frontiers | Frontiers in Computer Science

TYPE Original Research
PUBLISHED 24 April 2025
DOI 10.3389/fcomp.2025.1520344

OPEN ACCESS

EDITED BY
Cristian Iván Pinzón,
Technological University of Panama, Panama

REVIEWED BY
Jihyun Lee,
Jeonbuk National University, Republic of Korea
Cássio Leonardo Rodrigues,
Universidade Federal de Goiás, Brazil

*CORRESPONDENCE
Jose Navas-Su
✉ jnavas@tec.ac.cr
Antonio Gonzalez-Torres
✉ antonio.gonzalez@tec.ac.cr

RECEIVED 31 October 2024
ACCEPTED 03 April 2025
PUBLISHED 24 April 2025

CITATION
Navas-Su J, Gonzalez-Torres A and Hernandez-Castro F (2025) A visual clone analysis method.
Front. Comput. Sci. 7:1520344.
doi: 10.3389/fcomp.2025.1520344

A visual clone analysis method

Jose Navas-Su[1]*, Antonio Gonzalez-Torres[2]* and Franklin Hernandez-Castro[3]

[1]Computing School, Costa Rica Institute of Technology, Cartago, Costa Rica, [2]Computer Engineering School, Costa Rica Institute of Technology, Cartago, Costa Rica, [3]Industrial Design School, Costa Rica Institute of Technology, Cartago, Costa Rica

The process of creating high-quality software is always a significant challenge for developers. Besides acquiring and honing the trained skills and knowledge needed to fulfill this process successfully, they face the persistent need for better methods and tools to understand and analyze software systems. This research proposes a method for building metrics-based visual analytic tools fully integrated into the development environment to enhance the capabilities and convenience of these working environments. In particular, duplicate code metrics allow developers to locate and correct potential threats to software quality, understandability, maintainability, and scalability. Refactoring duplicate code should help avoid duplication of effort during maintenance and decrease the possibility of introducing errors and inconsistencies. The proposed approach is demonstrated by programming a plugin component that detects and allows the visual analysis of duplicate code in a working project. We developed this tool for a well-known integrated development environment.

KEYWORDS

software engineering, code clones, visual analysis, code metrics, understandability

1 Introduction

Code clones refer to fragments of source code that are identical or highly similar to each other. They are often categorized into different types, ranging from exact textual duplicates to purely semantic clones that share functionality but not syntax (Lei et al., 2022; Kaur and Rattan, 2023; Zakeri-Nasrabadi et al., 2023). Exact or Type 1 clones consist of identical code segments, where comments, layouts, and whitespace are ignored. Parameterized or Type 2 clones consist of Type 1 clones that may contain changes in identifiers, types, and literals. Near-miss or Type 3 clones are Type 2 clones that may include the addition or removal of statements. Semantic or Type 4 clones are code segments that are not syntactically similar but perform the same functionality.

Clone detection research continues to be crucial for software maintenance, evolution, and refactoring because unrefactored clones can propagate defects and increase technical debt over time (Kanwal et al., 2022; Sundelin et al., 2025). Recent advances suggest a trend toward using data-driven models and deep learning techniques to detect both syntactic and semantic clones with improved accuracy (Zhang and Saber, 2025b; Hu et al., 2023; Feng et al., 2024).

Clone detection approaches typically fall into one or more of the following categories: textual, lexical or token-based, tree-based, graph-based, and embedding-based (Li et al., 2023; Thaller et al., 2022; Wang et al., 2023; Xu et al., 2024). Textual techniques process the source code as a stream of text. Token-based methods parse source code into tokens and compare those token streams for near-duplicate segments. Tree-based approaches transform source code into Abstract Syntax Trees (ASTs) to capture syntactic structures more precisely. Graph-based methods employ control-flow or data-flow representations

to detect semantically similar code. Embedding-based techniques, including neural embeddings or transformer-based representations, leverage large datasets of code to learn generalizable features that are well suited to detecting near-miss or semantic clones (Yahya and Kim, 2023; Zhang and Saber, 2025a; Pinku et al., 2024).

Refactoring clone pairs or groups helps control maintenance complexity by unifying duplicate logic. In particular, refactoring operations such as the Extract Method can convert multiple clones into a single shared function, mitigating the risk of missing bug fixes or enhancements in one cloned location (AlOmar et al., 2024; Lei et al., 2022). However, clone refactoring can be challenging in large-scale systems, especially when clones occur across different modules, repositories, or even programming languages (Kanwal et al., 2022; Li et al., 2023). Cross-language clone detection and management present additional hurdles, as equivalent functionality may appear in distinct languages, each with its own APIs, idioms, and libraries (Yahya and Kim, 2023; Khajezade et al., 2024). Despite these challenges, recent work has begun to merge deep learning with cross-language embeddings to detect functional similarity at a higher level of abstraction (Alam et al., 2023; Zhang and Saber, 2025a).

The rise of transformer-based large language models (LLMs) that have been trained on extensive code corpora—such as GPT or CodeBERT—has opened new avenues for discovering non-trivial clones and recommending potential refactorings (Zhang and Saber, 2025b; Feng et al., 2024). Researchers are currently investigating how best to harness these models, for example, by fine-tuning them on labeled clone datasets to capture both syntactic and semantic relationships among code fragments (Xu et al., 2024; Pinku et al., 2024). Although LLMs show promise in capturing deeper semantic structures, they can also generate spurious matches or struggle with domain-specific libraries, highlighting the need for further investigation into model architectures and tokenization strategies (Khajezade et al., 2024; Assi et al., 2025).

Beyond detection, clone management is another essential area of study. Effective management involves ranking and tracking clones throughout the software lifecycle, visualizing their relationships, and prioritizing them for refactoring (Zakeri-Nasrabadi et al., 2023; Kaur and Rattan, 2023). Some tools now incorporate techniques like complexity metrics or usage frequency to automatically suggest which clones should be refactored first (Kanwal et al., 2022; Sundelin et al., 2025). In practice, decisions about whether to refactor or keep duplicated code can also depend on performance considerations, the developer's workflow, or domain-specific constraints.

Visual analysis is one emerging trend that can aid developers in spotting and managing clones at scale. By rendering clone groups or classes in an interactive graph or heatmap, developers can navigate clusters of related code and quickly pinpoint large or complex duplications (Zhang and Saber, 2025a; Zakeri-Nasrabadi et al., 2023). Furthermore, integrating clone detection and refactoring support directly into modern IDEs can facilitate a continuous clone management process, alerting developers whenever they introduce a new clone or significantly alter an existing one (AlOmar et al., 2024; Hu et al., 2023). This model of "just-in-time," clone awareness supports more proactive handling of code duplication, instead of relying on *post-hoc* analysis after technical debt has accrued.

In this study, we propose a visual clone management approach that builds upon these lines of research. By combining a hybrid detection technique—one that integrates textual, lexical, and tree-based features—with an interactive visualization layer, we aim to provide both robust detection of near-duplicate code and developer-friendly tools for prioritizing and refactoring high-impact clones. To validate our approach, we compare detection results against established benchmarks and demonstrate how visual insights in the IDE can streamline clone analysis and management. Our ultimate goal is to enhance maintainability by leveraging state-of-the-art code similarity insights while placing the developer at the center of the clone management loop (Wang et al., 2023; Pinku et al., 2024).

Finally, we emphasize the growing need for cross-project or cross-repository clone analysis, especially as organizations adopt microservices architectures or polyglot development practices. In that context, large-scale empirical evaluations with real-world codebases are crucial for demonstrating the feasibility and cost-effectiveness of advanced clone detection tools (Xu et al., 2024; Lei et al., 2022). By presenting our tool's design and empirical outcomes, we hope to further the conversation on how best to integrate clone detection and refactoring into everyday development workflows, mitigating risks associated with duplicated logic while helping developers maintain clear, consistent, and evolvable software (Alam et al., 2023; Khajezade et al., 2024).

The repository for the source code is available on GitHub under the account `jnavas-tec`, and the project is `vizclone`.

In the rest of this article, we first present the proposed method in Section 2, followed by the validation of our proposed Clone Detection Algorithm in Section 3. We present the results in Section 4, and Section 5 contains the discussion of the results.

2 Materials and methods

In this section, we present our proposed method for clone management, which consists of two main stages: a screening and detection stage described in Section 2.1, and a visualization and analysis stage described in Section 2.2. The diagram in Figure 1 depicts the complete steps of the proposed method.

2.1 Clones screening and detection

Our method detects code clones at the method granularity level, mainly because code clones are disseminated primarily across methods. However, even those clone classes with many cloned fragments across different methods represent a tiny fraction of a software system's total number of methods. Thus, comparing every method in the system against every other method is a time-wasting task. To avoid this, a fast and scalable algorithm for finding similar items performs a screening of candidate pairs of methods with a minimum expected probability of being code clones. This algorithm implements the technique of Near-Neighbor Search that combines the techniques of K-Shingles,

Minhash Signatures, and Locality-Sensitive Hashing, as described in Leskovec et al. (2020), and is a textual clone detection approach. We succinctly explain our implementation of their algorithm in Section 2.1.1. A Syntactic-Hierarchical Local-Global Sequence Alignment Algorithm verifies the set of candidate pairs of code clones returned by the screening algorithm. This verification algorithm combines a syntactic-hierarchical analysis of methods and their sentences with local and global sequence alignment, as shown in Section 2.1.2. The local sequence alignment-based sub-algorithm implements a lexical approach, while the global sequence alignment-based sub-algorithm implements a tree-based approach.

We use the IDE internals to syntactically collect and analyze all the methods in the project. The developer can choose whether to compare the actual instances of literals and identifiers (e.g., cloneList, 80.0, "a string") or to compare them by their token names (e.g., IDENTIFIER, FLOAT_LITERAL, STRING_LITERAL). The methods are then represented hierarchically as a sequence of sentences, with each sentence being a sequence of tokens.

2.1.1 Near-neighbor search screening algorithm

The problem of detecting code clones can be considered a subcase of finding similar documents from a large set of documents. In the context of document similarity, we employ the Jaccard similarity measure to determine how similar two documents are. The Jaccard similarity between two sets A and B is $|A \cap B|/|A \cup B|$.

Comments and extra white spaces are trimmed from the source code. The signature of the method is also ignored. The algorithm ignores the methods with fewer sentences than a minimum threshold of sentences or fewer tokens than a minimum threshold of tokens. The source code from the body of the methods is represented as a set of k-shingles to identify similar sequences in different methods. A shingle is a substring of length k extracted from the code. The following text "Sample_text" contains the following set of 5-shingles: {"Sampl", "ample", "mple_", "ple_t", "le_te", "e_tex", and "_text"}. We selected a value of k equal to 27 to keep a low probability that a shingle would appear in any method. The sentences and tokens of the methods are joined with whitespace and fed to the shingle extractor. All the shingles from all the methods are extracted and added to a list, and then their index replaces all the shingles in all the methods. The sets of shingle indices represent the methods, and we could obtain the similarity level between any two methods by calculating the Jaccard similarity of their two shingle index sets. However, doing so would leave us back at square one, comparing every pair of methods. Instead, the algorithm replaces all the shingle index sets from the methods with their Minhash signatures.

The algorithm calculates the Minhash signatures for all the methods by choosing the signature size n as the product of the number of b bands in the signature and the number of r rows per band. The number of bands and rows allows us to establish the minimum similarity level needed to screen any pair of methods as clone candidates. For each band, it generates a permutation of all the shingles' indices and selects each method's permuted first r shingles' indices as its signature part for the band.

The algorithm feeds the methods' Minhash signatures to a Locality-Sensitive Hashing (LSH) step. The LSH step takes b iterations over all the methods' Minhash signatures. The i-th iteration hashes the i-th band of each method's Minhash signature and collects the methods that are hashed to the same bucket as clone candidate pairs. The sets of buckets in each iteration are independent of each other.

Our implementation of LSH can detect clone candidate pairs with a similarity of at least s by choosing b and r such that $s = (1/b)^{1/r}$. For example, if we set the minimum similarity to $s = 0.8$ and the number of rows per band to $r = 18$, then the number of bands is $b = 56$, and the length of the Minhash signatures is $n = 1008$. On the one hand, to decrease the number of false negatives, we can adjust b and r to achieve a value of s lower than 0.8. On the other hand, to decrease the number of false positives, we can set b and r to produce s greater than 0.8.

2.1.2 Syntactic-hierarchical local-global sequence alignment algorithm

Once we have the list of clone candidate pairs, the algorithm must verify which are indeed code clones. Our clone verification algorithm takes every pair of candidates and performs a syntactic-hierarchical local-global sequence alignment step. Syntactic means leveraging the Abstract Syntax Tree (AST) constructed by the IDE's language compiler to extract each method and its tokenized sentences. Each method then consists of a syntactic hierarchy of lists: a list of sentences identified by their type (i.e., if-statement, while-statement, assignment-statement) and their sublists of tokens. Moreover, local-global alignment means applying the local sequence alignment algorithm from Smith and Waterman (1981) with the optimization by Gotoh (1982) at the sentence level while applying the global sequence alignment algorithm from Needleman and Wunsch (1970) at the token level.

The best local alignment found between the sentences of two clone candidate methods is flagged as a clone if the similitude of the alignment is at least the selected similitude threshold (i.e., 0.8 similitude) and if the alignment has at least a selected minimum number of sentences and tokens. The local alignment algorithm compares each of the sentences of a method against every other sentence of the other method. It considers them similar if they are of the same sentence type and if the global sequence alignment of their tokens' returned similitude is at least the selected similitude threshold. Otherwise, the algorithm treats them as a mismatch or introduces a gap, whichever yields the most similar alignment.

If the local alignment for a verified clone pair contains gaps or any mismatches in the alignment, then it recognizes it as a Type 3 clone. However, if it does not contain any gaps or mismatches and the source code of the sentences is identical, then it identifies it as a Type 1 clone. Otherwise, the algorithm detects it as a Type 2 clone.

Finally, the algorithm merges the verified clone pairs into clone classes or groups:

1. It collects all the fragments of the clone pairs in a list and sorts them by their owner method.
2. It identifies the fragments of the same method that overlap by at least a selected overlap ratio (i.e., 0.6) and merges their corresponding clone pairs into a clone class.
3. If a clone pair exists whose fragments match the other two fragments of the recently merged clone pairs, the

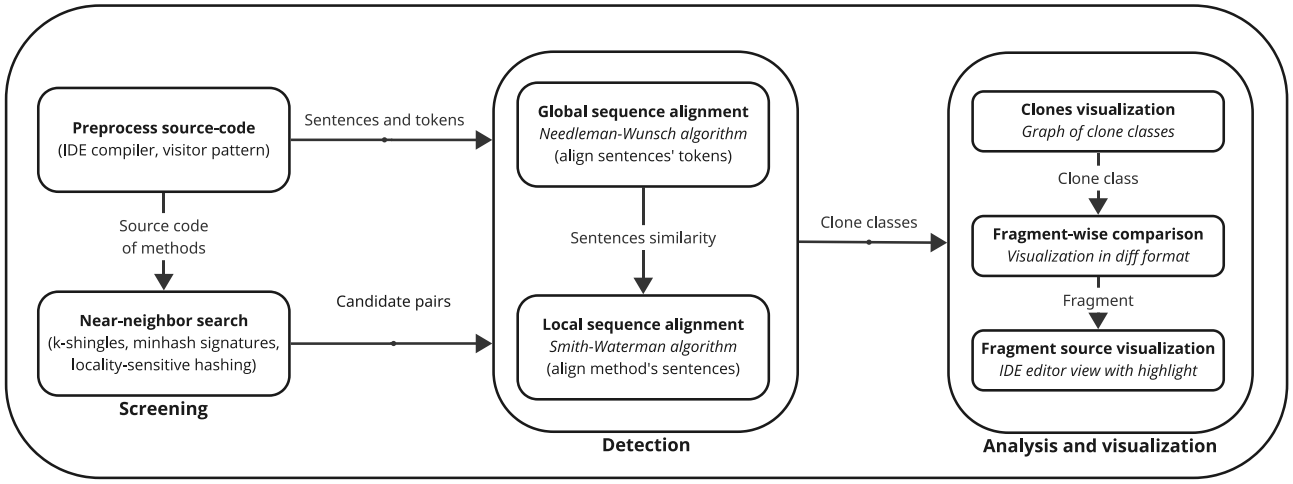

FIGURE 1
High-level illustration of the proposed method.

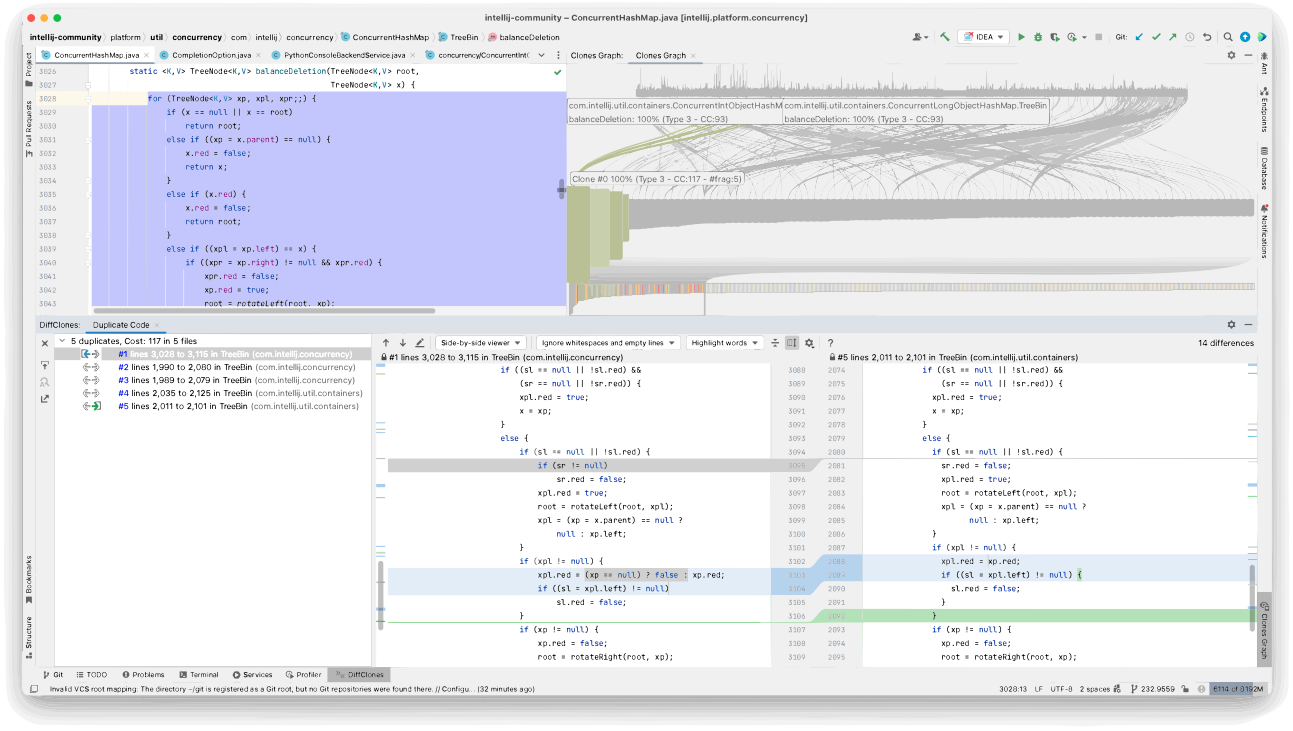

FIGURE 2
IDE view with all visualization components.

algorithm adds this clone pair to the clone class from the previous step.

2.2 Clones visualization and analysis

After screening and verifying the clone classes along with their clone fragments, the plugin shows an interactive visual representation of the graph assembled by the clones and the fragments in the IDE. As shown in Figure 2, the IDE includes the following:

1. A toolbar that features an interactive graphical visualization of all the clone classes.
2. A toolbar with the list of fragments from the selected clone class on the previous graph in.
3. An editor window shows the method containing the selected fragment from the fragments list in.

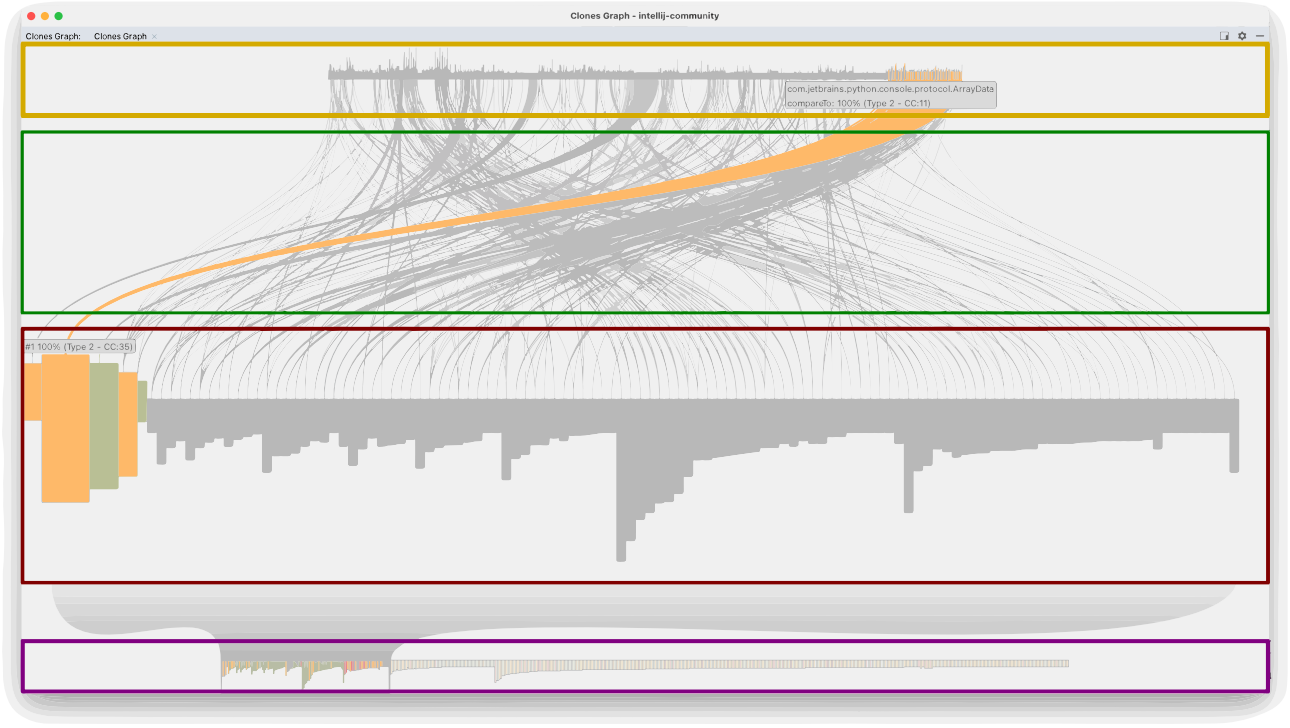

FIGURE 3
Clone classes visualization.

The region from Figure 2 is the interactive visualization of all the clone classes and shows all the clone groups and their code fragments. The visualization is shown in more detail in Figure 3, with its four subregions tagged in red:

1. A bar chart with all the clone classes found. It has a slider to zoom in on a subset of all the clone classes.
2. A zoomed-in subset of clone classes that fisheyes a hovered-over clone class and displays information about it (i.e., clone identifier, similarity percentage, clone type, cognitive complexity, and number of fragments).
3. A region with the arcs that connect the clone classes to their fragments.
4. A bar chart displays all the fragments from the clone classes. It allows users to hover over the fragments of the selected clone and shows its information (i.e., class signature, method name, similarity percentage, clone type, and cognitive complexity).

The clones visualized in Figure 3 belong to the code base of the JetBrains Open Source IntelliJ Community project on GitHub. The bar chart at the base of the visualization shows all the clone classes as bars in descending order by their number of fragments in the class and their cognitive complexity. The clone bar colors represent the clone type: red for Type 1, yellow for Type 2, and green for Type 3. The height of each clone bar is proportional to its cognitive complexity. The bar chart has a slider to select and enable zooming in on a subset of clone classes.

The visualization shows a zoomed-in subset of clone classes selected with the slider (see the bar chart in from Figure 3). When the developer selects or hovers over a particular clone class, the visualization displays a fisheye view of the neighboring clones and some clone information. It highlights the selected clone, its fragments, and the arcs that connect the clone class to its fragments while graying out the rest of the graph. The graph is not grayed when no class is selected, and the clone classes are sorted either by their Cognitive Complexity and then by their number of fragments or by their number of fragments and then by their Cognitive Complexity. Users can use the middle click to switch between both sort orders to make finding clones that need refactoring easier. If the developer clicks on a clone class, the highlight of its fragments remains visible until clicked elsewhere. If the developer right-clicks on a clone class (action on Figure 2), the list of its fragments is shown in the fragments toolbar (see the list in from Figure 2). The fragments toolbar allows users to select two fragments and show their differences using the diff format (action). We reused the fragments toolbar provided by the IntelliJ IDEA Community Edition IDE in their duplicate code search utility. If the developer double-clicks a fragment from the list (action), it opens the source code of that fragment in an editor from Figure 2.

The developers can analyze the clone classes to determine who needs refactoring. Depending on the development team, the maximum number of acceptable clone fragments in a clone class ranges from two to ten. We may choose five as an acceptable number of fragments in a clone class before labeling it for refactoring.

The clone classes with cognitive complexity over 15 for any of their fragments can also be labeled for refactoring, as they are considered too complex, according to the recommendation of the open-source platform SonarQube. Future research could provide

TABLE 1 Recall reported by BigCloneEval.

Clone type	NiCad6	VizClone
Type-1	0.99897	0.99938
Type-2	0.99324	0.99847
Type-2 (blind)	0.98854	1.0
Type-2 (consistent)	0.99360	0.99833
Very-strongly type-3	0.98407	0.92601

us with an appropriate threshold for cognitive complexity (Muñoz et al., 2020).

We might also label clone fragments for refactoring if they are under constant modification and avoid labeling those that are not. Moreover, one could prevent labeling code for refactoring that is boilerplate, generated, an interface implementation, or a pattern implementation. Ultimately, each development team will establish its own thresholds and rules, so our suggestions are only a starting point.

3 Validation of clone detection algorithm

The BigCloneEval framework enables researchers to perform evaluation experiments for clone detection tools using the BigCloneBench clone benchmark (Svajlenko et al., 2014; Svajlenko and Roy, 2021). We applied this benchmark to our clone detection tool. The parameters used for the validation were: `-st both -m 'CoverageMatcher 0.7' -mis 80 -mil 5 -mip 5 -mit 40`. Our tool reported a high precision of 0.99993, with 60,350 true positives and four false positives. It also reported a high recall, as shown in Table 1, and its performance is similar to that of the NiCad6 tool. It took 14:03 min for our tool to detect the clones from the BigCloneBench with 55,499 files, over 15.4M LOC, and 460,138 methods over the configured thresholds. Our algorithm outperformed NiCad and other tools in speed, as shown in Table 2 (Feng et al., 2020). Feng et al. ran their experiments on an Intel Core i7-7700K, with 16GB RAM and an SSD, while we ran ours on an i7-7700HQ with 16GB RAM and an SSD, which makes the results comparable. The list of clones found in compressed files, the complete benchmark reports of both the NiCad6 and VizClone tools, and the speed logs of VizClone can be found in our tool's GitHub repository under the `bcb` folder.

4 Results

We applied our method for visual clone analysis to the source code of the six open-source projects described in Table 3 and hosted on GitHub. The IntelliJ IDEA Community Edition and IntelliJ Platform project (JetBrains, 2024), owned by JetBrains, is an open-source IDE for several programming languages. The Elasticsearch project (Elastic, 2024), owned by Elastic, is a distributed search and analytics engine, a scalable data store, and a vector database optimized for speed and relevance on production-scale workloads. The Hadoop project (Apache Software Foundation, 2024), owned by the Apache Software Foundation, is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models with high availability. From Eclipse, the projects Eclipse Platform (Eclipse, 2024a) and Eclipse Platform UI (Eclipse, 2024b) provide the basis for the Eclipse IDE and the basic building blocks for user interfaces, respectively. From Spring, the Spring Framework project (Spring, 2024) is the foundation for all Spring projects and provides core support for dependency injection, transaction management, web apps, data access, messaging, and more.

The plugin's source code is hosted on GitHub under the account `jnavas-tec` and in the project `vizclone`.

The following subsections describe the analysis performed on each of the projects. Although some describe clone classes with a high number of fragments or high Cognitive Complexity, this does not necessarily imply that the projects need to refactor the clone classes found.

4.1 IntelliJ Community project analysis

Our method identified 35,136 Java files in the IntelliJ Community project and 29,962 methods with more sentences and tokens than their thresholds. The screening step yielded 8,354 clone candidate pairs, and the detection step confirmed 7,801 as true clones grouped into 591 clone classes. The number of clone classes with Cognitive Complexity over 15 is 129, while the number of clone classes with more than five fragments is 43. All of these clone classes highlight candidates for refactoring. Manual inspection is required to determine if refactoring is truly required. The search process took under 7 min.

The first clone class has 64 fragments, and its Cognitive Complexity is 3. All those fragments correspond to methods within the same Java class, `MantisConnectBindingStub.java`, and contain boilerplate code for SOAP action calls. The second clone class has 53 fragments, and its Cognitive Complexity ranges from 6 to 36. The corresponding methods implement the `equals` pattern. The third clone class has 47 fragments, and its Cognitive Complexity ranges from 5 to 45. The corresponding methods implement the `Comparable` interface for classes in the same package `com.jetbrains.python.console.protocol`. We inspected all the clone classes with more than five fragments, and all of them fall into similar categories as the previous ones.

The clone class with the highest Cognitive Complexity value of 117 has five fragments. Its fragments correspond to the method `balanceDeletion` for concurrent hash map classes inside different packages. The five fragments are almost identical. Two methods are from classes marked as deprecated: `com.intellij.util.containers.ConcurrentIntObjectHashMap` and `com.intellij.util.containers.ConcurrentLongObjectHashMap`, replaced by two others with the same names but in the package `com.intellij.concurrency`. The fifth fragment is from the class `com.intellij.concurrency.ConcurrentHashMap` in the same last package. Although these fragments are few and it appear unlikely they will change, it could be worth examining them because of their

TABLE 2 Execution time for VizClone and other tools (Feng et al., 2020).

LoC	NiCad	CCAligner	CCFinderX	SourcererCC	VizClone
10 M	11 h 42 min 47 s	24 min 56 s	28 min 51 s	32 min 11 s	-
15.4 M	-	-	-	-	14 min 3 s

TABLE 3 Duplicate code metrics for open-source projects.

Project	Java files	Methods	Sim	Clones	Groups	CC > 15	Frag > 5	Time
IntelliJ community	35,136	29,962	8,354	7,801	591	129	43	6:39
Elastic search	10,313	9,846	5,996	5,140	624	45	49	1:49
Apache Hadoop	7,343	7,563	1,037	843	313	23	10	1:50
Eclipse platform UI	5,669	5,460	862	650	267	41	11	1:07
Eclipse platform	4,223	4,805	455	365	217	33	2	0:40
Spring framework	4,507	2,482	237	180	136	23	1	0:38

high Cognitive Complexity value. The second clone class has three fragments, and its Cognitive Complexity ranges between 72 and 111. The corresponding methods are in the same class `com.intellij.concurrency.ConcurrentHashMap` and implement different flavors of remapping an existing key-value pair. The third clone class has five fragments, and its Cognitive Complexity ranges between 103 and 106. The corresponding methods come from the same previous classes `ConcurrentHashMap`, `ConcurrentIntObjectHashMap`, and `ConcurrentLongObjectHashMap`. The methods implement `transfer` to copy the nodes in each bin to a new table. We also inspected all the remaining 126 clone classes with a Cognitive Complexity above 15, and all implemented boilerplate code, pattern interfaces, lexer or parser actions, or protocol actions.

4.2 Elasticsearch project analysis

We visually analyzed the Elasticsearch project; Table 3 shows it has 10,313 Java files and 9,846 methods above the thresholds. It generated 5,996 clone pair candidates from the screening stage, and the detection stage yielded 624 clone classes and 5,140 confirmed clone pairs. There are 49 clone classes with more than five fragments and 45 with Cognitive Complexity values above 15. It took approximately 2 min to complete the search process.

We skipped the integration test suites interleaved with the source code because their methods contain numerous boilerplate source code patterns identified as clones.

The clone class with more fragments has 111, and its Cognitive Complexity value ranges from 1 to 23. All the fragments conform to a pattern that returns a JSON builder for multiple Java classes. The second clone class has 19 fragments, and its Cognitive Complexity varies between 10 and 44. The methods configure and return a JSON query builder from a JSON content parser. The third clone class has 17 fragments; its Cognitive Complexity values range between 1 and 7. The methods are owned by the three parsing Java classes `SqlBaseParser`, `EqlBaseParser`, and `PainlessParser`; they return different subclasses of `ParserRuleContext`. Upon inspection, all the other clone classes with more than five fragments are related to parsing, interface implementation, or boilerplate code.

The three clone classes with the highest Cognitive Complexity have values of 86, 86, and 56, respectively, with each containing two fragments. The first clone class consists of the method `innerFromXContent` from the Java classes `org.elasticsearch.client.tasks.ElasticsearchException` and `org.elasticsearch.ElasticsearchException`; this method yields an `ElasticsearchException` based on an `XContentParser`. The second clone class involves the method `parseMath` found in the Java classes `org.elasticsearch.common.joda.JodaDateMathParser` and `org.elasticsearch.common.time.JavaDateMathParser`; this method parses mathematical operations on a date using a time value, a rounding flag, and a timezone, returning the outcome in milliseconds. The other clone classes with a Cognitive Complexity exceeding 15 focus on implementing JSON builders, parsing actions, pattern configurations, sorting actions, and mathematical operations.

4.3 Apache Hadoop project analysis

When applying the method to the Apache Hadoop project (see Table 3), which has 7,343 Java files and 7,563 methods above the thresholds, the screening step produced 1,037 candidate pairs. In contrast, the detection step verified 843 clone pairs grouped into 313 clone classes. The number of clone classes with Cognitive Complexity over 15 is 23, and the number of clone classes with more than five fragments is 10. All of these clone classes point out candidates for refactoring. Manual inspection is required to determine whether refactoring is truly necessary. The search process took under 2 min.

The clone class with more fragments has 35, and its Cognitive Complexity is 2. All those fragments correspond to methods within the same Java class, `FSNamesystem.java`, a container for namespace state, and contain boilerplate code called by HadoopFS clients to modify and query the container. The second clone class has 24 fragments, and its Cognitive Complexity runs between 8 and 31. The corresponding methods

implement the `equals` pattern. The third clone class has 15 fragments, and its Cognitive Complexity ranges between 1 and 3. The corresponding methods implement HTTP requests for the `TimelineReaderWerServices.java` class. We inspected all the remaining clone classes with more than 5 fragments, and all of them fall into similar categories as the previous ones.

The clone class with the highest Cognitive Complexity value of 60 has two fragments. These fragments correspond to the methods `hbMakeCodeLengths` on the char array and byte array in the class `org.apache.hadoop.io.compress.bzip2.CBZip2OutputStream`. These methods appear very unlikely to change. The second clone class has two fragments; their Cognitive Complexity values are 50 and 41. The methods are `processMapAttemptLine` and `processReduceAttemptLine` from the `org.apache.hadoop.tools.rumen.HadoopLogsAnalyzer` class. The third clone class also has two fragments; their Cognitive Complexity values are 47 and 20. The corresponding methods are named `convertToApplicationAttemptReport` and come from the classes `org.apache.hadoop.yarn.server.applicationhistoryservice.ApplicationHistoryManagerOnTimelineStore` and `org.apache.hadoop.yarn.util.timeline.TimelineEntityV2Converter`. The first method generates a report on the events of the received entity, while the second generates a report on the entity's information. We inspected the remaining 23 clone classes with a Cognitive Complexity above 15, and the code implements boilerplate code, pattern interfaces, and protocol actions.

4.4 Eclipse platform UI project analysis

The analyzed revision of this project has 5,669 Java files and 5,460 methods with sufficient sentences and tokens. The screening stage delivered 862 candidate pairs for clones, and the detection stage verified 650 true clones grouped into 267 clone classes. A total of 41 clone classes have a Cognitive Complexity exceeding 15, and 11 clone classes have more than five fragments; all of these are candidates for refactoring and require manual inspection to determine whether to factorize. The search took about 1 min.

The top three clone classes with more than 15 fragments have 18, 9, and 8, respectively. The first clone class groups methods that determine identifiers for different UI components based on context information. Their Cognitive Complexity varies from 9 to 30, and the related classes are all within the package `org.eclipse.e4.ui.model.application`. The second clone class contains methods that synchronize extension points across several UI registry Java classes under the package `org.eclipse.ui.internal.genericeditor`, all of which have a Cognitive Complexity value of 4. The third clone class corresponds to the method `eIsSet`, which returns whether a corresponding feature in the UI is set for various Java classes within the same package `org.eclipse.ui.internal.genericeditor`; their Cognitive Complexity values range from 3 to 21. The remaining eight clone classes, each with more than five fragments, correspond to methods from classes that implement interfaces within the UI components inheritance hierarchy.

The clone class with the highest Cognitive Complexity has seven fragments with complexity values between 8 and 216. Its fragments correspond to the method `doSwitch` from Java classes in the package `org.eclipse.e4.ui.model.application`, which implement the `Switch` for the model's inheritance hierarchy. The second clone class has two fragments with a Cognitive Complexity of 81. The method implemented is `processChange` from the Java classes `org.eclipse.text.undo.DocumentUndoManager` and `org.eclipse.jface.text.DefaultUndoManager`, which records changes in a document to be managed. The third clone class also has two fragments with a Cognitive Complexity of 50. The method implemented is `refresh` from the Java classes org.eclipse.ui.internal.progress.ProgressInfoItem and org.eclipse.e4.ui.progress.internal.ProgressInfoItem, which refreshes progress updates on the UI. These classes represent items used to show jobs in the UI. The remaining 38 clone classes, with Cognitive Complexity over 15, correspond to methods from classes that implement interfaces under the UI components inheritance hierarchy.

4.5 Eclipse platform project analysis

The visual analysis method examined 4,223 Java files in this project, filtered out methods with insufficient sentences or tokens, and produced 4,805 methods for inspection. From the screening step, it generated 455 candidate pairs for clones, and from the detection step, it identified 365 verified clones arranged into 217 clone classes. As shown in Table 3, it found 33 clone classes with Cognitive Complexity over 33 and only two clone classes with more than five fragments. The search process took 40 s to complete.

The clone class with the highest Cognitive Complexity is the same as the one with the most fragments; it has eight fragments with Cognitive Complexity ranging from 47 to 233. All the methods are in the class `org.apache.lucene.demo.html.HTMLParserTokenManager`, which is an HTML parser generated with JavaCC, a popular Java parser generator. The second clone class with six fragments corresponds to the methods named `startElement`, which have Cognitive Complexity values from 2 to 5. These methods belong to parsing classes in the package `org.eclipse.help.internal.webapp.parser`, which are specializations of the Java class `ResultParser`.

The second clone class, which has a high Cognitive Complexity value of 57, contains two fragments with Cognitive Complexity values of 48 and 57, respectively. The method names are `findNextPrev`, which belong to the Diff TreeViewer classes `org.eclipse.team.internal.ui.synchronize.AbstractTreeViewerAdvisor` and `org.eclipse.compare.structuremergeviewer.DiffTreeViewer`. The third clone class, with a Cognitive Complexity value of 45, has two fragments with Cognitive Complexity values of 29 and 45, respectively. The method name is `loadContentForExtendedMemoryBlock`, associated with the Java classes`org.eclipse.debug.internal.ui.`

views.memory.renderings.TableRenderingContentProvider and org.eclipse.debug.internal.ui.elements.adapters.MemoryBlockContentAdapter. All the remaining 30 clone classes, which have a Cognitive Complexity over 15, contain fewer than five fragments and do not appear to be suitable candidates for refactoring.

4.6 Spring framework project analysis

Our method found 4,507 Java files in the Spring Framework project and 2,482 methods with more sentences and tokens than their thresholds. The screening step yielded 237 clone candidate pairs, and the detection step confirmed 180 as true clones, grouped into 136 clone classes. Only one of the clones has more than five fragments, but 23 clones have Cognitive Complexity over 15. To decide whether these clones should be refactored, manual inspection must be performed. The search took 38 s.

There is only one method with more than five fragments, and its Cognitive Complexity is as low as seven. It corresponds to the overloading of the method getValue from the Java class org.springframework.expression.spel.standard.SpelExpression. These do not apply as candidates for refactoring.

The three clone classes with the highest Cognitive Complexity have two fragments, with their Cognitive Complexity values being 60, 48, and 43, respectively. The first corresponds to the method parseSqlStatement from the Java classes org.springframework.r2dbc.core.NamedParameterUtils and org.springframework.jdbc.core.namedparam.NamedParameterUtils, which parses a SQL statement. The second is for the method skipCommentsAndQuotes from the same Java classes as in the previous clone, and it is called from the previous method. The third clone class consists of two methods: buildPersistenceMetadata from the Java class org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor and buildResourceMetadata from the Java class org.springframework.context.annotation.CommonAnnotationBeanPostProcessor. Both methods are used to generate injection metadata. The remaining 20 clone classes have a Cognitive Complexity over 15. have less than five fragments, and are unlikely to be suitable for refactoring.

5 Discussion

Our proposed method for developing metrics-based visual analytic tools is demonstrated by the implemented plugin as proof of concept. Although the tool is not intended for publication as a plugin in the IntelliJ Marketplace for IntelliJ IDEA Community Edition, it illustrates how to extract metrics from the source code of a functional project in the IDE and employ them to conduct visual analysis of various quality aspects that need measurement and control. The implemented use case is based on duplicate code metrics combined with Cognitive Complexity metrics, but it could be easily enhanced by incorporating the extraction and combined analysis of additional metrics, such as direct and indirect coupling metrics or source code repository revision metrics. The visual representation of several combined metrics and the use of thresholds facilitate the identification of code areas that are candidates for refactoring.

Future enhancements involve refining the code into a publishable open-source plugin, incorporating source code tracking and documentation capabilities, such as automatically adding annotations to metrics outlier methods, to enhance their visibility and clarify their evolution.

Integrating the method into the IDE requires significant effort and programming, but the rewards in terms of ease of control, speed, and convenience outweigh the costs involved. The visual analysis enhances the analytical capabilities of developers within the IDE by allowing them to interact with the metrics in a more direct manner. For example, sorting the measured elements using a combination of several metrics involves ordering the clone classes in descending order first by Cognitive Complexity and then by the number of fragments, or by ordering the clone classes in descending order first by the number of fragments and then by Cognitive Complexity. A wealth of information about all code hotspots can be visually accessed simultaneously in a compact space, and navigating directly into the code for inspection is easily available.

Data availability statement

The datasets presented in this study can be found in online repositories. The names of the repository/repositories and accession number(s) can be found at: https://github.com/jnavas-tec/vizclone.

Author contributions

JN-S: Investigation, Methodology, Software, Validation, Writing – original draft. AG-T: Conceptualization, Investigation, Methodology, Supervision, Writing – review & editing. FH-C: Visualization, Writing – review & editing.

Funding

The author(s) declare that financial support was received for the research and/or publication of this article. The authors wish to acknowledge the support provided by the Postgraduate Office of ITCR for this publication.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Generative AI statement

The author(s) declare that no Gen AI was used in the creation of this manuscript.

Publisher's note

References

Alam, A. I., Roy, P. R., Al-Omari, F., Roy, C. K., Roy, B., and Schneider, K. A. (2023). "GPTCloneBench: a comprehensive benchmark of semantic clones and cross-language clones using GPT-3 model and SemanticCloneBench," in *Proceedings of the 39th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 688–699. doi: 10.1109/ICSME58846.2023.00013

AlOmar, E. A., Mkaouer, M. W., and Ouni, A. (2024). Behind the intent of extract method refactoring: a systematic literature review. *IEEE Trans. Softw. Eng.* 50, 668–694. doi: 10.1109/TSE.2023.3345800

Apache Software Foundation. (2024). *Hadoop*. Available online at: https://github.com/apache/hadoop/tree/07627ef19e2bf4c87f12b53e508edf8fee05856a (accessed March 18, 2024).

Assi, M., Hassan, S., and Zou, Y. (2025). Unraveling code clone dynamics in deep learning frameworks. *ACM Trans. Softw. Eng. Methodol.* doi: 10.1145/3721125. [Epub ahead of print].

Eclipse. (2024a). *Eclipse Platform*. Available online at: https://github.com/eclipse-platform/eclipse.platform/tree/e14565e5022852f2e1eebadbe12d09f4cee88378 (accessed March 18, 2024).

Eclipse. (2024b). *Eclipse Platform UI*. Available online at: https://github.com/eclipse-platform/eclipse.platform.ui/tree/1eead54aac4c037be1bbc08870ccf27aa870cfc8 (accessed March 18, 2024).

Elastic. (2024). Elasticsearch. Available online at: https://github.com/elastic/elasticsearch/tree/4944959acff48ffa4979c406407d3abcbb371655 (accessed March 19, 2024).

Feng, C., Wang, T., Liu, J., Zhang, Y., Xu, K., and Wang, Y. (2020). "NiCad+: speeding the detecting process of NiCad," in *2020 IEEE International Conference on Service Oriented Systems Engineering (SOSE)* (Los Alamitos, CA: IEEE Computer Society), 103–110. doi: 10.1109/SOSE49046.2020.00019

Feng, S., Suo, W., Wu, Y., Zou, D., Liu, Y., and Jin, H. (2024). "Machine learning is all you need: a simple token-based approach for effective code clone detection," in: *Proceedings of the 46th International Conference on Software Engineering (ICSE)* (New York, NY: Association for Computing Machinery). doi: 10.1145/3597503.3639114

Gotoh, O. (1982). An improved algorithm for matching biological sequences. *J. Mol. Biol.* 162, 705–708. doi: 10.1016/0022-2836(82)90398-9

Hu, T., Xu, Z., Fang, Y., Wu, Y., Yuan, B., Zou, D., et al. (2023). "Fine-grained code clone detection with block-based splitting of abstract syntax tree," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 89–100. doi: 10.1145/3597926.3598040

JetBrains. (2024). *IntelliJ IDEA Community Edition*. Available online at: https://github.com/JetBrains/intellij-community/tree/33034c11b946c8ada97f7cef1590d40b60148e76 (accessed March 19, 2024).

Kanwal, J., Maqbool, O., Basit, H. A., Sindhu, M. A., and Inoue, K. (2022). Historical perspective of code clone refactorings in evolving software. *PLoS ONE* 17:e0277216. doi: 10.1371/journal.pone.0277216

Kaur, M., and Rattan, D. (2023). A systematic literature review on the use of machine learning in code clone research. *Comput. Sci. Rev.* 47:100528. doi: 10.1016/j.cosrev.2022.100528

Khajezade, M., Wu, J. J. W., Fard, F. H., Rodríguez-Pérez, G., and Shehata, M. S. (2024). "Investigating the efficacy of large language models for code clone detection," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension (ICPC 2024, ERA Track)* (New York, NY: Association for Computing Machinery). doi: 10.1145/3643916.3645030

Lei, M., Li, H., Li, J., Aundhkar, N., and Kim, D.-K. (2022). Deep learning application on code clone detection: a review of current knowledge. *J. Syst. Softw.* 184:111141. doi: 10.1016/j.jss.2021.111141

Leskovec, J., Rajaraman, A., and Ullman, J. D. (2020). *Mining of Massive Datasets*. Cambridge University Press, 3 Edition. doi: 10.1017/9781108684163

Li, J., Tao, C., Jin, Z., Liu, F., and Li, G. (2023). "ZC3: zero-shot cross-language code clone detection," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Echternach: IEEE Press), 400–412. doi: 10.1109/ASE56229.2023.00210

Muñoz, M., Wyrich, M., and Wagner, S. (2020). "An empirical validation of cognitive complexity as a measure of source code understandability," in *ESEM 20: Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement* (New York, NY: ACM Digital Library), 1–12. doi: 10.1145/3382494.3410636

Needleman, S. B., and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.* 48, 443–453. doi: 10.1016/0022-2836(70)90057-4

Pinku, S. N., Mondal, D., and Roy, C. K. (2024). "On the use of deep learning models for semantic clone detection," in *Proceedings of the 40th IEEE International Conference on Software Maintenance and Evolution (ICSME)* (Flagstaff, AZ: IEEE). doi: 10.1109/ICSME58944.2024.00053

Smith, T., and Waterman, M. (1981). Identification of common molecular subsequences. *J. Mol. Biol.* 147, 195–197. doi: 10.1016/0022-2836(81)90087-5

Spring. (2024). *Spring Framework*. Available online at: https://github.com/spring-projects/spring-framework/tree/9f7a94058a4bbc967fe47bfe6a82d88cb3feddfb

Sundelin, A., Gonzalez-Huerta, J., Torkar, R., and Wnuk, K. (2025). Governing the commons: code ownership and code-clones in large-scale software development. *Empir. Softw. Eng.* 30, 1–42. doi: 10.1007/s10664-024-10598-7

Svajlenko, J., Islam, J. F., Keivanloo, I., Roy, C. K., and Mia, M. M. (2014). "Towards a big data curated benchmark of inter-project code clones," in *Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME* (Victoria, BC: Institute of Electrical and Electronics Engineers Inc.), 476–480. doi: 10.1109/ICSME.2014.77

Svajlenko, J., and Roy, C. K. (2021). *BigCloneBench*. Springer Singapore: Singapore, 93–105. doi: 10.1007/978-981-16-1927-4_7

Thaller, H., Linsbauer, L., and Egyed, A. (2022). "Semantic clone detection via probabilistic software modeling," in *Fundamental Approaches to Software Engineering* (Munich: LNCS), 288–309. doi: 10.1007/978-3-030-99429-7_16

Wang, Y., Ye, Y., Wu, Y., Zhang, W., Xue, Y., and Liu, Y. (2023). "Comparison and evaluation of clone detection techniques with different code representations," in *Proc. of the 45th Int. Conference on Software Engineering: Software Engineering in Practice* (ICSE-SEIP), 84–93. doi: 10.1109/ICSE48619.2023.00039

Xu, Z., Qiang, S., Song, D., Zhou, M., Wan, H., and Zhao, W. (2024). "DSFM: enhancing functional code clone detection with deep subtree interactions," in *Proceedings of the 46th International Conference on Software Engineering (ICSE)* (New York, NY: Association for Computing Machinery). doi: 10.1145/3597503.3639215

Yahya, M. A., and Kim, D.-K. (2023). Clcd-i: cross-language clone detection by using deep learning with infercode. *Computers* 12:12. doi: 10.3390/computers12010012

Zakeri-Nasrabadi, M., Parsa, S., Ramezani, M., Roy, C. K., and Ekhtiarzadeh, M. (2023). A systematic literature review on source code similarity measurement and clone detection: techniques, applications, and challenges. *J. Syst. Softw.* 204:111796. doi: 10.1016/j.jss.2023.111796

Zhang, Z., and Saber, T. (2025a). Exploring the boundaries between llm code clone detection and code similarity assessment on human and ai-generated code. *Big Data Cogn. Comput.* 9:41. doi: 10.3390/bdcc9020041

Zhang, Z., and Saber, T. (2025b). Machine learning approaches to code similarity measurement: a systematic review. *IEEE Access* 13, 51729–51744. doi: 10.1109/ACCESS.2025.3553392