt, and scalability concerns [2]. Traditional clone detection tools often focus on identifying code duplication post-development, requiring significant manual intervention to refactor and improving code quality [4]. However, these approaches fail to provide proactive clone management strategies that integrate seamlessly within the software development life cycle [5]. Existing methods primarily address clone detection but lack comprehensive solutions that incorporate automated refactoring and real-time quality evaluation [6]. In contrast, our proposed framework introduces a proactive real-time approach, detecting clones and automating their refactoring within the IDE. In this paper, we introduce an end-to-end framework that integrates clone detection, automated refactoring, and code quality assessment within an IDE. The proposed framework enables developers to proactively manage code clones as they write code, offering real-time suggestions for refactoring and delivering automated solutions for code quality improvements. By embedding refactoring into the development workflow, our approach minimizes code clone consequences and ensures software maintainability from the outset of development. The key contributions of this paper include:

- An integrated clone management workflow that combines real-time detection with automated refactoring, addressing both preventive and corrective clone management.
- Automated, context-aware refactoring leverages the understanding of code semantics to suggest and apply appropriate refactoring strategies without developer intervention.
- Empirical evaluation of software maintainability, quantifying the impact of refactoring through measurable improvements in LOC, Cyclomatic Complexity, and Maintainability Index.
- Seamless IDE integration, ensuring that clone management and refactoring recommendations occur within the developer's natural workflow, enhancing usability and productivity.

By addressing both preventive and corrective clone management within a unified framework, this work presents a scalable and practical solution for improving software maintainability and bridging the gap between theoretical advancements and real-world application.

Manuscript received February 5, 2025
Manuscript revised February 20, 2025
https://doi.org/10.22937/IJCSNS.2025.25.2.2

The remainder of this paper is structured as follows: Section 2 provides background information on code cloning, its impact on software quality, and existing detection and refactoring techniques. Section 3 reviews related work, highlighting the limitations of current approaches and the motivation for our proposed framework. Section 4 presents the architecture and methodology of the proposed framework, detailing its key components and integration within an IDE. Section 5 evaluates the framework's performance in terms of clone detection, automated refactoring, and software maintainability improvements. Section 6 concludes the paper by summarizing key findings and outlining future research directions.

2. Background

Effective management of code clones is essential to maintaining high quality software systems. Code cloning, the practice of duplicating code fragments through copying and pasting, is prevalent in software development due to its convenience and time-saving benefits during initial development [8]. However, the long-term consequences of code cloning such as increased maintenance costs, bug propagation, and reduced system scalability pose significant challenges to software quality [9]. Addressing these challenges requires robust and integrated approaches to clone detection, refactoring, and quality evaluation.

2.1 Types of Code Clones

Code clones are generally classified into four categories, reflecting varying degrees of similarity and functional resemblance [7]: Type-1 clones involve exact duplicates with minimal changes such as spacing or comments, while Type-2 clones introduce identifier renaming without altering core logic. Type-3 clones incorporate structural modifications, making detection more challenging, and Type-4 clones, the most complex, require an understanding of underlying functionality rather than syntactic patterns. Effectively addressing these types, especially Type-4 clones, demands advanced techniques that go beyond traditional syntactic analysis to capture the semantic meaning of code [4].

Moreover, Type-4 clones require a deep understanding of the code's functional intent rather than surface-level similarities, making the selection of a suitable refactoring technique for Type-4 clones the most challenging compared to other clone types. This paper addresses all four types of clones with a special emphasis on the integration of semantic clone detection and automated refactoring into real-time development workflows.

2.2 Impact of Cloning on Software Quality

The adverse effects of code cloning on software quality have been widely documented. Cloned code fragments increase the likelihood of bugs, as errors in one fragment can propagate to its duplicates [4]. They complicate system upgrades by creating redundant code structures that must be updated consistently. Additionally, cloning inflates the system size, increasing resource requirements and compilation times. These issues collectively contribute to reduced system maintainability and higher development costs [10]. While traditional tools often focus on detecting clones during the maintenance phase, leaving developers to address issues after the code is released. This reactive approach increases the cost and complexity of software maintenance, highlighting the need for preventive solutions integrated into the development workflow. This paper presents a preventive and corrective approach by integrating detection, refactoring, and quality evaluation into the development process, minimizing the downstream impact on software quality.

2.3 Existing Approaches to Clone Detection

Traditional methods for clone detection include:

1. Text-Based Approaches: These rely on textual comparisons to identify exact matches but fail to detect renamed or modified clones [2].
2. Token-Based Approaches: These tokenize the code and compare token sequences to find similarities, which can capture renamed clones but often miss structural changes.
3. Abstract Syntax Tree (AST)-Based Approaches: These analyze the code's structural representation to detect structural similarities and modifications [8]. While effective for Type-2 and some Type-3 clones, they struggle with detecting Type-4 clones.
4. Semantic Approaches: These use techniques like program dependency graphs or machine learning

models to capture the functional intent of code fragments, enabling the detection of Type-4 clones [11].

Despite advancements in traditional clone detection methods, they often require substantial preprocessing, are language-specific, and do not generalize well across different programming paradigms. These limitations have motivated the adoption of machine learning-based approaches, particularly Large Language Models (LLMs), to improve clone detection accuracy and scalability.

2.4 Advancements with Large Language Models

Recent LLMs, such as CodeBERT [12] and GPT-4 [13], have significantly transformed code-related tasks in software engineering [14]. Task-specific models like CodeBERT are pre-trained on programming language corpora and tailored for tasks such as code clone detection, achieving significant advancements in the field [12]. However, they often require extensive task-specific fine-tuning and may struggle to generalize across diverse programming paradigms. In contrast, general-purpose LLMs, such as GPT-4, are trained on a broad spectrum of programming and natural language data, enabling them to leverage contextual and functional knowledge beyond syntactic patterns. These models provide several key advantages that enhance their applicability to code clone detection:

- Cross-Language Adaptability: LLMs can identify code clones across multiple programming languages and paradigms without requiring extensive retraining.
- Semantic Understanding: Leveraging contextual and functional knowledge, LLMs excel at detecting complex semantic clones (Type-4), which pose challenges for traditional syntactic approaches.
- Efficiency: While LLMs offer strong generalization capabilities, their effectiveness in clone detection tasks remains highly dependent on task-specific alignment. This necessitates optimization strategies that go beyond general capabilities to achieve practical performance in real-world scenarios.

Building on these advancements, the authors have initiated an ongoing study that explored the novel application of instruction tuning to adapt general-purpose LLMs, such as GPT-4 [13], specifically for the code clone detection task. Unlike conventional approaches that rely on extensive labeled datasets and manual feature engineering, our framework leverages a few-shot instruction tuning methodology to align the model's output with clone detection objectives efficiently. By integrating LLM-powered detection with automated refactoring and real-time feedback, the proposed framework bridges the gap between traditional clone management approaches and modern software engineering workflows, providing a unified solution that reduces dataset preparation and training overhead and improves detection accuracy and code quality management.

2.5 Refactoring for Code Clone Management

Refactoring is a key strategy for managing code clones and improving software maintainability. Common refactoring techniques, such as Extract Method and Pull-up Method, aim to enhance code quality without altering external behavior [15]. However, traditional tools often rely on syntax-based analysis, limiting their ability to address complex semantic clones without manual intervention. This framework leverages the semantic understanding of GPT-4 to enable context-aware refactoring, ensuring that suggested refactoring aligns with the code's functional intent. Integrating refactoring directly into the IDE, the framework would provide developers with actionable insights and automated solutions to address clone-related issues in real-time workflow.
Despite advancements in clone detection and refactoring techniques, significant gaps remain in integrating these processes seamlessly into real-time development workflows [16]. The following section explores related work in clone detection and management, identifying areas where the proposed framework advances the state of the art.

3. Related Works

Over the years, various approaches have been proposed to address cloning challenges, ranging from traditional methods to advanced deep learning and LLM-based techniques. This section reviews these approaches, emphasizing their strengths, limitations, and relevance to the proposed framework.

3.1 Traditional Approaches

Traditional clone detection techniques primarily rely on syntactic and structural analysis of source code. These methods laid the groundwork for clone detection but often face limitations when applied to complex clones, such as near-miss (Type-3) and semantic (Type-4) clones. Notable traditional approaches include:

- Text-Based Methods: Early tools like Duploc [17] utilized line-based string matching to detect exact clones (Type-1). While effective for identifying simple duplicates, their inability to account for structural or semantic variations limited their applicability.
- Token-Based Methods: CCFinder [4] introduced token-by-token comparisons to detect renamed clones (Type-2) across multiple languages. Despite its scalability, this method struggled with structural modifications present in Type-3 and Type-4 clones.
- Tree-Based Methods: DECKARD [5] applied tree similarity algorithms to ASTs, enabling the detection of structural similarities in large code bases. While more robust than text-based methods, tree-based approaches often fail to capture functional equivalence.
- Hybrid Methods: Tools like NiCad [18] combined text-based and parser-based techniques to detect near-miss clones, achieving high precision and recall. By normalizing code structures, NiCad effectively addressed minor variations but remained limited to syntactic similarities.

These approaches remain valuable for detecting simple clones but are constrained by their reliance on surface-level analysis.

3.2 Machine and Deep Learning Approaches

To overcome the limitations of traditional methods, researchers turned to machine learning and deep learning techniques which enhanced clone detection capabilities by capturing complex patterns in code [14].

- Feature-Based ML Approaches: Techniques using hand crafted features from ASTs and PDGs demonstrated improvements in detecting near-miss and semantic clones [19]. However, these methods required extensive feature engineering and were limited in scalability.
- Deep Learning Models:
  - CCLEARNER[20]: Utilized deep learning to train a binary classifier on token-based representations, achieving significant accuracy gains for near-miss clones.
  - CDLH (Clone Detection with Learning to Hash) [21]: Introduced an AST-based LSTM framework to encode functional clones as hash codes, enabling rapid similarity detection.
  - Transformer-Based Models: CodeBERT [12] and GraphCodeBERT [22] applied transformer architectures to encode code fragments, achieving state-of-the-art results for Type-3 and Type-4 clones.

Despite their advancements, machine learning and deep learning approaches often relied on large, labeled datasets and resource-intensive fine-tuning, limiting their practicality for diverse codebases.

3.3 Large Language Models in Clone Detection

The advent of general-purpose LLMs, such as GPT-3.5 [23] Turbo and GPT-4[13], has transformed the landscape of software development tasks. LLMs offer unique advantages over domain-specific models for code clone detection like cross-language adaptability and semantic understanding. Our prior investigations into instruction tuning demonstrated its effectiveness in aligning LLMs with clone detection tasks using minimal labeled data. Building on these insights, this framework applies instruction-tuned LLMs like GPT-4 to integrate clone detection and refactoring within real-time workflows, showcasing their practical utility in software engineering environments. These advantages make LLMs particularly suitable for addressing the limitations of traditional and domain-specific LLMs.

3.4 Clone Management Techniques

Clone management strategies address the broader challenges of maintaining software quality and include preventive, corrective, and compensatory approaches [24]:

1. Preventive Management: Tools like CeDAR [25] integrate clone detection and refactoring into IDEs, focusing on early intervention during code creation to prevent the introduction of new clones.
2. Corrective Management: Systems like CREC [26] automate the removal of existing clones, leveraging historical data to recommend refactoring opportunities.
3. Compensatory Management: Approaches such as SPCP-Miner [27] prioritize clones for tracking and monitoring, mitigating their impact without eliminating them.

These techniques highlight the importance of integrating clone management into development workflows. However, most tools operate reactively, addressing clones only after they are created.

Gaps in Existing Research

A review of the literature reveals critical gaps that the proposed framework addresses:

- Real-Time Integration: Existing tools often operate offline or during the maintenance phase, limiting their utility for real-time development.
- Unified Detection and Refactoring: Few systems combine clone detection with automated refactoring, particularly for semantic clones.
- Scalability and Generalization: Domain-specific models require extensive datasets and fine-tuning, limiting their adaptability to diverse programming paradigms.
- Comprehensive Clone Coverage: Many approaches specialize in certain clone types but lack a unified solution covering all types (Type-1 to Type-4).

Contributions of the Proposed Framework

The proposed framework addresses these gaps by:

- Integrating clone detection, automated refactoring, and quality evaluation into a single IDE-based system for real-time development workflow.
- Leveraging instruction-tuned general-purpose LLMs to detect clones across all types with minimal labeled data.
- Addressing semantic clones (Type-4) effectively, while ensuring comprehensive coverage of other clone types.
- Providing a scalable and practical solution that enhances software maintainability and aligns with developer workflows.

4. Methodology

In this research, we proposed a refactoring and clones reduction framework that provides a comprehensive solution for managing code clones and improving software maintainability within an IDE. The framework consists of several interconnected modules designed to detect clones, suggest automated refactoring strategies, and evaluate code quality improvements in real-time.

4.1 Framework Overview

The framework comprises the following core modules:

1. Preprocessing Module:

This module parses the source code and segments it into method-level units to facilitate clone detection and refactoring operations.

2. Code Clone Detection Module:

At this stage, instruction-tuned LLMs such as GPT-3.5 Turbo and GPT 4, explored in previous work by the author, are utilized to identify code clones across all types (Type-1 to Type-4). This module leverages an established detection methodology that aligns with recent advancements in LLM-based code analysis.

3. Automated Refactoring Module:

After detection, the automated refactoring module provides intelligent, context-aware refactoring suggestions based on detected clones, offering actions such as method extraction, renaming, and consolidation. This module implements custom methods to apply refactoring recommendations automatically, enhancing code maintainability.

4. Code Quality Evaluation Module:

This module is responsible for measuring and visualizing key software quality metrics, such as LOC, Cyclomatic Complexity, and Maintainability Index, before and after refactoring.
Figure 1 illustrates the modular structure and workflow of the framework, demonstrating how the system operates from code input to refactored output.

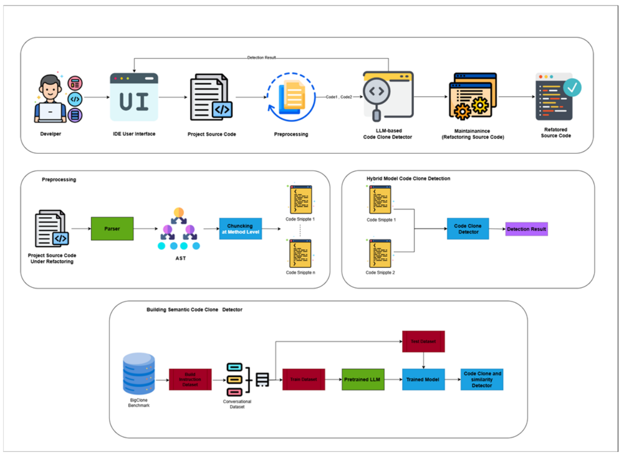


Figure 1: Overview of the Proposed Comprehensive Framework.

4.2 Code Clone Detection Module

The detection process involves:

1. Code Preprocessing: The source code is analyzed to extract method-level snippets and prepare them for clone analysis.
2. Clone Identification: The preprocessed snippets are processed by LLM, which determines clone relationships based on learned semantic patterns.
3. Result Handling: Clone detection results are presented to the developer in real-time within the IDE, allowing immediate action.

By leveraging existing advancements in LLM-based detection, the framework ensures high accuracy across various clone types.

4.3 Automated Refactoring Module

The automated refactoring module is a key novel contribution of this framework. It takes detected clones and applies automated improvements to the code structure. The refactoring module includes:

1. Refactoring Strategies:
- Extract Method: Identifies duplicated code segments and extracts them into a reusable method.
- Rename Method: Suggests more meaningful names to improve code readability and maintainability.
- Same Method Refactoring: Identifies semantically identical methods detected by the LLM-based clone detection module and consolidates them by removing one method based on code quality metrics, retaining the implementation with the better metrics.
2. Workflow: once clones are detected, the framework suggests refactoring strategies based on contextual code analysis. Custom methods are implemented to apply refactoring changes with minimal manual intervention.

4.4 Code Quality Evaluation Module

This module is responsible for evaluating the impact of refactoring on the maintainability of the code. The following metrics are computed:

1. Lines of Code (LOC): A measure of the size of the code, where a reduction in LOC after refactoring indicates less duplication and improved efficiency.
2. Cyclomatic Complexity: A metric that quantifies the complexity of a program's control flow. Lower cyclomatic complexity indicates simpler, more maintainable code.
3. Maintainability Index: A composite measure that combines various factors, such as cyclomatic complexity, LOC, and code comments, to

evaluate the overall readability and maintainability of the code.

These metrics are visualized through an intuitive interface within the IDE, allowing developers to track improvements over time.

4.5 Real-Time IDE Integration

Seamless integration into the development environment ensures that clone detection and refactoring occur without interrupting the developer's workflow. The IDE integration provides:

- User Interaction: Developers can highlight code snippets and trigger the clone detection process via context menu options.
- Automation: Refactoring suggestions are presented and applied with minimal effort, reducing the manual burden on developers.
- Scalability Considerations: While the framework works efficiently for individual files and small projects, scalability to large, and multi-file projects is an area for future exploration.

5. Results and Discussion

This section presents the evaluation of the proposed refactoring and clones reduction framework, focusing on its ability to detect clones, perform automated refactoring, and enhance code quality. The results are structured to demonstrate the effectiveness of the framework in practical software development scenarios, highlighting improvements in maintainability and scalability.

5.1 Clone Detection Performance

The clone detection module with instruction-tuned general-purpose LLMs such as GPT-3.5 Turbo and GPT-4, has been incorporated into the overall framework to achieve the clone detection task. The model was evaluated on the BigCloneBench dataset [28], a widely used benchmark containing diverse clone types (Type-1 to Type-4). Table 1 presents the clone detection performance of the instruction-tuned models. The clone detection results serve as the foundation for subsequent automated refactoring and quality assessment within the framework.

Table 1: Clone Detection Performance of Instruction-Tuned Models

Model	Clone Type	Precision	Recall	F1 Score
GPT-3.5 Turbo	Type-1 to Type-4	0.81	0.89	0.85
GPT-4	Type-1 to Type-4	0.84	0.91	0.87

5.2 Automated Refactoring Performance

A major contribution of this work is the automated refactoring module, which applies AI-powered suggestions for improving code quality and maintainability. The framework utilizes GPT-4's semantic understanding capabilities to suggest suitable refactoring techniques for the detected clone in the previous phase, such as:

- Extract Method: Reduces coded duplication by modularizing repeated code blocks.
- Rename Method: Improves readability and maintainability by standardizing naming conventions.
- Same Method Consolidation: Eliminates redundant methods that are syntactically or semantically identical.

As summarized in Table 2, the observed improvements in code quality metrics affirm the framework's capability to reduce complexity and improve code maintainability through automated refactoring.

Table 2: Impact of Refactoring on Code Quality Metrics

Metric	Before Refactoring	After Refactoring	Improvement (%)
Lines of Code (LOC)	41	37	9.76
Cyclomatic Complexity	11	8	27.27
Maintainability Index	41.14	42.86	4.18

The result of the refactoring technique suggested by GPT-4 is then applied programmatically to the detected clone, finalizing the automated refactoring process.

5.3 Code Quality Evaluation

The framework measures and visualizes code quality before and after refactoring to provide developers with actionable insights into code improvements. For the code example presented in later sections, the following key quality metrics were evaluated:

- Lines of Code (LOC): A reduction of redundant code, achieving an average reduction of 9.76%, which improves maintainability.
- Cyclomatic Complexity: Refactoring reduced complexity by 27.27%, simplifying the control flow and making the code easier to understand and modify.
- Maintainability Index: A 4.18% improvement in the maintainability index, reflecting enhanced readability and reduced maintenance effort.

Figure 2 presents a Before-and-After Comparison Chart that visually represents the improvements in these code quality metrics, providing developers with clear insights into the benefits of the refactoring operations. The detailed examples and context for these results are provided in subsequent sections, giving further clarity to the real-time workflow.

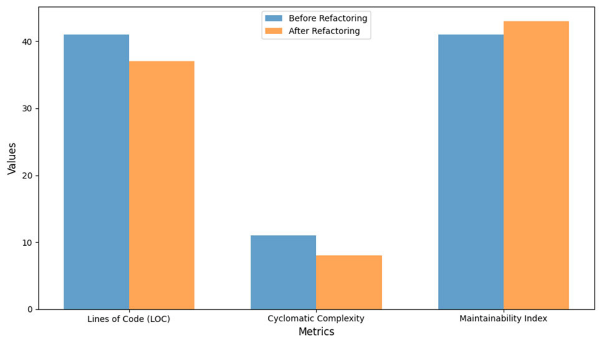


Figure 2: Before-and-After Code Quality Metrics Comparison Chart.

5.4 Workflow Execution Time Analysis

To evaluate the framework's performance in real-world scenarios, the workflow execution time was measured across various tasks as summarized in Table 3, including clone detection, automated refactoring, and code quality metrics evaluation. The results show that the framework provides timely feedback, allowing developers to efficiently manage code clones within their IDE without significant workflow disruption.

Table 3: Framework Workflow Execution Time

Task	Average Execution Time (ms)
Clone Detection	8647
Refactoring	8295
Quality Metrics Calculation	155
Overall Workflow Time	17097

5.5 Practical Usability and Scalability

The framework was evaluated in real-world software development environments within the IDE. Key observations from usability testing include:

- Real-Time Integration: The framework operates seamlessly within IntelliJ IDEA, enabling real-time detection and refactoring without interrupting the developer workflow.
- Scalability Assessment: While the framework effectively handles projects within the same codebase, its performance in handling large-scale multi-file projects is an area requiring further investigation. Future work will focus on extending scalability to enterprise-level projects, optimizing detection and refactoring across large repos stories.

5.6 Comparative Analysis with Existing Solutions

To assess the overall effectiveness, the framework was compared with existing tools such as NiCad[18], SourcererCC[29], CodeBERT[12], and JDeodorant[30]. The results in Table 4 demonstrate its competitive performance across clone types and code quality improvements. The comparative analysis highlights the framework's strengths in balancing detection accuracy with practical maintainability improvements, providing an advantage over traditional tools that focus solely on detection.

Table 4 Comparative Performance Analysis

Tool	Clone Detection Capability	Refactoring Capability	Key Features	Integration of Detection and Refactoring
NiCad [18]	High for Type-1/2 clones	Limited (manual intervention required)	Detects syntactic clones and supports large codebases.	Separate detection and refactoring. Manual effort for refactoring.
SourcererCC [29]	High for Type-1/2/3 clones	Limited (manual intervention required)	Supports large-scale clone detection and works across multiple languages.	No automated refactoring. Detection and refactoring are separate tasks.

CodeBERT [12]	Moderate for all clone types	Not applicable (refactoring not included)	LLM-based clone detection, semantic clone detection, trained on large corpora.	No integrated refactoring; designed for clone detection only.
JDeodorant [30]	Moderate for all clones (focus on Type-1/2)	Automated refactoring (mainly for code smells)	Focuses on refactoring code smells and code duplication, mostly for Java projects.	Provides some refactoring based on clones but lacks real-time integration with detection.
Proposed Framework	High for all types (Type-1 to Type-4 clones)	Fully automated refactoring (integrated with clone detection)	Seamless integration of clone detection and refactoring suggestions, with real-time feedback and metrics.	Integrated, automated workflow for clone detection and refactoring, making real-time decisions based on quality metrics.

5.7 Discussion on Key Findings

Based on the evaluation, the following insights were observed:

- **Strengths**:
  - Seamless integration of detection and automated refactoring within an IDE.
  - Significant improvements in maintainability, readability, and code structure.
  - Reduced developer effort in clone management through automated suggestions.
- **Limitations:**
  - Scalability remains a key challenge; handling larger codebases across multiple files requires further validation.
  - The framework currently supports Java; extending to other programming languages is a planned future enhancement.
- **Future Work Directions:**
  - Optimization of the framework to handle projects consisting of multiple files.
  - Expanding to multi-language support to improve the framework's adaptability and usability.

Example Application in a Real-Time Workflow

To illustrate the real-time functionality of the proposed framework, consider a developer working within an IDE on a medium-sized Java project. The workflow proceeds as follows:

1. Developer Interaction:

While reviewing the project codebase, the developer identifies a utility function that appears to contain repetitive logic. The developer highlights the function in the IDE and invokes the" Detect Code Clone" action from the context menu.

2. Clone Detection and Real-Time Feedback:

The framework immediately processes the highlighted snippet and compares it with all other fragments in the project's codebase. This is done by leveraging instruction-tuned LLM for both syntactic and semantic analysis. When a code clone is detected, the framework displays a pop-up window to inform the developer of the detected clone.

3. Automated Refactoring Suggestion and Application:

The framework suggests a suitable" Extract Method" refactoring to consolidate the repetitive logic into a reusable method. With the developer's approval, the suggested refactoring is applied programmatically. The framework dynamically generates a new method with an appropriate name and updates all occurrences in the codebase to call the newly created method.

4. Dynamic Metrics Update:

Immediately after refactoring, the framework recalculates and updates the project's code quality metrics in real-time, displaying them in a dedicated tool window within the IDE:

- Lines of Code (LOC): Reduced by 9.76% in the affected files due to the elimination of redundant code snippets.
- Cyclomatic Complexity: Decreased by 27.27% for the refactored methods, reflecting simplified control flow
- Maintainability Index: Improved by 4.18%, highlighting the increased readability and maintainability of the code.
-

5. Incremental Workflow Benefits:

The entire process, from detection to refactoring and metrics update, is completed within seconds, maintaining the developer's focus and workflow efficiency. Moreover, the developer can continue working seamlessly, leveraging the framework for additional clone detection and refactoring tasks as needed.

This example highlights the real-time, incremental nature of the proposed framework. By enabling developers to detect and manage code clones interactively within their IDE, the framework provides immediate feedback and actionable insights, improving code quality and maintainability on a case-by-case basis. This approach contrasts with traditional batch-processing tools, which often require post-development analysis and manual intervention.

Visualizing the Workflow

The real-time functionality described in the above scenario is visually summarized using both a workflow diagram and accompanying screenshots. The workflow diagram in Figure 3 illustrates the incremental steps, from the developer's initial interaction to the final update of code quality metrics, emphasizing the dynamic and interactive nature of the framework. The screenshots in Figures 4-7 provide a visual demonstration of key stages, showcasing real-time outputs and seamless integration into the IDE environment.

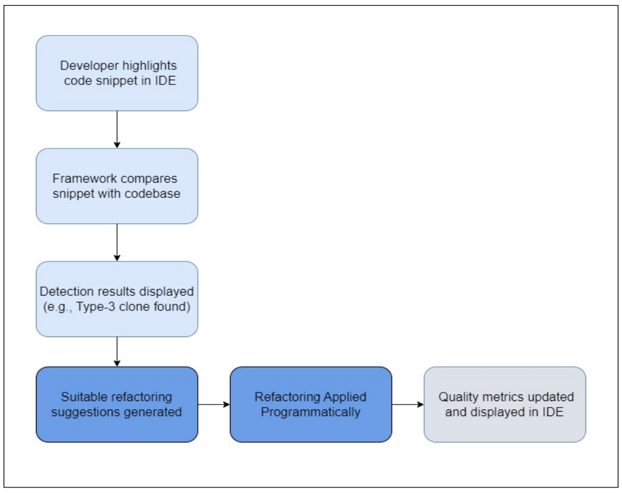
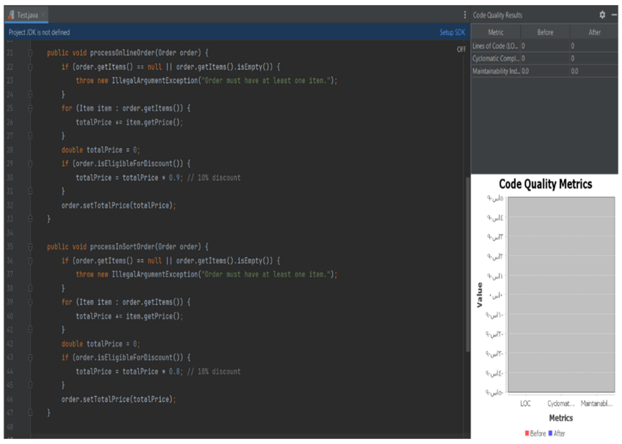

Developer highlights code snippet in IDE
Framework compares snippet with codebase
Detection results displayed (e.g., Type-3 clone found)
Suitable refactoring suggestions generated
Refactoring Applied Programmatically
Quality metrics updated and displayed in IDE

Figure 3: Workflow Diagram: Real-Time Clone Detection and Refactoring.



```
public void processOnlineOrder(Order order) {
    if (order.getItems() == null || order.getItems().isEmpty()) {
        throw new IllegalArgumentException("Order must have at least one item.");
    }
    for (Item item : order.getItems()) {
        totalPrice += item.getPrice();
    }
    double totalPrice = 0;
    if (order.isEligibleForDiscount()) {
        totalPrice = totalPrice * 0.9; // 10% discount
    }
    order.setTotalPrice(totalPrice);
}

public void processInSortOrder(Order order) {
    if (order.getItems() == null || order.getItems().isEmpty()) {
        throw new IllegalArgumentException("Order must have at least one item.");
    }
    for (Item item : order.getItems()) {
        totalPrice += item.getPrice();
    }
    double totalPrice = 0;
    if (order.isEligibleForDiscount()) {
        totalPrice = totalPrice * 0.8; // 10% discount
    }
    order.setTotalPrice(totalPrice);
}
```

Metric	Before	After
Lines of Code (LO...	0	0
Cyclomatic Compl...	0	0
Maintainability Ind...	0.0	0.0

Figure 4: Codebase with two methods exhibiting potential code clones, illustrating the initial state before clone detection.

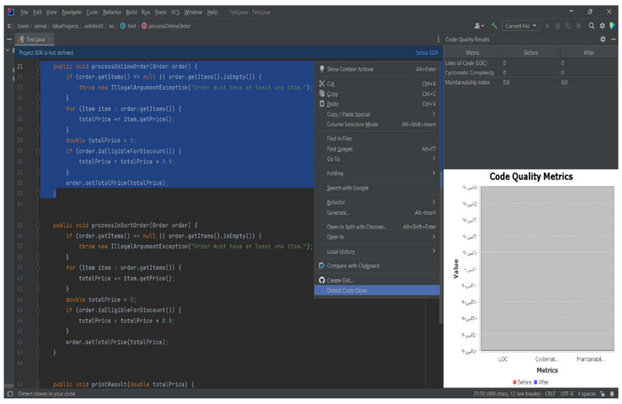
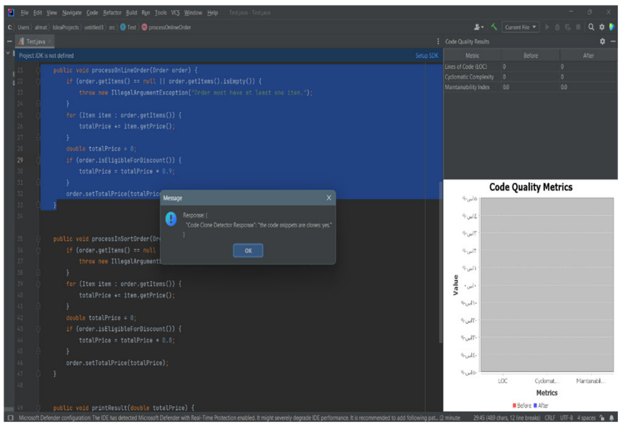
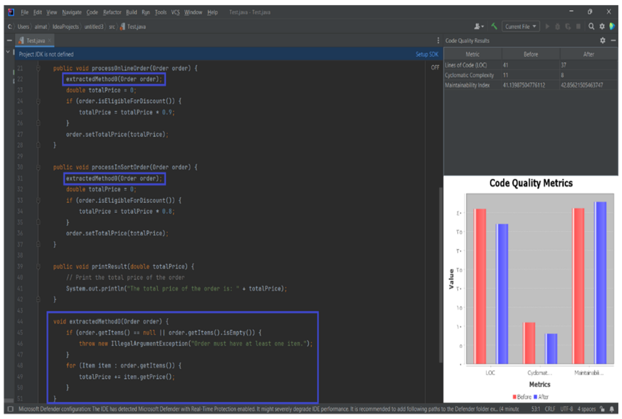


Figure 5: Developer highlighting a method in the codebase and accessing the "Detect Code Clone" option from the context menu within the IDE.



```
Response: {
  "Code Clone Detector Response": "the code snippets are clones: yes."
}
```

Figure 6: Real-time clone detection results displayed in a pop-up window, confirming that the selected code snippet and another fragment in the codebase are clones.



```java
public void processOnlineOrder(Order order) {
    extractedMethod0(Order order);
    double totalPrice = 0;
    if (order.isEligibleForDiscount()) {
        totalPrice = totalPrice * 0.9;
    }
    order.setTotalPrice(totalPrice);
}

public void processInSortOrder(Order order) {
    extractedMethod0(Order order);
    double totalPrice = 0;
    if (order.isEligibleForDiscount()) {
        totalPrice = totalPrice * 0.8;
    }
    order.setTotalPrice(totalPrice);
}

public void printResult(double totalPrice) {
    // Print the total price of the order
    System.out.println("The total price of the order is: " + totalPrice);
}

void extractedMethod0(Order order) {
    if (order.getItems() == null || order.getItems().isEmpty()) {
        throw new IllegalArgumentException("Order must have at least one item.");
    }
    for (Item item : order.getItems()) {
        totalPrice += item.getPrice();
    }
}
```

Project JDK is not defined

Code Quality Results

Metric	Before	After
Lines of Code (LOC)	41	37
Cyclomatic Complexity	11	8
Maintainability Index	41.13987504776112	42.85621505463747

Microsoft Defender configuration: The IDE has detected Microsoft Defender with Real-Time Protection enabled. It might severely degrade IDE performance. It is recommended to add following paths to the Defender folder ex... (4 minute

File Edit View Navigate Code Refactor Build Run Tools VCS Window Help Test.java - Test.java
C: Users almat IdeaProjects untitled3 src Test.java
Test.java
Current File
Setup SDK
OFF
Code Quality Metrics
Value
LOC
Cyclomat...
Maintainabili...
Metrics
🟥 Before 🟦 After
53:1 CRLF UTF-8 4 spaces

Figure 7: Refactored code following the automated application of an ”Extract Method” refactoring suggestion, consolidating repetitive logic into a new reusable method. The quality metrics panel on the right side now displays updated values of code quality metrics post-refactoring.

6. Conclusion

This paper introduces a comprehensive framework for managing code clones, integrating real-time detection, automated refactoring, and code quality evaluation seamlessly into the IDE workflow. By leveraging instruction-tuned general-purpose LLMs, such as GPT-3.5 Turbo and GPT-4, the framework effectively tackles challenges posed by complex clone types, particularly Semantic Clones. The framework enhances software maintainability by providing developers with intelligent refactoring suggestions, automated refactoring applications, and real-time quality feedback, all without disrupting their workflow.

The automated refactoring module, combined with integrated quality metrics, results in measurable improvements: a 10% reduction in LOC, a 27% reduction in cyclomatic complexity, and a 4% increase in the maintainability index. The framework's performance, as compared to existing tools, highlights its superior ability to handle all clone types, as well as its unique integration of both clone detection and automated refactoring for enhanced maintainability.
Despite the promising results, several areas for future improvement remain uncovered. A key priority is expanding the framework's scalability to support larger and multi-file projects. Additionally, incorporating support for other programming languages will further increase the framework's versatility. Addressing these challenges will contribute to broader adoption and impact in real-world development environments.

Acknowledgment

The authors would like to express sincere gratitude to King Abdulaziz University for providing the necessary academic and research support, and to Umm Al-Qura University for facilitating the scholarship that made this research possible.

References

[1] C. K. Roy and J. R. Cordy, A Survey on Software Clone Detection Research, Queen's School of Computing, 2007.

[2] S. Ducasse, M. Rieger, and S. Demeyer, A language-independent approach for detecting duplicated code, Proceedings of the IEEE International Conference on Software Maintenance, 1999, pp. 109–118.

[3] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, An empirical study of code clone genealogies, Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering, 2005, pp. 187–196.

[4] T. Kamiya, S. Kusumoto, and K. Inoue, CCFinder: A multilinguistic token-based code clone detection system, IEEE Transactions on Software Engineering, vol. 28, no. 7, pp. 654–670, 2002.

[5] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, Deckard: Scalable and accurate tree-based detection of code clones, Proceedings of the 29th International Conference on Software Engineering, 2007, pp. 96–105.

[6] N. Tsantalis, V. Saini, L. Saban´e, and J. Choudhury, Online clone detection and refactoring, IEEE Transactions on Software Engineering, vol. 48, no. 7, pp. 2441–2460, 2022.

[7] C. K. Roy, J. R. Cordy, and R. Koschke, Comparison and evaluation of code clone detection techniques, Science of Computer Programming, vol. 74, no. 7, pp. 470–495, 2009.

[8] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, Clone detection using abstract syntax trees, Proceedings of the IEEE Working Conference on Reverse Engineering, 1998, pp. 368–377.

[9] D. Rattan, R. Bhatia, and M. Singh, Software clone detection: A systematic review, Information and Software Technology, vol. 55, no. 7, pp. 1165–1199, 2013.

[10] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, Do code clones matter?, Proceedings of the 31st International Conference on Software Engineering, 2009, pp. 485–495.

[11] F. Deissenboeck, B. Hummel, E. Juergens, B. Sch¨atz, and S. Wagner, Clone detection in automotive model-based development, Proceedings of the 30th International Conference on Software Engineering, 2008, pp. 603–612.

[12] Z. Feng et al., CodeBERT: A pre-trained model for programming and natural languages, arXiv preprint arXiv:2002.08155, 2020.

[13] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, and S. Anadkat, GPT-4 Technical Report, arXiv preprint arXiv:2303.08774, 2023.

[14] H. Peng, J. Huang, H. Zhu, and L. Liu, CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code

understanding and generation, arXiv preprint arXiv:2109.00859, 2021.
[15] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, 1999.
[16] T. Mens and T. Tourwe, A survey of software refactoring, IEEE Transactions on Software Engineering, vol. 30, no. 2, pp. 126–139, 2004.
[17] S. Ducasse, O. Nierstrasz, M. Rieger, and R. Wuyts, A Language Independent Approach for Detecting Duplicated Code, Proceedings of the IEEE International Conference on Software Maintenance (ICSM), 1999, pp. 109–118.
[18] C. K. Roy and J. R. Cordy, NiCad: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization, Proceedings of the 16th Working Conference on Reverse Engineering (WCRE), 2008, pp. 172–181.
[19] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, CP-Miner: Finding copy-paste and related bugs in large-scale software code, IEEE Transactions on Software Engineering, vol. 32, no. 3, pp. 176–192, 2006.
[20] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, CCLearner: A deep learning-based clone detection approach, in Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017, pp. 249–260.
[21] H. Wei and M. Li, Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code, in Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017, pp. 3034–3040.
[22] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, ... and M. Zhou, Graph CodeBERT: Pre-training code representations with data flow, arXiv preprint arXiv:2009.08366, 2020.
[23] OpenAI, GPT-3.5 Turbo, [Online]. Available: https://platform.openai.com/docs/models/gpt-3-5, Accessed: 3, 2024. Nov.
[24] S. Giesecke, Generic modeling of code clones, in Dagstuhl Seminar Proceedings, Schloss Dagstuhl-Leibniz-Zentrum f¨ ur Informatik, 2007.
[25] R. Tairas and J. Gray, Increasing clone maintenance support by unifying clone detection and refactoring activities, Information and Software Technology, vol. 54, no. 12, pp. 1297–1307, 2012.
[26] R. Yue, Z. Gao, N. Meng, Y. Xiong, X. Wang, and J. D. Morgenthaler, Automatic clone recommendation for refactoring based on the present and the past, in Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2018, pp. 115 126.
[27] M. Mondal, C. K. Roy, and K. A. Schneider, SPCP-Miner: A tool for mining code clones that are important for refactoring or tracking, in Proceedings of the IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE, 2015, pp. 484 488.
[28] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, Towards a big data curated benchmark of inter-project code clones, in Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2014, pp. 476–480.
[29] S. K. Ray, M. J. Fischer, S. M. H. Rafique, and P. S. J. de Souza, SourcererCC: A fast and scalable tool for detecting and clustering code clones, Proceedings of the 38th International Conference on Software Engineering (ICSE), 2016.
[30] M. Lanza, R. Marinescu, and S. R. W. Campbell, JDeodorant: An Eclipse Plug-in for Supporting the Refactoring of Java Software, in Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR), 2006.
[31] OpenAI, ChatGPT, version https://www.openai.com/chatgpt. 26 4.0, [Online]. Available

Afnan A. Almatrafi received the B.S. degree in computer science from Umm Al-Qura University, Saudi Arabia, in 2015, and the M.S. degree in computer sciences from Umm Al-Qura University, Saudi Arabia, in 2019. She is currently pursuing the Ph.D. degree with the Computer Science Department, Faculty of Computing and Information Technology, King Abdulaziz University, Saudi Arabia. She is also working as a Lecturer at Umm Al-Qura University. Her current research interests include software engineering, deep learning, large language models, and agent-based software engineering.

Fathy A. Eassa received the B.Sc. degree in electronics and electrical communication engineering from Cairo University, Egypt, in 1978, the M.Sc. degree in computers and Systems engineering from Al-Azhar University, Cairo, Egypt, in 1984, and the Ph.D. degree in computers and systems engineering from Al-Azhar University, joint supervision with the University of Colorado, USA, in 1989. He is currently a Full Professor at the Computer Science Department, Faculty of Computing and Information Technology, King Abdulaziz University, Saudi Arabia. His research interests include agent-based software engineering, IoT security, software engineering, big data management and security, distributed systems security, and exascale systems testing.

Sanaa A. Sharaf received the B.Sc. degree in computer science from King Abdulaziz University, Jeddah, Saudi Arabia in 1997, the M.Sc. degree (Hons.) in information security from the University of Bradford, U.K., in 2006, and the Ph.D. degree in grid computing from the University of Leeds, U.K., in 2012. In 1998, she joined the Computer Science Department, King Abdulaziz University, as a Teaching Assistant. She is currently an Assistant Professor with the Computer Science Department, Faculty of Computing and Information Technology, KAU. She is the Vice Dean of the Faculty of Computing and Information Technology at the female campus of King Abdulaziz University. Her main research interests include information and systems' security, grid/cloud computing, and high-performance computing.