SN Computer Science (2023) 4:470
https://doi.org/10.1007/s42979-023-01910-1

ORIGINAL RESEARCH

Detecting Semantic Clones in Microservices Using Components

Amr S. Abdelfattah[1] · Alejandro Rodriguez[1] · Andrew Walker[1] · Tomas Cerny[1]

Received: 28 August 2022 / Accepted: 27 April 2023

Abstract
The growing popularity of enterprise technologies for decentralized systems leads to commonalities in using components. This direction, however, opens new challenges to code clone detection. Approaches can no longer look at the low-level code but must deal with the higher-level component semantics. Yet, not many works addressed this trend. One of the quality issues that can be identified in large systems is duplicated behavior with different syntactic structures. It is crucial to detect these issues for enterprises where software's codebase(s) grows and evolves, and maintenance costs rise significantly. This issue is referred to as a semantic clone. The detection of semantic clones requires semantic information about the given program. Unfortunately, while many code clone detection techniques are proposed, there is a lack of solutions targeted explicitly toward enterprise systems and even fewer solutions dedicated to semantic clones. To reason about semantic clones, we consider different pairs of component call-graphs in the system. Since different component types are common in enterprise systems, we can ensure that only relevant fragments are matched, using targeted enterprise metadata. When applied to an established system benchmark, our method indicates high accuracy in detecting semantic clones. We also assessed different system versions to elaborate on the method's applicability to decentralized system evolution.

Keywords Code Quality · Code Clone · Semantic Clone · Static Analysis · Enterprise Technology · Microservices · Cloud Systems · System Evolution

Introduction

Source code duplication by copying and pasting into another section of source code, even with minor modification, often happens throughout software development and maintenance. It can result from lacking development skills or rushed development, prioritizing visible profit in the form of new feature delivery over less visible code quality [1]. This copied code is called code clone, and the process is called code cloning [2–4]. A code clone is a code fragment with other code fragments identical or similar to it in the source code [5]. Various studies suggested that almost 20–50% of large software systems consist of cloned code [2–4]. These clones have two main categories: a syntactic clone [5] considers the code structure and syntactic variants, and a semantic clone [6] is concerned with similar functionality regardless of the different syntactic variants and implementation. Obviously, semantic clones can emerge from other means than copying and pasting and could result from distinct developer ambitions or appear in system integration.

Code cloning impedes software maintenance, as extended efforts are needed to apply fixes at multiple clone locations, possibly leading to a ripple effect or leaving inconsistencies in the codebase. Identified errors require code corrections; however, if the relevant code segment is a clone, it is essential to identify all related segments throughout the source code. However, it is difficult to manually detect the code cloning to be refactored from a large number of lines of code. As a result, this increases software maintenance costs [6]. Clone detection tools could considerably simplify the

This article is part of the topical collection "Advances on Cloud Computing and Services Science" guest edited by Donald F. Ferguson, Claus Pahl and Maarten van Steen.

✉ Tomas Cerny
tomas_cerny@baylor.edu

Amr S. Abdelfattah
amr_elsayed1@baylor.edu

Alejandro Rodriguez
alejandro_rodriguez4@baylor.edu

Andrew Walker
andrew_walker2@baylor.edu

[1] Department of Computer Science, Baylor University, One Bear Place #97141, Waco, TX 76798, USA

Published online: 23 June 2023

tedious work of clone identification. Scalability is an important property of the tools that detect the code to be refactored [5]. Ajad et al. [6] surveyed 23 research articles published in top-tiered venues during 2008–2020 in the domain of semantic code clone detection and identified various available tools that detect clones in software systems. The survey observed that cloning increases maintenance costs, the number of lines of code, and bug propagation. Furthermore, it sometimes limits further system development through the existing source code.

Still, most existing clone approaches detect syntactic clones. The semantic clones require essential analysis of the actual behavior and a deeper understanding of the intent of the code fragment. For example, two code blocks that perform the same calculation, but use different variables and functions, would be considered semantically similar. However, semantic-based approaches can detect syntactic clones as well [2, 7]; therefore, semantic clones are considered more meaningful than syntactic clones because they indicate that the same functionality is duplicated in the code either with similar or different syntax, which can lead to maintenance issues and increased complexity. In general, this is a challenging problem; therefore, very few approaches attempt to detect semantic clones [8].

Analyzing decentralized enterprise systems (i.e., microservices or cloud-native systems) can be challenging due to their distinct characteristics, such as the presence of multiple codebases within a system and the use of different languages and technologies across services. Thus, a high-level, component-based approach is necessary to overcome these challenges rather than relying solely on low-level code analysis and its fundamental properties.

This work targets semantic clone detection in decentralized enterprise systems. It takes into account that current development practice does not use low-level coding but rather builds on well-established components like endpoints, controllers, services, repositories, entities, remote calls, messaging, etc. Considering that components augment code segments with additional semantics, these components and their attributes can be used when determining whether selected code fragments could be semantic clones.

In particular, the proposed method extracts call graphs from across system endpoints. These call graphs are constructed in a way that nodes represent cross-cutting components capturing their types and additional attributes. To determine the semantic similarity in the system, we consider the similarity of such graphs across system endpoints. This work introduces a comprehensive methodology that evolved from our previous work [9]. It illustrates how a system abstraction can be constructed using high-level design elements that are common across enterprise frameworks.

We use an established microservice benchmark to illustrate the use of our method and detail multiple perspectives. We elaborate on the initial weight calibration for our method across components and their properties in the call graph and demonstrate how to apply it in system evolution. We also illustrate supportive tools developed for integration with Software Development Life Cycle (SDLC) and CI/CD pipelines for real-life application usage.

This manuscript brings the following outcomes:

- A method to detect semantic clones in enterprise systems using components.
- A prototype tool for semantic clone detection assessed on an established microservice benchmark.
- A enterprise system semantic clone dataset containing 27,221 labeled items.
- A plugin for integrating our clone detection into the development process.
- An interactive tool for visualizing the identified clones and inconsistencies in system components was created.

The rest of the paper is organized as follows: Section "Background" discusses the background, and the section "Related Work" details related work. Section "Semantic Code Clone Methodology" outlines the semantic clone methodology and its phases. Then, Section "Case Study" illustrates the method implementation in a case study along with a discussion. Section "Integration to the Development Process" elaborates on tools developed for practical application in development workflows. Section "Discussion" provides a discussion, and the section "Threats to Validity" discusses the threats to the validity of the proposed approach. Finally, the paper is concluded in the section Conclusion.

Background

Code clones can be detected at different levels of code matching. The below list iterates over the Basic types of clones [2, 10]:

- Exact clones (Type 1): Identical code segments except for changes in comments, layouts, and whitespaces.
- Renamed clones (Type 2): Code segments that are syntactically or structurally similar other than changes in comments, identifiers, types, literals, and layouts. These clones are also called *parameterized clones*.
- Near Miss clones (Type 3): Copied pieces with further modification such as addition or removal of statements and changes in whitespaces, identifiers, layouts, com-

ments, and types but outcomes are similar. These clones are also known as gapped clones.
- Semantic clones (Type 4): More than one code segment that is functionally similar but implemented by similar or different syntactic variants like iterative and recursion approaches for the same algorithm implementation.

The researchers grouped these types into two main categories of clones: syntactic clones, which include Type-1, Type-2, and Type-3 [5], and semantic clones [6]. The syntactic similarity spectrum is extensive and includes subcategories for Type-3 based on the syntactical similarity percentage [11]. For instance, "Very Strongly Type-3" has a similarity range of [90–100%), "Strongly Type-3" falls within [70–90%) and "Moderately Type-3" ranges from [50–70%), while "Weakly Type-3" has a range of [0–50%).

Classifications of Clone Detection

Code clones are often operationally defined by individual clone detection methods. The established detection techniques can be categorized as suggested by Kumar et al. and others [5, 6]:

Text-based approaches are capable of detecting exact clones. Each line of the source code is compared with other lines. If consecutive lines of code are identical to other lines of code, they are detected as code clones. This method cannot detect code clones with different identifiers such as variable names, function names, or type names [5].

Token-based approaches can detect code clones that have different formats, such as different indention, white space, tab, and different identifiers. After the source code is divided into tokens, identical token sequences that are longer than a certain length are detected as code clones. They are competent in detecting all syntactic types of clones. Nevertheless, they can not detect semantic clones efficiently.

Abstract Memory States (AMS) provide a quick, lower-level analysis [12]. AMS methods cannot handle scopes beyond single methods. While using AMS helped lessen the run time and lowered false-positive rates, its usefulness is limited to decentralized systems. This limitation renders them much less useful for enterprise applications, where the flow of method calls is more important for determining duplicate behavior due to the separation of concerns making some methods extremely short.

Abstract Syntax Tree (AST)-based gives the abstract syntactic structure of the code and is used as the base for other methods. It is often associated with syntactic clones [6] but has also been used for semantic clone detection. After building an AST from the source code, subtrees having the same structure are detected as code clones. It identifies cohesive parts of the program source code as code clones, such as whole methods or consecutive statements in the same scope. However, comparing subtrees is expensive, so it is difficult to apply the AST-based technique to large-scale software systems, given their volumes of low-level detail.

Metric-based techniques specify and measure several metrics from a certain unit as the function, method, or class of the software system. The units having identical or similar metrics values are detected as code clones. It can detect syntactic clones; however, they have low accuracy in semantic clones in comparison to the graph-based approaches.

Program Dependency Graph (PDG)-based approaches can detect semantic clones with good accuracy. After building a PDG as a result of the semantic analysis of the source code, isomorphic subgraphs are detected as code clones. It is the most expensive technique available, so it is difficult to apply the technique to middle-scale or large-scale software systems. Sometimes, it is observed that PDG-based tools are highly scalable and portable rather to other approaches like text-based, token-based, etc.

To avoid the time complexity of the PDG, Control-Flow Graph (CFG) [9, 13] can capture the same control dependence as PDG. CFG uses graph representation that depicts the relationships and order of operations, of which all paths might be traversed through a program during its execution. It captures the behavior of the program at a component level, making it easier to analyze than low-level code, and provides a clear understanding of the system's behavior. It can easily encapsulate the information per each basic block. It locates inaccessible codes of a program, and syntactic structures such as loops are easy to detect in a CFG. It shows notable performance in measuring semantic clones. Still, CFG provides fine detail of program structure, and the reduction of the low-level details leads to call graphs. Call graphs represent the calling relationships between subroutines in a program.

There are some attributes that influence clone detection, such as the granularity level that could be a method, function, code fragment, or procedure block, depending on the particular programming paradigm followed by a language [14]. A code fragment is a set of code lines, not necessarily contiguous. It is the slice as computed at the variable-level, function-level, or file-level [15].

The choice of system intermediate representation and detection algorithm has a significant influence on clone detection. Clone detection is related to the representation that the detection approach relies on. Thus, Type-1 clones are discovered by text-based approaches, Type-2 by lexical or token-based approaches, while Type-3 clones are discoverable by syntactical approaches based on AST, possible with the involvement of information from software metrics. Using AST as a system intermediate representation suits better for detecting Type-3 clones. Kumar et al. [6] suggest that AST cannot detect Type-4 clones with better accuracy. Finally, graph-based and hybrid approaches

Table 1 Related Work Approaches (SA: Static Analysis, MA: Manual Analysis, ML: Machine Learning, IR: Intermediate Representation)

Ref.	GUI	Clone	Technique	IR	System	Enterprise
[5]	Yes	Syntactic	SA	AST Token	Monolithic	No
[18]	No	Semantic	SA/ ML	Comment	Monolithic	No
[19]	No	Syntactic Semantic	SA/ ML	PDG AST BDG	Monolithic	No
[13]	No	Syntactic Semantic	SA/ ML	AST CFG	Monolithic	No
[20]	No	Syntactic	SA	CFG Token	Monolithic	No
[15]	No	Syntactic Semantic	SA	CFG LSH	Monolithic	No
[21]	No	Semantic	SA	CFG	Monolithic	No
[14]	Yes	Semantic	SA	eCST	Monolithic	No
[22]	Yes	Syntactic	MA	Manual: -method, -file	Microservice	Yes
This work	Yes	Semantic	SA/ ML	CCG	Microservice	Yes

are involved in the detection of Type-4 clones. Hybrid approaches combine several different methods [14, 16, 17].

In this paper, the intermediate representation is Component Call Graph (CCG). It is extracted from the CFG and augmented with cross-cutting components. It provides a comprehensive overview of the architectural structure and its flow. The CCG indicates the control flow between components and implies component dependencies. The calls include both internal and external calls across microservices.

Related Work

Clone detection is a wide-ranging field of research. However, several limitations arise when detecting semantic clones, including scalability issues for enterprise application clone detection. The various approaches are evaluated based on multiple criteria, as summarized in Table 1. This table considers the presence of a Graphical User Interface (GUI), the type of clones analyzed (syntactic or semantic), the techniques used, the artifacts utilized, the focus on monolithic or microservices-based systems, and finally if the approach considers multi-layer systems in enterprise settings.

Approaches that are fantastic for any of the four types of code clones may not provide useful analysis for enterprise applications. For instance, the tool CCFinder [23] is an example of code clone detection that has been implemented and can discover Types 1 through 3 code clones efficiently and effectively. They focused heavily on maintainability and can show the impact of removing a code clone from the system. Yoshiki et al. [5] utilized CCFinder as an analysis component to be integrated with a GUI component to introduce the tool called Aries. The proposed tool automatically computes metrics that are indicators for specific refactoring methods rather than suggesting the refactoring methods themselves. In conclusion, they stated that CCFinder has high scalability. It can finish code clone detection within an hour, even if the source code has millions of lines of code. However, the CCFinder acknowledges that inter-method flows are challenging to capture, and they focus exclusively on source code analysis. Thus this tool is not beneficial for large and complex enterprise systems that are dependent on inter-flow communication. Therefore, even tools that are fantastic for Types 1 through 3 code clones may not provide useful analysis for enterprise applications.

Generally, existing techniques do not scale very well to large-scale software systems since most of them use PDGs for computation, which is a costly process in the large and distributed context [15]. That supports the usage of CFG as an optimal alternative, such that it can achieve the required task with much fewer computations [15]. Furthermore, addressing the semantic showed tools are using machine learning and deep learning-based approaches to detect semantic clones, and those approaches can detect semantic clones with good value of the result. Therefore, most of the related work shows that detecting the semantic clones recommend one or both approaches of machine learning and CFG to be employed in combination with supportive algorithms to achieve good measurements.

Andrian et al. [18] employed the machine learning approach to propose a PROCSSI tool that uses the profiles generated by information retrieval methods, in this case, a vector representation from Latent Semantic Indexing (LSI) [24], to compare components and classify them into clusters of semantically similar concepts. The profile of each source code document generated by LSI can then be used to cluster the documents into related groups. Moreover, the authors augment the real-valued vector representation of the source code documents produced by LSI by adding more dimensions that would represent structural attributes of the source code derived from metrics. Preliminary research on this approach seems promising and would enhance the descriptiveness of the LSI output by including structural type information. However, in the case of renaming the data structure and operation names in a code clone, and when comments are discarded, their measures were unable to detect similarities between the two such implementations. Nevertheless,

this demonstrates the importance of internal documentation for source code understanding. From the perspective of our approach, considering enterprise practices, component types, and their attributes brings a certain level of insight to the code fragment that would otherwise need to be documented. Thus, involving component recognition seems like a well-justified argument for semantic clone detection.

In combination between machine learning and graph-based approaches, Abdullah et al. [19] propose a detection framework for detecting both code obfuscation and clones. The detection of code obfuscation is one of the significant use cases for semantic clones. While Bytecode Dependency Graphs (BDG) and PDGs are two representations of the semantics or meaning of a program, AST captures the structural aspects. The authors build their prototype using the combination of PDG, AST, and Java BDG. Moreover, they integrated many machine-learning-based classifier algorithms and combine them based on the majority decision to obtain the final class label. They concluded that their approach could scale to process millions of files with billions of lines of code in a reasonable amount of time. However, all the Java files have to be compiled into a Java byte before generating BDG and extracting its features. Therefore, all Java files are required to have no errors before compiling to Java ByteCode and generating BDG. The limitations remain for non-bytecode languages such as C# and C/C++.

Wu et al. in [13] combined machine learning with the CFG approach to propose a novel joint code representation that applies fusion embedding techniques to learn hidden syntactic and semantic features of source codes. Each method can be generated into an AST and CFG. As the representations of AST and CFG have different structures, they cannot be fused directly. Therefore, they apply embedding learning techniques [25] to generate fixed-length continuous-valued vectors. These vectors are linearly structured, and thus the syntactic and semantic information can be fused effectively. Word embedding [13] is a collective term for language models and representation learning techniques in natural language processing (NLP).

Much research weighs the CFG approach more for achieving semantic clone detection. Fang et al. [20] propose SCDetector to combine the scalability of token-based methods with the accuracy of CFG for software functional clone detection. To avoid the high-cost graph matching, they first extract the control flow graph by static analysis. Then, they transform the CFG into certain semantic tokens to avoid the high-cost graph matching. They designed a Siamese network [26] to measure the similarity of a code pair. Siamese network has been widely applied in many areas, such as paraphrase scoring, where the inputs are two sentences and the output is a score of how similar they are. However, SCDetector used the Soot framework [6] for achieving the static analysis phase, which requires successfully compiling the given codes to be able to extract the CFG. In addition to SCDetector can only detect method-level code clones and can not handle clones in other code granularity units when the same functionality is implemented using different APIs and different graph structures. Therefore, many semantic considerations are required to be handled, such as the source code normalization and analysis of more accurate orders of all tokens.

SrcClone tool [15] is developed as a slice-based scalable approach that detects both syntactic and semantic code clones. The slice [27] is an executable statement that should preserve the behavior of the original software. SrcClone uses control-flow information as retrieved by srcSlice [28]. However, these slices do not include complete control-flow information; they use limited-flow information that includes dependent variables, called functions, and aliases. As the control flow information is not used and no control dependence is computed, the slices computed and used by srcClone are similar to flow-insensitive data-only slices. The Locality Sensitive Hashing (LSH) algorithm [8] helps srcClone to efficiently find near neighbors of a given slicing vector. This algorithm reports any vector with a specific distance from the query.

Saed et al. [21] propose a novel technique that extracts the semantics of binary code in terms of both data and control flow. They applied data-flow analysis to extract the semantic flow of the registers as well as the semantic components of the control flow graph, which are then synthesized into a novel representation called the semantic flow graph (SFG). Subsequently, they employ the graph edit distance for this purpose. The edit distance between two graphs measures their similarity in terms of the number of edits required to transform one into the other. Given two data flow graphs, to transform one graph into another. They concluded that the system would be efficient enough for most real-world applications. Since it shows an increased accuracy with the languages constrained by the object-oriented paradigm, that is not common to place functions with different semantics in the same source file.

While microservices are a common software architecture for enterprise and scalable distributed systems. The microservices-based systems commonly contain different service components; these components could be written in different programming languages. Tijana et al. [14] presented Language Independent Code Clone Analyzer (LICCA), a tool for the identification of duplicate code fragments across multiple languages. LICCA is integrated with the Set of Software Quality Static Analysers (SSQSA) [29] platform and relies on its high-level representation of code in which it is possible to extract syntactic and semantic characteristics of code fragments positing full cross-language clone

Table 2 Comparative Analysis Results using manual analysis on TrainTicket (0.1.0) benchmark. (μs: Microservice)

Ref.	Clone	Inter- service	Granularity	#clones	#cloned μs
[22]	Syntactic	No	Methods	201 Pairs of Methods	26 μs
This Work	Semantic	Yes	CCG	27 Pairs of CCG	9 μs

detection. SSQSA manages single implementations of each analysis algorithm for all supported languages to guarantee the reliability of software analysis and consistency of results, regardless of the input language. The authors utilize the intermediate representation of source code entities in SSQSA to achieve cross-language analysis. It generates an enriched Concrete Syntax Tree (eCST) [29], which is universal to all languages. eCST is a syntax tree that integrates concrete syntax with specific abstractions. This technique combines the higher recall and the fast detection of the token-based algorithms using the Longest Common Subsequent (LCS) [30] algorithm with the higher precision of AST-based algorithms using eCST representation. It is evaluated using 16 scenarios written in the five languages currently supported by LICCA, i.e., Java, JavaScript, C, Modula-2, and Scheme. However, LICCA has limitations. For example, clone detection is limited to semantically similar fragments written using syntactic elements with similar shapes, while functionally similar fragments implemented by constructs with different shapes (e.g., loops vs. recursion) are not yet covered. Besides, statement order insensitivity is an additional limitation caused by the comparison algorithm choice. Therefore, a refinement of the implementation and involvement of a filtering mechanism is required to increase the precision of results. In addition, code transformation techniques could be used to increase sensitivity.

Continue in deeper contribution related to microservices architecture. Zhao et al. [22] claim that their work is the first that analyzes the reasons systematically for cross-service clones' occurrences. They first adopted the widely-used Nicad tool [31] to detect code clones in microservice projects. Then they created a tool to automatically identify cross-service clones from the detection results. Next, they extracted the files and methods involved in the clones. Finally, they manually analyzed the characteristics of files and methods to understand the reasons for the emergence of cross-service clones. Most microservice projects are implemented in Java [22]; therefore, the authors only considered projects implemented in Java, especially with Spring Boot and Spring Cloud frameworks. Although this study considers the syntactic clone more than the semantic one, it states the methodology that has a deeper understanding of the cloning analysis granularity. They categorized all services into one of three groups: 1) DPFile (Data-processing File), which means a file directly adds, deletes, or modifies data; 2) DRFile (Data-related File), which means a file contains data queries or data beans; 3) DIFile (Data-irrelevant File), means a file that is not directly related to data. They conducted a pilot study on three benchmarks: train-ticket[1], wanxin[2], and swarm[3]. Through the quantitative analysis, they have presented that DRFiles are more likely to induce cross-service clones. They constructed a dataset with 2,722 cross-service code clones from 22 open-source projects.

The summary of the approaches discussed above and listed in Table 1 highlights the lack of focus on enterprise architecture and microservices in the field of code clone detection, particularly in regard to semantic clones. This lack of coverage represents a gap in the current literature and highlights the need for further research and attention in this area. This paper brings new insights into the field of code clone detection by addressing the shortcomings in previous studies of enterprise and microservices-based systems [9, 32]. It aims to improve semantic clone detection by reducing the training time for machine learning methods and simplifying PDG computations. The paper introduces a Component-based approach specifically tailored for enterprise and microservices systems and leverages a machine learning approach with a focus on the semantic analysis of architecture layers and their flows for more efficient results.

Relevant Comparative Analysis to Our Approach

From prior research, Zhao et al. [22] addressed microservice syntactic clones. They conducted their study on the train-ticket benchmark, which is also used in this study. To motivate the reader and justify the needs for our approach, we examine their results and compare them with our method findings as summarized in Table 2. Therefore, we identify the gap that motivates our methodology for filling it. Details of our method and the case study results presented here are provided in later sections.

We analyzed the results from Zhao et al. [22] and investigated the 201 method pairs they published at this link[4].

[1] Train-Ticket benchmark: https://github.com/FudanSELab/train-ticket, accessed on 2/5/2023.

[2] Wanxin benchmark: https://github.com/mikuhuyo/wanxin-p2p, accessed on 2/5/2023.

[3] Swarm benchmark: https://github.com/macrozheng/mall-swarm, accessed on 2/5/2023.

[4] Syntactic Clone results from [22]: https://microservicedata.github.io, accessed on 2/5/2023.

Out of the 201 clones in the compared study, none of them matched the 27 we detected manually using our approach. The compared study found many instances of syntactic clones due to duplicated switch-case statements, utility methods, unit tests, and initialization methods (i.e., constructors) among multiple services. It is common to have similar syntax and structure while they can hold different semantics. For example, the initialization of `AccountInfo` and `SecurityConfig` entities in services `ts-inside-payment-service` and `ts-security-service`, respectively, are classified as clones, but they have different data and purposes. However, with the microservice architecture introducing bounded context, it is common to find duplicated entities between services, regardless of exact fields or different fields. This results in many clones produced from encoding and decoding the same entity across services communications, such as the `Order` entity constructor appearing many times as a clone in multiple services.

Additionally, the compared study recognizes many other pairs that have the same structure but different semantics. Methods `getAllContacts` and `queryAll` in services `ts-contacts-service` and `ts-config-service`, respectively, appear similar but communicate with different services and repositories to produce lists of `Contacts` and `Config` entities, respectively. Methods `listFoodStoresByStationId` and `queryByAccountId` in services `ts-food-map-service` and `ts-consign-service` have different entity types and return data types. Methods `updateFoodOrder` and `updatePriceConfig` in services `ts-food-service` and `ts-price-service` work on different data entities but have similar syntax. However, the same `updateFoodOrder` method is also a clone with `deletePriceConfig`, but they serve different purposes for update and delete functionality. As a result, these outcomes will likely be discarded by developers and produce unnecessary effort rather than direct benefits. This could demotivate the developers to pay attention to future subsequent reports that provide a large unprioritized set of suspected smells with unobvious benefits to the system quality.

Next, we compare the results from the manual performance of our methodology to results by the Zhao et al. [22] study's results. Zhao et al. approach detected 53 cloned pairs from the benchmark's service layer, but none of them intersected with our 27 clone pairs. These 27 pairs came from 9 microservices of the same system.

Out of these 27 clone pairs, our manual classification identified *14* clone pairs between services `ts-order-other-service` and `ts-order-service` as clones, but none of them appeared in the syntactic approach. This could be due to their service methods containing long source code with additional logs and empty line differences, which are syntactic; however, they share similar semantics. The same reasoning applies to *one* clone of the `preserve` method in services `ts-preserve-service` and `ts-preserve-other-service`. Additionally, our classification identified *eight* pairs of clones between `ts-travel2-service` and `ts-travel-service`. Moreover, *three* pairs between `ts-contacts-service` and `ts-admin-basic-info-service` services were not syntactically recognized due to the usage of a REST call on one side instead of a repository method call on the other side. However, they serve the same functionality of the same entity type and match the rest of the control graph attributes. The Zhao et al. study results also could not detect the *one* clone of the `deleteConfig` method in services `ts-config-service` and `ts-admin-basic-info-service` due to different code structures that hold similar logic and persistence data on both sides.

In conclusion, the results of the syntactic analysis have confirmed that there is a significant degree of duplication between the structures of enterprise systems and the structures of the services they offer. This overlap is particularly noticeable in terms of the communication layers and the call graphs in these systems. This highlights the need for abstracting components based on their function rather than their structure, as well as the importance of a semantic approach to detecting clones in enterprise systems. By combining both the syntactic and semantic approaches, a comprehensive analysis of clones in enterprise systems can be achieved. This will provide valuable insights and help to optimize the use of developer efforts and improve the overall quality of enterprise systems.

Semantic Code Clone Methodology

Enterprise applications build on well-established development standards and components which are recognized across various platforms [33]. These types of applications usually follow a three-layer architectural design that manages data processing and storage. In addition, current trends emphasize decentralized design solutions, such as microservices, which can result in the use of multiple programming languages within a single application (a.k.a. polyglots). All these factors should be considered when attempting to impact the enterprise industry.

Our proposed Component-based Semantic Clone (CSC) detection method concentrates on the detection of semantic clones in the call graphs of enterprise applications[5], which span different layers of the application's architecture, from

[5] Note that while the examples and implementation demonstrations of our method are specific to the Java platform, it is not limited to just this platform

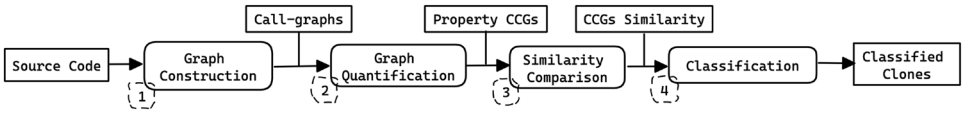


Fig. 1 Component-based Semantic Clone Detection Process Phases

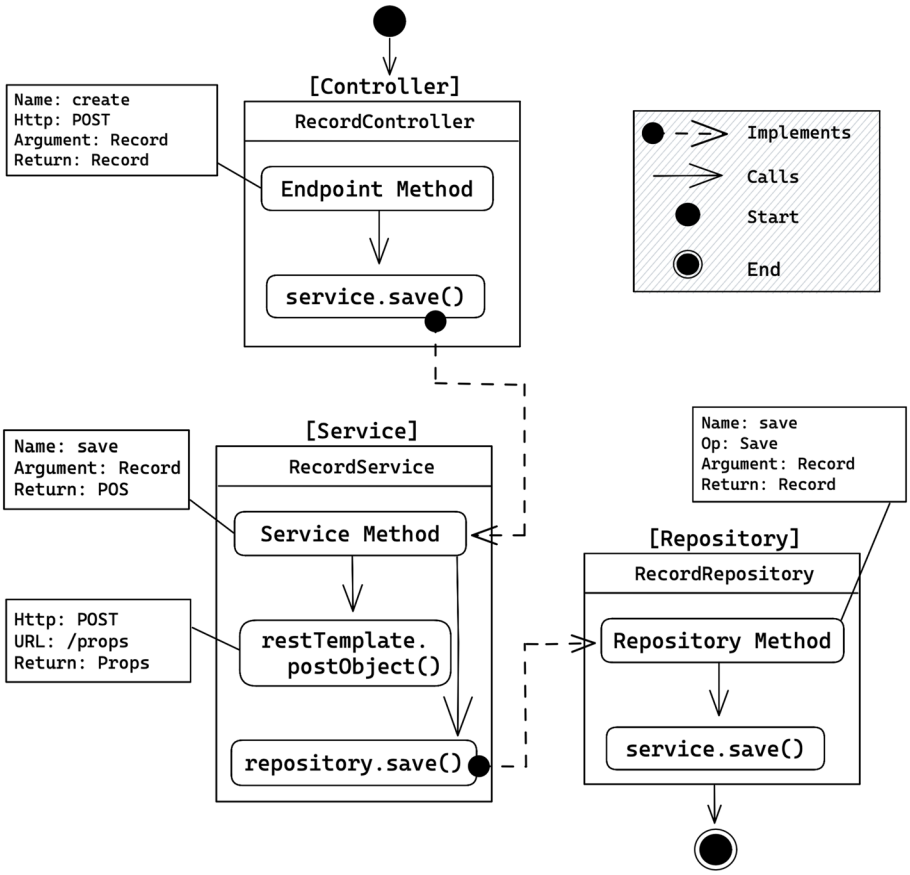


Fig. 2 The CCG of the example in Listing 1

system endpoints to data persistence. Moreover, it examines the semantics and properties of the components utilized in the source code. These components are employed as a best practice to design the application's functionality according to its relevant semantic context.

Therefore, we start the process by deriving the call graphs of the enterprise application endpoints into which we embed additional metadata about the cross-cutting components and their properties. Similar to related works, to avoid the high-cost graph matching [20, 28], we reduce the component-augmented call graph to Component Call Graph (CCG). Such intermediate representation provides a higher-level view of the system, reflecting its architectural elements and their dependencies. It shows the flow of control from one component to another, and the way components interact and depend on each other within and across the boundaries of a single microservice.

The following steps outline the method for identifying semantic clones in enterprise application call graphs. First, we construct the call graphs, then quantify their properties. Afterward, we perform a similarity comparison to determine the overlap of two compared CCGs. Finally, we use a machine learning classification model to identify the clones. The entire process is illustrated in Fig. 1 and will be discussed in further detail in the next subsections.

Graph Construction

The first phase of the CSC process is the call graph construction phase. Using static analysis, we can recognize arbitrary low-level language constructs used in the enterprise application source code. This can utilize language-specific parsers (i.e., JavaParser or Javassist [34] for the Java platform) for scanning the code and identifying all declared methods and classes within the application or specific modules of a multi-module application. These parsers also enable the identification of method calls within each method's body to build a method call graph depicting the relationship between methods.

Table 3 Component types and their properties

Component Type	Properties
Controller	Method name, HTTP method, arguments, return type
Service	Method name, arguments, return type
Repository	Database operation, arguments, return type
REST Call	URL, HTTP method, return type

The entry points of an enterprise application, usually represented as REST endpoints, are crucial to its understanding. To locate these entry points, a depth-first search finds all methods with no calling methods.

For illustration, Listing 1 shows a code example, such that there is an endpoint method `create` in the `RecordController` which calls the `save` method in the service component `RecordService`. Next, `RecordService` makes two procedure calls, first to some third-party API using `restTemplate`, and the second to `save` routine in `RecordRepository`. The resulting call graph is depicted in Fig. 2; it starts from the endpoint interface in the controller component, `RecordController`, following the arrows that move through the graph regarding each method call until it ends up with the constructing a complete graph.

Listing 1 Source code example. Note *Record* is a domain object.

```
@Controller
public class RecordController {
    @Autowired
    private RecordService service;

    @PostMapping
    public Record create(@RequestBody Record record) {
        return service.save(record);
    }
}

@Service
public class RecordService {
    @Autowired
    private RecordRepository repository ;

    public Pos save(Record record){
        Props p = restTemplate.postObject("/props");
        record.setProps(p);
        return repository.save(record);
    }
}

@Repository
public interface RecordRepository {
    Record save(Record record);
}
```

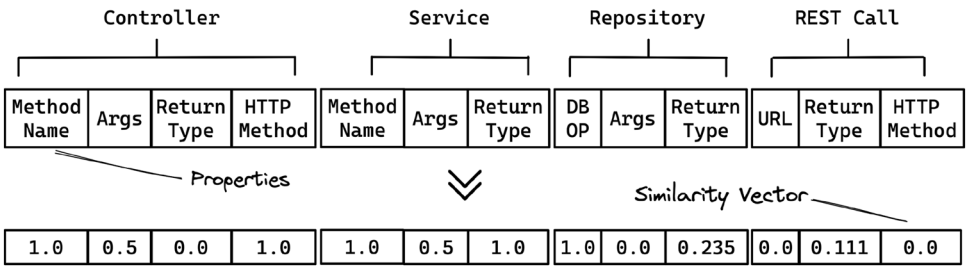


Fig. 3 Components' Properties and their Similarity Vector

Graph Quantification

After constructing call graphs, we augment the graph with cross-cutting components. The Component Call Graphs (CCG) include broader semantic details from the enterprise perspective. This involves linking each component of the CCG to its component type and its set of properties. As illustrated in Fig. 2, the component type is indicated within brackets ([]), and the properties for each component are presented in square rectangles that are connected to its component.

To annotate the CCG, we first identify the component type of each method. We search for four types of components, as outlined in Table 3. This is done by analyzing the wrapping components using common enterprise architecture practices such as annotations in Java Spring framework.

The properties that are linked to each component type are different and can be seen in Table 3. In the context of an enterprise system, these properties reflect the semantic role that the component plays in the system, such as a database connector, entry point, or service.

Language-specific parsers can be used to extract those properties that have different extraction approaches based on the source code language. For example, a controller method in Java may have a piece of metadata (i.e., annotation) for the HTTP method, which the parser can extract and link with the component type. However, a generalized approach using detection patterns can be used [33]. In summary, the added properties transform the call graph into a CCG and provide a deeper understanding of each component and its role in the system for the next phase.

Individual CCGs derived for each endpoint combine into a representation of one microservice. Furthermore, similar to the detection of components, we can detect external calls since these are realized through well-defined interfaces or constructs [33]. These REST calls can then connect to other endpoints of external microservices by method signature match, which forms a more comprehensive perspective of system dependencies.

Similarity Comparison

The similarity phase in our method involves comparing each component type in a pair of CCG to its counterpart in the other CCG. We compare the extracted properties listed in Table 3 for each component type to generate a similarity ratio as shown in Fig. 3. For example, for the controller component, we compare the method names, HTTP method, argument lists, and return types of each controller component from the CCG pair. We calculate a similarity value between 0 and 1 for each property pair. However, the comparison of properties has different considerations in calculating the similarity value.

We begin with name comparisons, which apply to method names, argument names, and custom type names properties. We utilize a project based on WordNet [35] to detect names similarity percentage based on the meaning of the name.

For the data type comparisons, which apply to arguments and return type properties. There are two cases to consider, native data types (e.g., String, int) and custom types (e.g., class, struct, DTO, entity). Our approach uses a literal comparison of native types to determine if they are the same type or not. For custom types, the microservice architecture promotes the use of bounded context, where the same domain entity can be duplicated in multiple services with slightly different names and the number of variables. Therefore, we calculate the similarity ratio between the entity name and its variables as well. Our approach extracts the type name and its contained variables from the source code, and then it compares the two types based on their names' similarity and the best matches between their contained variables. For example, if one type contains three variables and the other contains four, the method checks for the best similarities combinations between the three variables and the four variables. Moreover, the method considers the comparison of names and types of those variables, such that the same technique is used for native and custom types similarity checks. The calculated similarity value is the average of the name and all variable's similarities ratios.

For the database operation property, a different comparison logic is used. Our approach considers logical matching, meaning that the operation type should match exactly, except for insert and update operations which are considered equal since they serve a similar purpose of retaining information in databases. Some frameworks have the same method for both purposes, so it only depends on whether the item already exists in the database to decide if it's an insert or update operation. For example, the same method name "save" in the Spring framework is used for both insert and update.

Finally, the similarity of the HTTP method and URL properties are calculated as a literal comparison. The similarity of the HTTP method (e.g. GET, POST) depends on an exact match with its counterpart on the other side. The same applies to the URL, especially for microservices-based systems that commonly use a service discovery identifier instead of a literal IP address and Port. Comparing the URL base considers the invoked service rather than the deployed instance since every service could have multiple deployed instances with different IP addresses. In addition, components that make REST calls may have parameters that are included in the URL (such as Path parameters) or in the request body. To reflect this, we combined the similarity

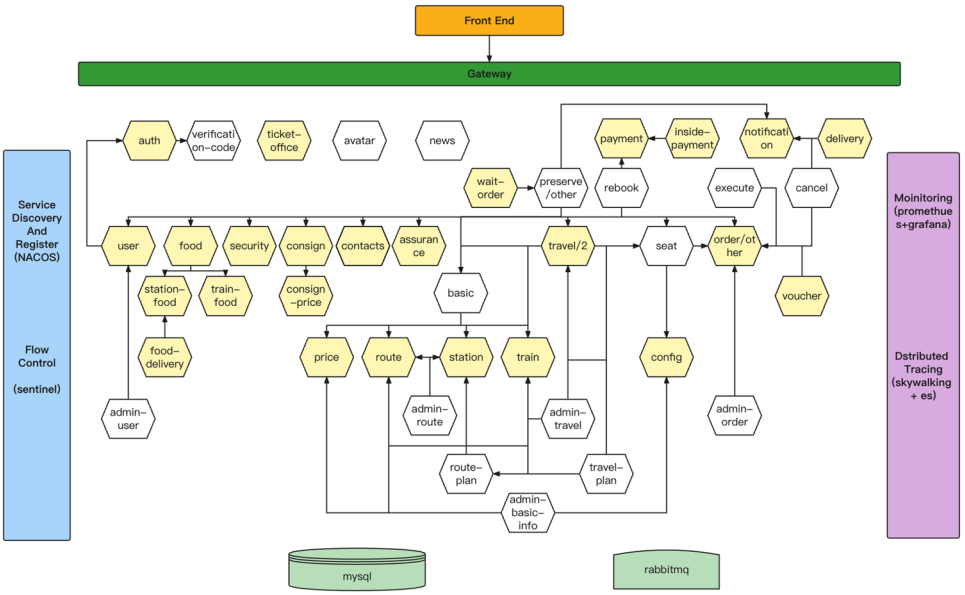


Fig. 4 Benchmark microservice overview. Sourced from the TrainTicket [1] documentation

value into the URL property as a single value that represents the complete request path.

Classification

In the previous phase, the comparison properties described generate numerical features that indicate the semantic similarity between two CCGs. These numerical features are aggregated into a similarity vector as shown in Figure 3.

Given the potential availability of labeled data for CCG pairs, we propose using supervised machine learning to predict whether two CCGs are clones based on their similarity vector. We believe these features have descriptive semantic meaning for their CCG and its components, so a simple linear model, such as a logistic regression classifier [36, 37], should be sufficient to capture the relationship.

The logistic regression model is a simple yet effective and robust approach that works well with scarce tabular data (as in this usecase). It is a parametric model that attempts to learn the following function:

$$f(\mathbf{x}) = \mathbb{P}[y = +1|\mathbf{x}) \tag{1}$$

where x is the similarity vector, and y is a binary target[6], with $y = 1$ indicating that the CCGs are clones and $y = 0$ indicating they are not.

The hypothesis function h used by the model to approximate f is given by:

$$h(\mathbf{x}) = \theta(\mathbf{w}^T\mathbf{x} + b) \tag{2}$$

where $\mathbf{w}$ is a vector of scalar weights, $b \in \mathbb{R}$ is the bias term, and θ is the logistic function defined as:

$$\theta(z) = \frac{1}{1 + \exp(-z)}; z \in \mathbb{R} \tag{3}$$

The model's weights and the bias term are adjusted using gradient descent [36] to maximize the likelihood of the training data. This means maximizing the probability that the training data came from the model.

Furthermore, a logistic regression model does not directly output a number that is 0 or 1. Instead, it outputs a continuous value in the range [0, 1], which is an estimate of the probability $\mathbb{P}[y = +1|\mathbf{x})$. This value can be interpreted as the degree of certainty the model has about a similarity vector being classified as a clone. To convert it to a binary classification, a user of the model must set a threshold that defines the range of values that indicate a positive result. This is, on input $\mathbf{x}$, the positive class is predicted if $h(\mathbf{x}) \geq thr$, for a chosen threshold value thr.

Case Study

In this section, we demonstrate the effectiveness of our proposed methodology through a case study on an existing microservice benchmark. We have developed a prototype that incorporates our methodology, consisting of both static code analysis and machine learning projects. Our study demonstrates that we can accurately detect code clones by utilizing the properties we have extracted.

[6] The domain of this binary output is usually modeled in one of two ways: $\{1, 0\}$ or $\{1, -1\}$ for positive and negative classes respectively. We assume the $\{1, 0\}$ model for the purposes of the explanation.

Benchmark Application

As our benchmark, we chose a public, mid-sized microservices application that follows enterprise conventions. TrainTicket[1] was designed as a train ticketing service that logs the application's output for fault detection. This benchmark is structured into distinct controllers, services, and repositories, and communication between the modules of the application is facilitated through REST API calls. It utilizes the Java Spring Boot framework for its implementation. An overview of the TrainTicket structure is depicted in Fig. 4. The benchmark has multiple versions with varying numbers of microservices. In this study, we examine three of these versions, labeled 0.0.1, 0.1.0, and 1.0.0, which contain 41, 37, and 29 microservices, respectively. We use them to identify semantic code clones in the application.

Prototype Implementation

The implementation of the prototype is split into two parts. The first project primarily focuses on analyzing the source code of enterprise Java applications, while the second project implements a machine learning model to identify semantic clones from the data generated by the first project. The source code for the prototype is publicly available at this link[7].

For the static analysis project, we use Java-specific tools to extract and construct quantified CCGs from the source code using the Java Reflection library and Javassist [34]. The focus is on the Java Spring framework, which uses annotations for component implementations, such as `@Controller` and `@RestController` for controllers, and `@Repository` and `@Service` for repositories and services. We use the WS4J project[8] - and WordNet [35] to determine the similarity percentage based on the meaning of the names. The properties of each component are extracted and associated, and the similarity between each pair is calculated as described in section 4.3.

For the machine learning project, we use logistic regression from the scikit-learn [38] library in Python to build our classification model. It operates on the data generated from the static analysis project, classifying the CCG pairs into clones or non-clones, and calculates the accuracy metrics for detecting clones from the testing project.

Manual Analysis

To construct the ground truth for semantic clones, we executed our prototype on the TrainTicket benchmark (release 0.1.0) and obtained 238 CCGs. These CCGs resulted in 27,221 pairs of combinations. We investigated and assessed the source code of these components, not just the extracted properties that are discussed and shown in Table 3. Two authors evaluated the semantic similarity of these pairs whether they are clones or not based on the components' semantics of each CCG (i.e., Controller, Service, Repository, REST Call), and one more author validated the classification result.

Our focus was on semantic meaning rather than syntactic similarities. This means that the two call-graphs have the same logic and similar impact on the system, regardless of whether they employ the same syntax or not. For example, a syntactic analysis may identify clones where the CCG components have the same syntax structure, but handle different entity types. However, in our approach, this does not mean that they are clones, but it could be a sign of non-clones classification. On the other hand, Two semantically matching service components that retrieve similar entity data, one by calling a repository and the other by calling a Rest service, can be considered a semantic clone since they retain logic with a difference in the source of data. Similarly, when the logic of all components matches in the CCG pair, we label them as a clone. As a result, we categorized 27 pairs as clones, while the rest were labeled as non-clones. This led to an imbalanced dataset that we will consider in our processing. The classified dataset can be accessed at this link[9].

Measurement Metrics

To evaluate the effectiveness of our implementation, we selected certain metrics as a means of assessment. Utilizing the labeled data created through the manual analysis, our regression model produces a compilation of the following values:

- True Negative (TN): Non-clone examples correctly predicted as non-clones.
- False Positive (FP): Non-clone examples incorrectly predicted as clones.
- False Negative (FN): Clone examples incorrectly predicted as non-clones.
- True Positive (TP): Clone examples correctly predicted as clones.

[7] Our Prototype:https://github.com/cloudhubs/Distributed-Systems-Semantic-Clone-Detector, accessed on 2/5/2023.

[8] WS4J: https://github.com/Sciss/ws4j, accessed on 2/5/2023.

[9] Our Semantic Clone Dataset (V1): https://zenodo.org/record/7632839, accessed on 2/11/2023.

Table 4 Classification metrics on the training dataset

Accuracy	Balanced Accuracy	Precision	Recall	F_1 Score
0.999	0.944	0.75	0.888	0.813

Table 5 The similarity properties and their assigned weights of the model

Property	Weight
Controller: method name	**9.06608**
Controller: HTTP method	1.38333
Controller: arguments	1.86721
Controller: return type	**4.39439**
Service: method name	1.87319
Service: arguments	0.28254
Service: return type	**4.02174**
Repository: database operation	− 1.56188
Repository: arguments	− 2.21971
Repository: return type	2.91226
REST Call: URL	− 0.58029
REST Call: HTTP method	1.24877
REST Call: return type	0.00397

Bolder are the weights with the highest absolute values, i.e., the weights contributing more to the output, which could be interpreted as the features that are more important

We utilize these values to compute valuable metrics to assess the quality of a binary classification model. Each of the following metrics reflect a valuable assessment of the model effectiveness: accuracy, balanced accuracy, precision, recall, and F_1 score [39, 40].

The model's accuracy reflects the share of correct classifications the model makes. It is defined as:

$$A = \frac{TP + TN}{TP + FN + TN + FP} \tag{4}$$

The balanced accuracy is somewhat similar to the accuracy. However, it is particularly useful in our case when working with an imbalanced dataset because it operates in such a way that correctly predicting an instance of an undersampled class contributes more to the final score than correctly predicting an instance of an oversampled one. It is defined as follows:

$$BA = \frac{1}{2}\left(\frac{TP}{TP + FN} + \frac{TN}{TN + FP}\right) \tag{5}$$

The precision, a.k.a. positive predictive value (PPV), indicates in this domain the share of the model's predictions of clones that are true clones. It is defined as:

Table 6 CSC vs. Manual Analysis: Comparison of Clone Detection

Technique	#clones	#cloned μs
Manual Analysis	27	9 μs
CSC	32	14 μs

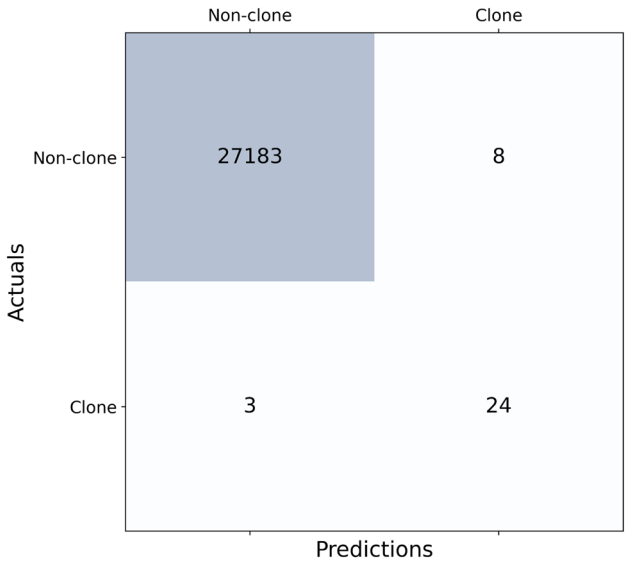


Fig. 5 Confusion matrix of the model's predictions on the training dataset.A confusion matrix is a compilation of 4 different values, each in a cell of the matrix. From left to right, top to bottom, these are: TN, FP, FN, and TP

$$P = \frac{TP}{TP + FP} \tag{6}$$

The recall, a.k.a. true positive rate (TPR), indicates in this domain the share of the actual clones that are classified as clones by the model. It is defined as:

$$R = \frac{TP}{TP + FN} \tag{7}$$

Finally, the F_1 score measures how well the model is doing considering both precision and recall simultaneously. Formally, it is defined as the harmonic mean of precision and recall:

$$F_1 = 2\left(\frac{P \cdot R}{P + R}\right) \tag{8}$$

Study Execution

The evaluation of our methodology starts with running the prototype on the TrainTicket benchmark (release 0.1.0) and generating 238 CCGs, which resulted in 27,221 combinations. These data points were then manually labeled,

Table 7 TrainTicket microservices' clones analysis (μs: Microservice, MA: Manual Analysis)

#	$\mu s1$	#CCGs	$\mu s2$	#CCGs	#clones (MA)	#clones (CSC)
1	ts-contacts-service	8	ts-admin-basic-info-service	21	3 **	1
2	ts-config-service	6	ts-admin-basic-info-service	21	1 **	–
3	ts-order-other-service	16	ts-order-service	16	14	15 *
4	ts-preserve-service	2	ts-preserve-other-service	2	1	1
5	ts-travel2-service	12	ts-travel-service	12	8	11 *
6	ts-route-service	6	ts-admin-route-service	4	–	1 *
7	ts-rebook-service	3	ts-inside-payment-service	9	–	1 *
8	ts-basic-service	3	ts-ticketinfo-service	3	–	2 *

The 3 false negatives were actual clones, they are found in the two pairs marked with (**) in the table at lines 1 and 2. When examining the 8 false positives, it was found that 6 of them were actual clones which we missed in the manual analysis. These are found in the pairs marked by (*) in rows 3, 5, 6, 7, and 8 of the table

as outlined in the section "Prototype Implementation". A logistic regression model was fitted using these 27,221 data points.

Due to the imbalance in the dataset, a weighted version of the logistic regression model was employed to give more weight to classifying the undersampled class correctly and less weight to the oversampled class. The threshold for prediction was set at 0.99, as it maximized the F_1 score in the training data. This threshold value, referred to as a hyperparameter in machine learning, is typically found using cross-validation techniques [41], however, due to limited data, especially limited clones samples in the dataset, cross-validation was not performed in this case study.

Metrics Results Analysis

A summary of the model's performance in the training data in terms of accuracy, balanced accuracy, precision, recall, and F_1 score is given in Table 4.

As an analysis of the measurements, we found that the model has a very high accuracy rate, with the majority of examples being correctly classified. However, this value is misleading due to the imbalance of the dataset. The balanced accuracy, which compensates for the imbalance, provides a more accurate representation of the model's performance and it stands at 94.4%. A precision percentage of 75% suggests that out of every 4 examples that the model predicts as clones, 1 will actually be a non-clone. On the other hand, the high recall indicates that the model is highly effective at identifying the existing clones in the data. As the recall value is less than 1, it is anticipated that the model may classify some real clones as non-clones. Finally, the F_1 score, which combines precision and recall into a single metric, supports the aforementioned results.

Moreover, the logistic regression model produces weight values that reflect the most important features considered in making predictions. The results of these weights are displayed in Table 5. Upon analysis of the weights, it was observed that the controller method name and return type, and the service return type were the most significant factors in the model's predictions.

Table 8 Semnatic Clones Dataset for TrainTicket (0.1.0)

Version	#clones (CCGs Pairs)	#non-cloned (CCGs Pairs)
V1[9]	27	27,194
V2[1]0	33	27,188

Table 9 TrainTicket releases' clones analysis

Train-Ticket Release	#clones	# services	#cloned services	#CCGs	#cloned CCGs
0.0.1	36	41μs	18μs	250	71
0.1.0	32	37μs	14μs	238	64
1.0.0	30	29μs	12μs	257	60

Clones Results Analysis

The results of the semantic clone classification by the model are presented in Table 6. This table demonstrates that the model detects a larger number of clones and a greater number of microservices containing clones than were classified in the labeled training data.

A confusion matrix in Figure 5 provides a detailed evaluation of the model's predictions on the training dataset. The matrix shows that the model has a high level of precision in recognizing clones, with 24 of the 27 total clones in the dataset being correctly classified. Additionally, the model has a high level of accuracy in classifying the vast majority of non-clones, only misclassifying 8 pairs of them.

The cloned services are extracted and listed in Table 7. This table shows the pairs of microservices along with

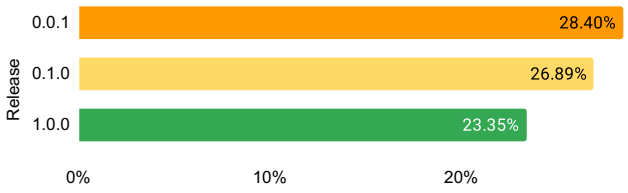


Fig. 6 TrainTicket CCGs clones percentage

their total number of CCGs and how many were classified as clones by both manual analysis and the model. Further analysis was conducted on the 11 misclassified pairs, which included 8 false positives and 3 false negatives.

When examining the 8 false positives, **it was found that 6 of them were actual clones which we missed in the manual analysis**. This is marked by (*) in rows 3, 5, 6, 7, and 8 of the table. However, two of them were found to be correct non-clones after a manual investigation. These were the pair between `ts-rebook-service` and `ts-inside-payment-service` at row 7, and one of the two CCGs at row 8 between `ts-basic-service` and `ts-ticketinfo-service`. This shows that **the model has a promising ability to learn the properties of components and identify faults in the training dataset created through manual analysis.**

On the other hand, the 3 false negatives marked with (**) in the table were actual clones. This may have occurred because these 3 CCGs pairs have similar properties, where one calls a repository routine, and the other invokes a Rest service to achieve the same functionality. As analyzed, `ts-admin-basic-info-service` calls a Rest service to delete the contact, while `ts-contacts-service` calls a repository method to delete the contact. Similarly, the call-graph for modifying contact functionality. The same case exists between `ts-config-service` and `ts-admin-basic-info-service` for deleting configuration. This highlights the need for the model to be trained with cross-similarity values between repository components and REST call components, as they both serve as data sources for the other components in the call-graph.

Semantic Clone Dataset

In this study, a new Semantic Clone Dataset was created and verified through manual analysis and the results of our method. This dataset provides a valuable resource for the community in improving semantic clone detection and it contains 27,222 pairs of CCGs.

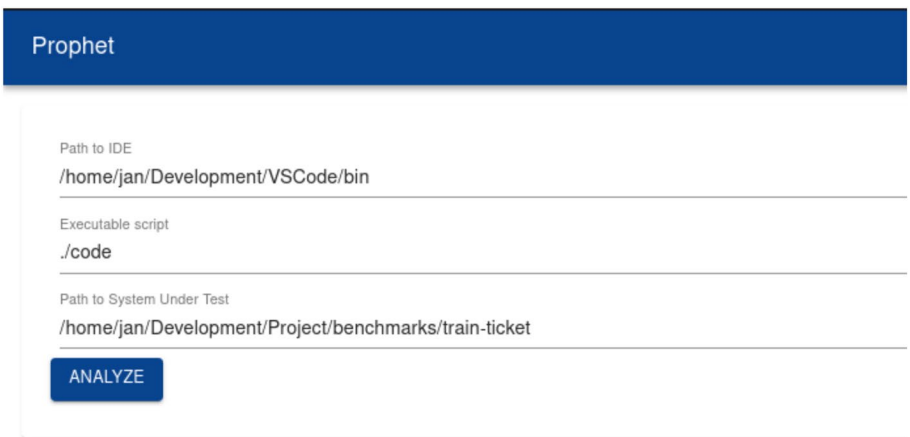
The dataset includes 16 columns, with 13 of them corresponding to the 13 properties extracted and listed in Table 3. Two columns contain the pair of CCGs, and the final column provides a label indicating whether the pair is a clone or a non-clone.

To make the dataset accessible and useful, we have published two versions of the dataset, as described in Table 8. The first version (V1[9]) was classified manually and used to train our model. The second version (V2[10]) includes an additional 6 clones that were detected using our CSC method and manually verified; these are not present in the first dataset version. These extra clones are marked in the dataset for easy tracking and were explained in previous sections of the study.

Tracing Clones through System Evolution

The analysis of semantic code clones while the system evolves over time is an interesting use case. This can be achieved by fitting a model to a specific version of a system and then evaluating the clone prevalence in a different

Fig. 7 Configuration Page



[10] Our Semantic Clone Dataset (V2): https://zenodo.org/record/7632842, accessed on 2/11/2023.

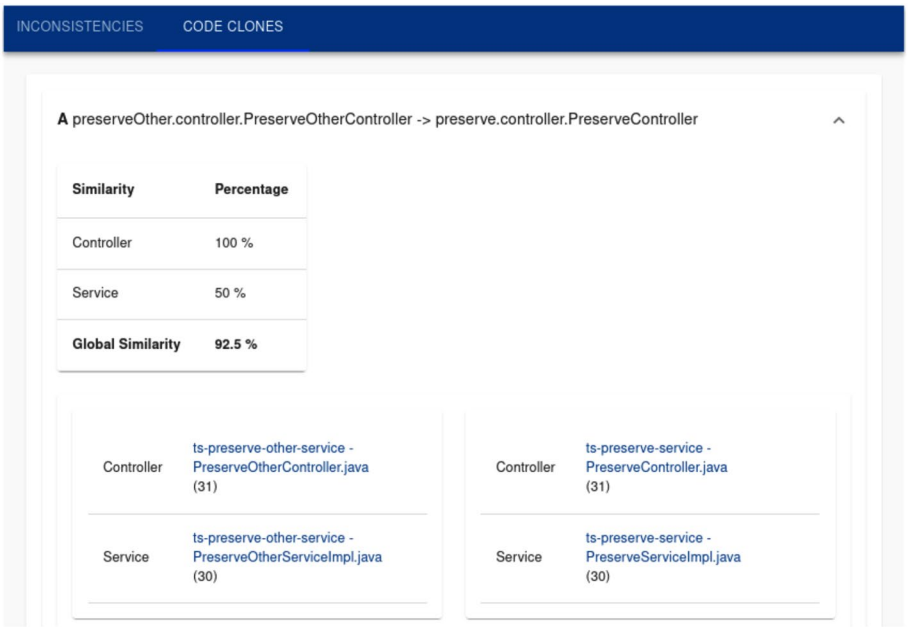


Fig. 8 The first two sections of Clone Page

version. This approach assumes that the weights obtained from fitting the model are dependent on the system but not time-dependent, meaning the model generalizes well to different versions of the same system.

The TrainTicket system was analyzed using our framework to compare two versions of the system, an older version (0.0.1) and a newer version (1.0.0), with the version used for training (0.1.0). According to Table 9, the number of microservices decreases as the system evolves, likely due to merging services to reduce communication overhead. The analysis reveals that the overall number of clones, the number of services with clones, and the number of cloned CCGs all decrease over time, with the newest version having the fewest clones in terms of services and CCGs, even though it has the highest number of CCGs among the different versions. The percentage of CCGs clones per each release is illustrated in Fig. 6. This demonstrates that the system is being refactored and kept clean by reducing the number of clones in previous versions.

In addition, the older version (0.0.1) introduced 11,558 new CCG pairs not seen in the training version. Our model classified 8 of these pairs as clones, and after investigation, it was found that 5 of the 8 were correct. This highlights the stability of the model in detecting clones even for out-of-sample pairs. The newer version (1.0.0) generated 7,132 new CCG pairs, all of which were found to be non-clones.

Integration to the Development Process

Code clones can lead to several problems in the codebase (or distributed codebases) if left unaddressed by developers. Moreover, developers will likely not run reports with each change to the application. While there are tools such as SonarQube[11] which can catch Type 1 of syntactic clones during the CI/CD process and force developers to address them before merging their code, nothing like that exists for semantic code clone detection.

We address the developer interaction through two communication channels tapping into the SDLC. One perspective we address is integration with Integrated Development Environment (IDE). The other perspective is integration with the CI/CD.

IDE Interactive Tool

To allow developers to quickly assess semantic code clones in their application using graphical user interfaces we integrated our prototype tool with a selected Integrated Development Environment (IDE). This integration allows developers to localize and remove code clones from their codebase(s). We published this tool[12] - as open source. It consists of a Configuration Page and Code Clone Page.

[11] SonarQube: https://www.sonarqube.org, accessed on 2/5/2023.

[12] Our Interactive Tool: https://github.com/svacina/prophet, accessed on 2/5/2023.

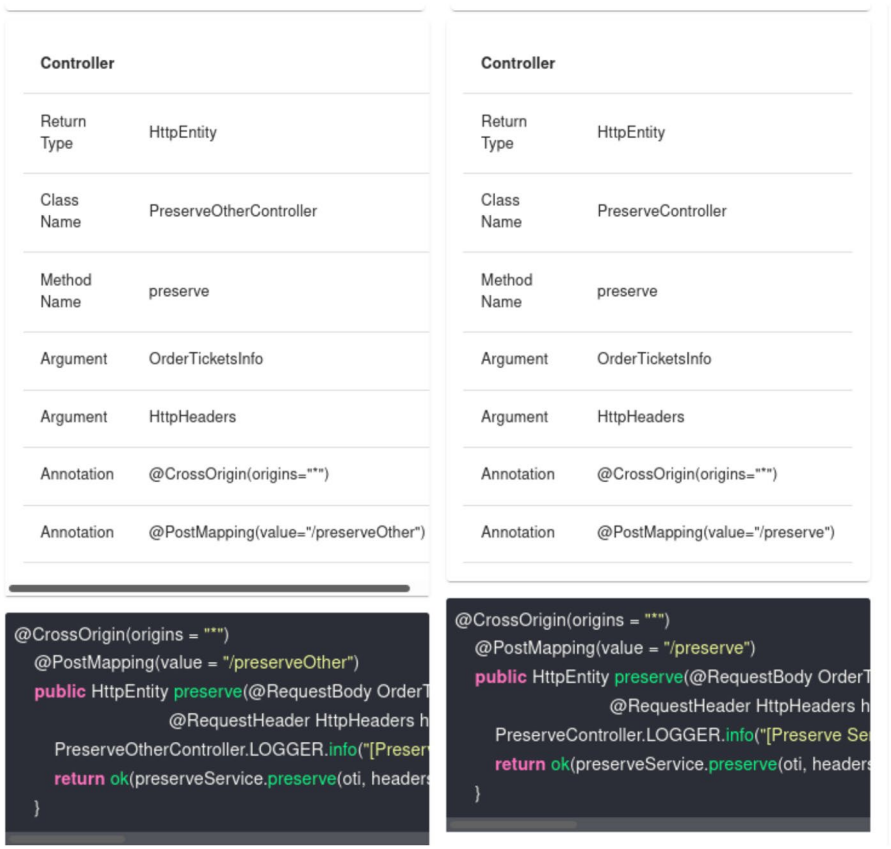


Fig. 9 The third section of Clone Page

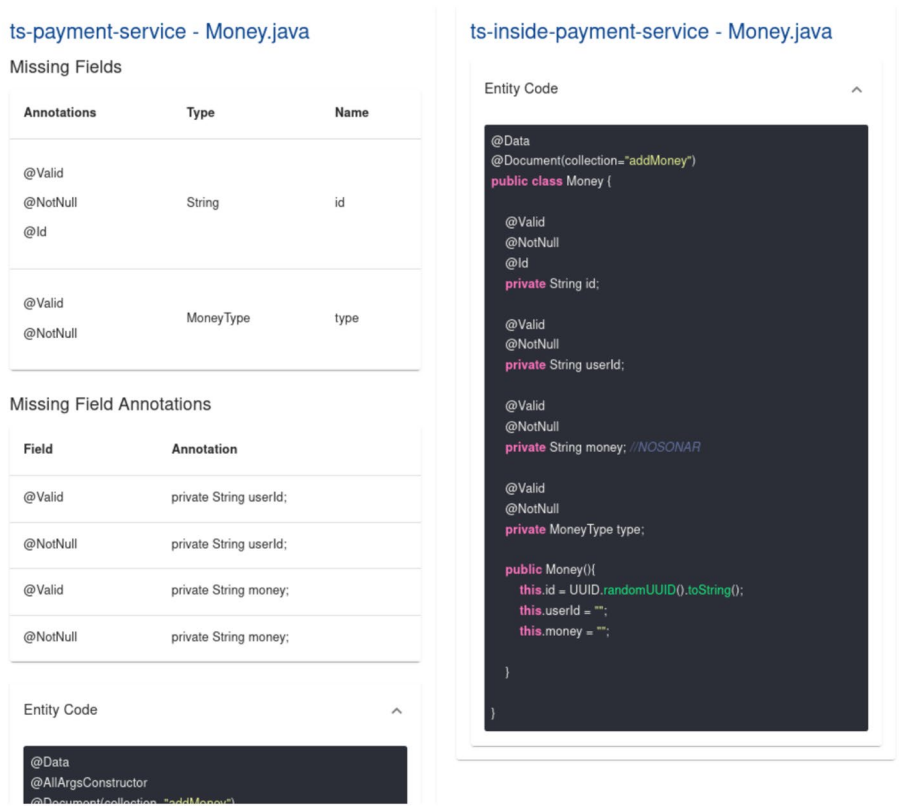


Fig. 10 Entity Inconsistencies Page

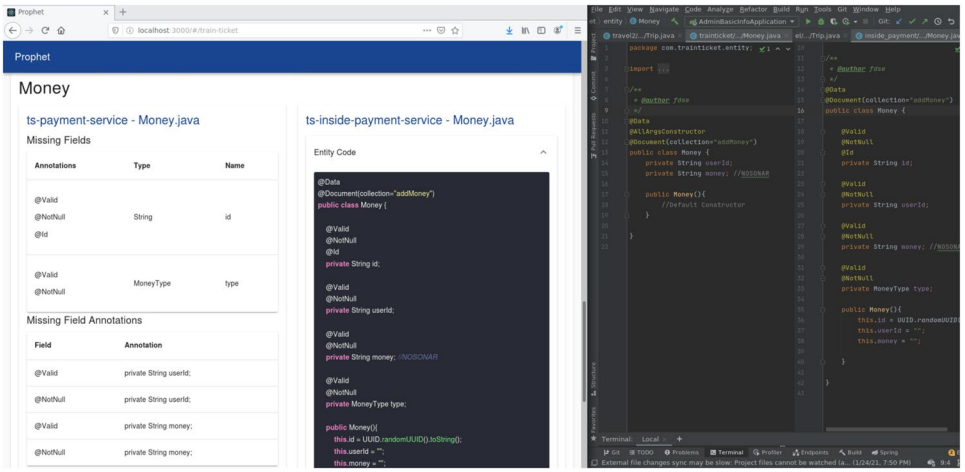


Fig. 11 VSCode IDE Integration

Fig. 12 Gradle plugin output

The developer configures essential properties, as shown in Fig. 7. The path to the system on the local file system. We assume that the developer downloads the system and has at least read access to the files. Next, a path to a script that can open the IDE. We assume that the developer configures IDE and has to execute access to the IDE. We configured our example with TrainTicket[1] - and VSCode IDE[13].

The code clone page is displayed with a list of code clones. Each code clone component has three sections. The first section is a table with an overview of how similar are the parts of one CCG with the other. The table, as shown in Fig. 8 has two columns. The first column lists all parts of the CCGs, controller, services, repositories, and the REST call. The second column shows the percentages of similarity, which are calculated based on each CCGs components similarity, and also each component similarity is considered as an average of the similarity between each component-related properties. For example, the controller similarity is the average of the following properties: method name, arguments, return type, and HTTP method as listed prior in Table 3. This section enables users to quickly assess which parts of the CCGs are most likely to include the same behavior. The second section displays an identification of a code clone pair, showing the percentage of similarity to stipulate code clone severity between the CCGs components. The third section is a two-column view of both CCGs side by side as shown in Fig. 8. This layout enables users to see the same parts of each CCG next to each other and enhances the overall ability to compare related parts. It displays a comparison of CCGs properties together with a code of methods that participate in the CCGs (Fig. 9). Each part of CCGs has its section (controller, service, repository, REST call). Each part contains a table that displays relevant extracted properties as the following properties: class name of the parent class, method name, return type, arguments, and annotations. We also included a code snippet with the actual code of the method per each CCG part.

In addition to code clones, our tools also provide a quick assessment for data entity layer inconsistencies across decentralized microservices. It layouts visual verification of the missing fields between data entities. As shown in Fig. 10, there is an entity object `Money` that has inconsistent usage in two microservices, ts-payment-service, and ts-inside-payment-service. Entity Money is missing two fields with their annotations and four validation annotations on two other fields. The boxes are displayed with missing fields and annotations for entity Money on the left next to the code of the complete entity on the right.

[13] VSCode IDE: https://code.visualstudio.com, accessed on 2/5/2023.

Furthermore, the names of individual microservices are represented as active links that open the file with code in IDE (i.e., VSCode[13]) as depicted in Fig. 11. Users can click on links to files of the analyzed system to use an external IDE to make changes in the code to remove inconsistencies and code clones.

CI/CD Process Integration

To support the enforcement of our prototype tool usage across large teams we also developed a Gradle-plugin[14] - to scan the code against the clones and inconsistencies. Applying our plugin to an application project allows the plugin to extend the project's capabilities. Following the installation guide in the repository, it can be integrated with existing CI/CD pipelines and automatically warn developers of any semantic code clones in their application. Upon execution, it prints out in the IDE console semantic analysis results. It shows the cloning CCGs analyzed class paths, then the percentage of the matching according to their components similarities, as depicted in Fig. 12.

This Gradle plugin leverages the semantic code-clone detection on the root of the project that holds each of the microservices or only on the module one likes to analyze. The benefit of using an established standard like Gradle for the plugin is the wide configuration options available. The plugin also outputs the results to a series of files in a user-defined output directory.

Discussion

There are multiple perspectives related to the presented CSC method that deserve further discussion. First, this method has been evaluated on enterprise system middleware, not taking into account the user interface part. However, even a complete system with a user interface would typically proceed with interaction through application endpoints. The user interface portion would remain not considered by this method.

The CSC method provides promising results on the considered system benchmark. However, it needs broader assessments across other system benchmarks to see the implication in a broader context. In addition, it did not detect all semantic clones and this may present a limitation to identifying only a certain class of semantic clones in enterprise systems that are not yet categorized and our research may contribute to their taxonomy. Moreover, the practical value of such detection needs to be further evaluated. On the other hand, it must be emphasized that the detection is very quick compared to other approaches mentioned in the literature.

This manuscript illustrates weight calibration on a single established microservice benchmark version. In this direction, developers would manually assess their current application for semantic code clones to strengthen the ground truth to establish appropriate weights for their application. This might be influenced by the used framework and programming conventions for utilizing the components' properties. However, this should be seen as the most pessimistic perspective. Once established weights on a large system benchmark will likely be transferable across projects building on the top of the same development framework. However, to make such a claim we are missing data and experiments to prove it, as well as a proper system, and benchmarks to evaluate it which remains for future work.

On the other hand, the CSC method showed impressive results, as indicated in Table 4 and Fig. 5. The analysis demonstrated the ability of the method to detect 6 extra clones that were not present in the training dataset. This also illustrates that if the initial calibration was necessary to be used for a particular system, the method is robust to an imperfectly labeled dataset resulting in promising outcomes. Additionally, the system evolution analysis showed that the method can be generalized to different versions of the same system, performing well on unseen data.

In order to evaluate the method's applicability to application polyglots there are multiple steps to be accomplished. First, we must implement parsers and component detectors for other platform development frameworks to build CCG for applications coded in these frameworks (i.e., Django, C#, etc.). While preliminary work was performed by Schiewe et al. [33], it still does not build CCG. Second, experimentation must be performed to identify proper component attributes and calibrate weights for the method. Thirds, the same component types across platforms must be assessed to identify equivalent component properties and then analyze conversions in equivalence across platforms. These steps present a plethora of future work. However, there are no known limits to the method rather than efforts to perform this.

Furthermore, it could be considered to include CFG instead of CG in the method and recognize fine details like conditionals or loops. These are currently reduced to a single path, possibly influencing to false-positive clone detection. The method could thus extend and associate multiple CFGs with each current CCG. The similarity evaluation would match each of these CFGs across the assessed CCG pairs.

[14] Our Plugin: https://github.com/cloudhubs/prophet-gradle-plugin, accessed on 2/5/2023.

Threats to Validity

Using Wohlin's taxonomy [42], we examined four types of validity threats: external validity, concept validity, internal validity, and conclusion validity. We also elaborate on our custom thresholds used in the proposed method.

Construct Validity

Construct validity is the degree to which the measures correctly reflect real-world constructs. In our assessment, we used a third-party enterprise system benchmark that is based on microservices and utilizes a well-established development framework utilizing up-to-date development standards and components. This benchmark is used by the microservice community to demonstrate their research outcomes. This represents realistic conditions under which our approach operates. At the same time, there are numerous other frameworks that could be used and yield alternative results, which illustrate that our results must be considered in such a context. Furthermore, the selected benchmark may introduce a biased representation of coding practice; however, we negated this by manual analysis.

Our method considers components like controllers, services, repositories, and entities that are recognized across enterprise platforms. In addition, we considered endpoints and REST calls to these endpoints from other microservices, but we omitted messaging systems in our assessment. This implies that our method is unsuitable for general programs that do not consider components (i.e., Java Standard Edition).

Furthermore, it must be recognized that there is no clone benchmark using enterprise constructs for semantic clone detection with an established ground truth; however, we used the same system applied by related work when searching syntactic code clones, and the outcomes of our work bring such a benchmark for community use.

Internal Validity

Internal validity challenges the study methods and data analysis; this involves experiment errors and bias. Our method uses several weights for semantic clone detection. These are implied from manually labeled data which could be biased. To reduce the bias, multiple authors were involved in labeling the data. Next, we used logistic regression over the component attribute values in the labeled dataset to produce an optimal solution for our labeled dataset. Yet, upon evaluation, our model produced false positives for the manually labeled dataset; however, upon careful evaluation, multiple of these proved to be true positives, and our manually labeled dataset did not prove to be ground truth. Also, from the perspective of Machine Learning, we cannot ensure the weights obtained generalize to out-of-sample data because we did not have data to validate the generalization capabilities of the model.

A non-standard development practice for enterprise application development can influence the accuracy of our method. We have optimized the method to sustain the wrong system cuts in a way if a component in the component call graph is missing in compared pair, the component is not considered. This perspective is relevant to the evaluated system of version 0.0.1.

Our experiment only included a single system, and to strengthen our conclusions, many other systems across platforms need to be tested. It remains to be determined whether weights transfer across systems or across systems using the same platforms.

Furthermore, it is possible to perform a database delete through a *GET* endpoint call, where standard practice is to use a *DELETE* endpoint call. If two methods with different HTTP types perform the same operation, their similarity might be impacted, and our method could produce false negatives. Similarly, two different methods with the same name, parameters, and HTTP type can perform entirely different operations. Our method may produce false positives. However, this indicates poor coding practice; these situations require more evaluation.

External Validity

These threats concern the generalization of our results. Given that enterprise systems use component-based development, using components that realize enterprise standards across platforms, the approach generalizes well to these systems.

However, the main threat is the weight setting for component properties used for similarity comparison. To calibrate the weights, we conducted the tests on the TrainTicket[1], a community-established system benchmark that follows enterprise standards. We used machine learning to make a statistically informed decision on how to set the weights.

We focused on the Java platform in this prototyping because of its strong presence in the enterprise domain. Our case study demonstrates that we can successfully analyze such a Java-based system following enterprise standards. We, however, cannot conclude that in the same setting, we can transfer the method across platforms or even across the different systems using the same platforms as it was not tested.

To mitigate the limitation of assessing the method at a single system was addressed by considering different versions of this system, which are significantly different (41 microservices down to 29). Testing other systems is planned

in future work, and assessment for different or across domains requires the development of component call graph parsers across these domains. However, the system intermediate representation format is already defined for this aim by this work. Despite the method target to microservices, this approach can be used in monolithic systems as well.

On the other hand, in case an enterprise application would not use standard components, then it would likely not follow best practices. As a result, for the method to work, it would require modifications to consider application-specific naming conventions of used modules across architectural layers.

Conclusion Validity

The threats concern whether the conclusions are based on the available data. Our work aimed at proof-of-concept of a method capable of addressing problems not yet solved in enterprise systems. It did not aim at statistically significant conclusions but rather to produce preliminary data and draw possible implications for the community. As a secondary result, it also produced a semantic clone dataset necessary for future research advancements and experimentation.

Our finding's reliability might have been compromised by the limited variation in the considered system benchmark. At the same time, our conclusions are limited to what we observed in the case study. There is no reason to believe that it is not more generally applicable to other enterprise systems and platforms; however, this has not been empirically assessed. We share our method as open-source, and so are the different versions of the used TrainTicket benchmark[1], which facilitates replication of the study.

Conclusion

Semantic code clones pose a significant challenge in enterprise systems, as it has not yet been widely addressed. The CSC method, proposed in this manuscript, targets microservices as the mainstream architecture for enterprise applications, attracting both industrial and academic interests. By considering an established enterprise system standards and components, the use of CCG extracted from these systems is proposed as a means to analyze semantic code clones. It's noteworthy that this approach is relatively inexpensive compared to existing methods, such as those involving PDGs, as CCG carries high-level semantic information that might not be easily extracted from low-level code analysis without recognizing standard enterprise constructs through established component types. The abstraction of CCG also reduces the number of pairs to be compared and focuses on a specific number of features, resulting in a machine-learning model that fits the data more efficiently.

The proposed CSC method has shown promising results with high accuracy when assessed on a third-party system benchmark and exhibits robustness to an imperfectly labeled dataset. It was also demonstrated to be usable across system evolution, with calibration being a one-time task. Although it is believed that such calibration can be transferred across different systems (i.e., the same platform), future work will need to demonstrate such a perspective.

In addition to its practical applications, this work provides several outcomes that could benefit the community. Two versions of datasets containing classified semantic clones are published publicly, along with a visualization tool that presents the code clones in a user-friendly format. Furthermore, the method is implemented as a plugin that can be integrated with development tools to detect semantic clones during the development lifecycle.

Future work will consider polyglot systems. It will also consider using CFG within identified clone candidates to better cope with false positives while still offering a quick turnaround. From a short-term perspective, we plan to replicate the experiment on a well-established C# system benchmark, which we currently statically analyze. Thus, we can experiment one more step towards the model generalization over the different platforms.

Acknowledgements This material is based upon work supported by the National Science Foundation under Grant No. 1854049 and a grant from Red Hat Research.

Data availability The dataset generated in this work is available at https://doi.org/10.5281/zenodo.7632839 and https://doi.org/10.5281/zenodo.7632842. Our prototype tools are available at GitHub as open source: Semantic Clone Detector: https://github.com/cloudhubs/Distributed-Systems-Semantic-Clone-Detector Gradle Plugin: https://github.com/cloudhubs/prophet-gradle-plugin, Interactive Tool: https://github.com/svacina/prophet.

Declarations

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

1. Besker T, Martini A, Bosch J. Technical debt cripples software developer productivity: A longitudinal study on developers' daily software development work. In: Proceedings of the 2018 International Conference on Technical Debt. TechDebt '2018:18,105–114; Association for Computing Machinery, New York, NY, USA . https://doi.org/10.1145/3194164.3194178
2. Ain QU, Butt WH, Anwar MW, Azam F, Maqbool B. A systematic review on code clone detection. IEEE Access. 2019;7:86121–44.
3. Baker BS. On finding duplication and near-duplication in large software systems. In: Proceedings of 2nd working conference on reverse engineering, 1995;86–95 https://doi.org/10.1109/WCRE.1995.514697. IEEE
4. Ducasse S, Rieger M, Demeyer S. A language independent approach for detecting duplicated code. In: Proceedings

IEEE international conference on software maintenance-1999 (ICSM'99).'Software maintenance for business change'(Cat. No. 99CB36360), 1999;109–118 . https://doi.org/10.1109/ICSM.1999.792593. IEEE
5. Higo Y, Kusumoto S, Inoue K. A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. J Softw Maint Evol: Res Pract. 2008;20(6):435–61. https://doi.org/10.1002/smr.394.
6. Kumar A, Yadav R, Kumar K. A systematic review of semantic clone detection techniques in software systems. In: IOP conference series: materials science and engineering, 2021;1022:012074 https://doi.org/10.1088/1757-899X/1022/1/012074. IOP Publishing
7. Vislavski, T., Rakić, G., Cardozo, N., Budimac, Z.: Licca: A tool for cross-language clone detection. In: 2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER), pp. 512–516 (2018). https://doi.org/10.1109/SANER.2018.8330250. IEEE
8. Saini V, Farmahinifarahani F, Lu Y, Baldi P, Lopes CV.: Oreo: Detection of clones in the twilight zone. In: Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, pp. 2018;354–365 https://doi.org/10.5281/zenodo.1317760
9. Svacina J, Bushong V, Das D, Cernỳ, T. Semantic code clone detection method for distributed enterprise systems. In: CLOSER, pp. 27–37 (2022). https://doi.org/10.5220/0011032200003200
10. Roy CK, Cordy JR. A survey on software clone detection research. Queen'Sch Comput TR. 2007;541(115):64–8.
11. Svajlenko J, Roy CK Evaluating clone detection tools with bigclonebench. In: 2015 IEEE international conference on software maintenance and evolution (ICSME), pp. 131–140 (2015). https://doi.org/10.1109/ICSM.2015.7332459. IEEE
12. Nasirloo H, Azimzadeh F Semantic code clone detection using abstract memory states and program dependency graphs. In: 2018 4th international conference on web research (ICWR) 2018:19–27 https://doi.org/10.1109/ICWR.2018.8387232. IEEE
13. Wu, Y., Zou, D., Dou, S., Yang, S., Yang, W., Cheng, F., Liang, H., Jin, H.: Scdetector: software functional clone detection based on semantic tokens analysis. In: Proceedings of the 35th IEEE/ACM international conference on automated software engineering, pp. 821–833 (2020). https://doi.org/10.1145/3324884.3416562
14. Vislavski T, Rakić G, Cardozo N, Budimac Z. Licca: A tool for cross-language clone detection. In: 2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER), 2018;512–516 https://doi.org/10.1109/SANER.2018.8330250. IEEE
15. Alomari HW, Stephan M. Clone detection through srcclone: a program slicing based approach. J Syst Softw. 2022;184: 111115. https://doi.org/10.1016/j.jss.2021.111115.
16. Juergens E, Deissenboeck F, Hummel B Code similarities beyond copy & paste. In: 2010 14th European conference on software maintenance and reengineering,2010;78–87 : https://doi.org/10.1109/CSMR.2010.33. IEEE
17. Sheneamer A, Kalita J. A survey of software clone detection techniques. Int J Comput Appl. 2016;137(10):1–21.
18. Marcus, A., Maletic, J.I.: Identification of high-level concept clones in source code. In: Proceedings 16th annual international conference on automated software engineering (ASE 2001), pp. 107–114 (2001). https://doi.org/10.1109/ASE.2001.989796. IEEE
19. Sheneamer A, Roy S, Kalita J. A detection framework for semantic code clones and obfuscated code. Expert Syst Appl. 2018;97:405–20. https://doi.org/10.1016/j.eswa.2017.12.040.
20. Fang C, Liu Z, Shi Y, Huang J, Shi Q. Functional code clone detection with syntax and semantics fusion learning. In: Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis, pp. 2020;516–527 https://doi.org/10.1145/3395363.3397362
21. Alrabaee S, Wang L, Debbabi M. Bingold: towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (sfgs). Digit Investig. 2016;18:11–22. https://doi.org/10.1016/j.diin.2016.04.002.
22. Zhao Y, Mo R, Zhang Y, Zhang S, Xiong P. Exploring and understanding cross-service code clones in microservice projects. In: 2022 IEEE/ACM 30th international conference on program comprehension (ICPC), 2022:449–459 ; https://doi.org/10.1145/3524610.3527925. IEEE
23. Kamiya T, Kusumoto S, Inoue K. A token-based code clone detection tool-ccfinder and its empirical evaluation. Techinal report, Osaka University, Department of Information and Computer Scineces, Inoue Laboratory (2000)
24. Papadimitriou CH, Raghavan P, Tamaki H, Vempala S. Latent semantic indexing: a probabilistic analysis. J Comput Syst Sci. 2000;61(2):217–35. https://doi.org/10.1006/jcss.2000.1711.
25. Hou C, Nie F, Li X, Yi D, Wu Y. Joint embedding learning and sparse regression: a framework for unsupervised feature selection. IEEE Trans Cybern. 2013;44(6):793–804. https://doi.org/10.1109/TCYB.2013.2272642.
26. Baldi P, Chauvin Y. Neural networks for fingerprint recognition. Neural Comput. 1993;5(3):402–18. https://doi.org/10.1162/neco.1993.5.3.402.
27. Weiser M. Program slicing. IEEE Trans Softw Eng SE. 1984;10(4):352–7. https://doi.org/10.1109/TSE.1984.5010248.
28. Alomari HW, Collard ML, Maletic JI, Alhindawi N, Meqdadi O. srcslice: very efficient and scalable forward static slicing. J Softw: Evol Proc. 2014;26(11):931–61. https://doi.org/10.1002/smr.1651.
29. Rakić G. Extendable and adaptable framework for input language independent static analysis. PhD thesis, University of Novi Sad (Serbia) 2015
30. Koschke R, Falke R, Frenzel P. Clone detection using abstract syntax suffix trees. In: 2006 13th Working conference on reverse engineering, pp. 2006;253–262 https://doi.org/10.1109/WCRE.2006.18. IEEE
31. Cordy JR, Roy CK. The nicad clone detector. In: 2011 IEEE 19th international conference on program comprehension, pp. 219–220 (2011). https://doi.org/10.1109/ICPC.2011.26. IEEE
32. Koschke R, Baxter ID, Conradt M, Cordy JR. Software clone management towards industrial application (dagstuhl seminar 12071). In: Dagstuhl Reports, vol. 2 (2012). https://doi.org/10.4230/DagRep.2.2.21. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik
33. Schiewe M, Curtis J, Bushong V, Cerny T. Advancing static code analysis with language-agnostic component identification. IEEE Access. 2022;10:30743–61. https://doi.org/10.1109/ACCESS.2022.3160485.
34. JBoss: Javassist : Java bytecode engineering toolkit (2020). https://www.javassist.org Accessed 2021-06-18
35. Christiane F, Brown K. Wordnet and wordnets. In: Encyclopedia of Language and Linguistics. UK, Oxford: Elsevier; 2005. p. 665–70.
36. Bishop CM, Nasrabadi NM. Pattern recognition and machine learning **4**(4) (2006)
37. James G, Witten D, Hastie T, Tibshirani R. An introduction to statistical learning 2013;112
38. Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E. Scikit-learn: machine learning in Python. J Mach Learn Res. 2011;12:2825–30.
39. Manning CD, Introduction to information retrieval 2008.

40. Lemnaru C, Potolea R Imbalanced classification problems: systematic study, issues and best practices. In: Enterprise information systems: 13th international conference, ICEIS 2011, Beijing, China, June 8-11, 2011, Revised Selected Papers 13, pp. 35–50 (2012). https://doi.org/10.1007/978-3-642-29958-2_3. Springer
41. Abu-Mostafa YS, Magdon-Ismail M, Lin H-T. Learn Data, vol. 4. NY, USA: AMLBook New York; 2012.
42. Wohlin C, Runeson P, Höst M, Ohlsson M, Regnell B, Wesslén A. Experimentation in Software Engineering: An Introduction. Germany: The Kluwer International Series In Software Engineering. Springer; 2000.