OmniCCG: Agnostic Code Clone Genealogy Extractor

Denis Sousa
State University of Ceará
Fortaleza, Brazil
denis.sousa@aluno.uece.br

Matheus Paixao
State University of Ceará
Fortaleza, Brazil
matheus.paixao@uece.br

Thiago Lima
State University of Ceará
Fortaleza, Brazil
thiaguinho.lima@aluno.uece.br

Adriely Silva
State University of Ceará
Fortaleza, Brazil
adri.silva@aluno.uece.br

Italo Uchoa
State University of Ceará
Fortaleza, Brazil
italo.uchoa@aluno.uece.br

Chaiyong Ragkhitwetsagul
Faculty of ICT, Mahidol University
Nakhon Pathom, Thailand
chaiyong.rag@mahidol.ac.th

Abstract

When two or more code snippets are identical or sufficiently similar, they form code clones. Such duplication can harm system maintainability as the software evolves. Code clone genealogy (CCG) extraction involves analyzing successive versions of a software system to identify code clones, their modifications, additions, and removals. Visualizing clone genealogies helps developers manage their clones, improving code comprehensibility and maintainability. Despite their importance, to the best of our knowledge, no fully functional, easily executable clone genealogy extractor exists. Furthermore, all extractors proposed in the literature are specifically designed to work with a particular set of clone detectors, resulting in strong coupling. To address these shortcomings, this paper presents OmniCCG, a code clone genealogy extractor that is agnostic to clone detectors. Given a Git repository and user settings, OmniCCG extracts code clone genealogies from the repository, along with common genealogy metrics, such as *clone density*, *k-volatile*, and others. Moreover, one may use OmniCCG in two different ways. The first is via a modern and responsive user interface, in which one can easily track the genealogies in their repository alongside a dashboard of relevant metrics. The second is via a console application that supports local execution. OmniCCG is available as an web application [27] and console application [28].

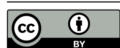
CCS Concepts

• **Software and its engineering → Software configuration management and version control systems**.

Keywords

Code Clone, Code Clone Detector, Code Clone Genealogy

ACM Reference Format:
Denis Sousa, Matheus Paixao, Thiago Lima, Adriely Silva, Italo Uchoa, and Chaiyong Ragkhitwetsagul. 2026. OmniCCG: Agnostic Code Clone Genealogy Extractor. In *23rd International Conference on Mining Software Repositories (MSR '26), April 13–14, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3793302.3793316



This work is licensed under a Creative Commons Attribution 4.0 International License.
MSR '26, Rio de Janeiro, Brazil
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2474-9/2026/04
https://doi.org/10.1145/3793302.3793316

1 Introduction

When two or more code snippets are identical or sufficiently similar, they form code clones [15, 23, 25, 33]. These snippets may represent methods, classes, or generic source code fragments. Such duplication can harm system maintainability as the software evolves. Code clones can force developers to repeat similar changes across several parts of the codebase. If they are not updated consistently, a bug may be only partially fixed, or new bugs may be introduced [14, 19, 22]. Developers need to be aware of code clones in their system and manage their evolution over time. To investigate the potential harms arising in the evolution of code clones in software systems, code clone detectors were developed [1, 8, 12, 18, 21, 26]. With these tools, developers can identify code clones in their codebases, enabling better management.

By leveraging clone detectors, one may conduct in-depth investigations into the evolution of clones over time. Code Clone Genealogy (CCG) extraction consists in analyzing subsequent versions of a software system to identify code clones, their modifications, additions and removals [16, 17, 20, 24, 30, 34]. When analyzing two consecutive versions of a code clone, one needs to observe and identify the occurrence of *Change patterns* and *Evolution patterns*. *Change patterns* are characterized by the number of modified code snippets in a code clone. *Evolution patterns* are characterized by the number of code snippets that were added or removed in a code clone. By capturing all code clones from a system and analyzing their respective *Change patterns* and *Evolution patterns*, a clone genealogy is built.

In recent studies on code clone genealogy, authors tend to develop their own genealogy extractor, rather than reusing existing ones [4, 11, 13, 31]. Currently, most studies related to genealogy extractors no longer have their replication package available [17, 20, 24, 29, 34]. For the existing replication packages, the code does not work, requiring modifications to the available code, and providing little to no documentation [4, 11, 13, 30]. Furthermore, all existing genealogy extractors are specifically designed to work with a particular set of clone detectors, resulting in strong coupling and creating a barrier to users with a preferred clone detector.

To address these shortcomings, this paper presents OmniCCG, a code clone genealogy extractor that is agnostic to clone detectors. Given a Git repository and user settings, OmniCCG extracts the entire clone genealogy from the repository. The extraction involves running a clone detector for each software version in the repository, then building a genealogy that references the change and

evolution patterns between versions. OmniCCG is agnostic because it is designed to be able to integrate with any clone detector. To achieve this, the user needs to implement an API that encapsulates the clone detector of their preference. OmniCCG also captures common genealogy metrics, such as *clone density*, *k-volatile*, and others [17, 24, 30].

The tool offers two ways to use it. The first is a modern and responsive interface that enables tracking clone genealogies in repositories and visualizing relevant metrics through a dashboard. The second way of using OmniCCG is through a console application that supports local execution and manipulation of the extracted genealogy and metrics. In its current implementation, OmniCCG provides built-in clone detection through Simian [1] and Nicad [8]. OmniCCG is available as an web application [27] and console application [28].

2 Terminology

This section presents the terminology related to code clone genealogy as discussed in the literature [4, 16, 17, 20, 24, 30, 34].

Change pattern: describes how the content of code snippets within clone group changes from one version of the system to the next. *Change patterns* can be classified into three types. The *Same* pattern occurs when the developer makes no modifications in code snippets within clone group. The *Consistent* pattern occurs when all code snippets within clone group receive equivalent modifications, resulting in the code snippets remaining clones between each other. The *Inconsistent* pattern occurs when only some code snippets within a clone group receive equivalent modifications, resulting in a new set of code snippets that are clones of each other, while the remaining code snippets are no longer clones of the new set.

Evolution pattern: describes how the number of code snippets within a clone group changes from one version of the system to the next. *Evolution patterns* can be classified into three types. The *Same* pattern occurs when the developer makes no additions or deletions in code snippets within clone group. The *Addition* pattern occurs when the number of code snippets within a clone group increases. The *Subtraction* pattern occurs when the number of code snippets within a clone group decreases.

Clone lineage: is the ordered set of all versions of one code clone throughout a system's history. In a clone lineage, a clone in version n of the system is connected with a clone in version $n-1$ by a *Change pattern* and a *Evolution pattern*.

Clone genealogy: is a set of clone lineages originating from the same code clone. The change pattern and evolution pattern may differ between lineages after they diverge from the common origin.

Figure 1 illustrates the terminology presented above, with an example of the genealogy of code clones formed from two lineages. Code snippets A and B represent a code clone in its first version in the system. These code snippets constitute the origin of the genealogy. In the second version of the system, snippets A and B remained the same and snippets C and D are added, representing a *Same* change pattern and an *Addition* evolution pattern. In the third version of the system, only snippets C and D are modified equivalently, representing an *Inconsistent* change pattern. Moreover, no snippets were added or removed, indicating a *Same* evolution pattern. The *Inconsistent* pattern marks the moment when the lineages split, with a first lineage composed of snippets A and B, and a second lineage composed of snippets C and D. In the fourth version of the system, in the first lineage, snippets A and B are modified equivalently, and snippet E is added, representing a *Consistent* change pattern and an *Addition* evolution pattern. In the second lineage, snippet D is removed without modification to the code in snippet C, representing a *Same* change pattern and a *Subtraction* evolution pattern. With this removal, snippet C ceases to be a clone with other snippets, marking the death of a clone and its lineage.

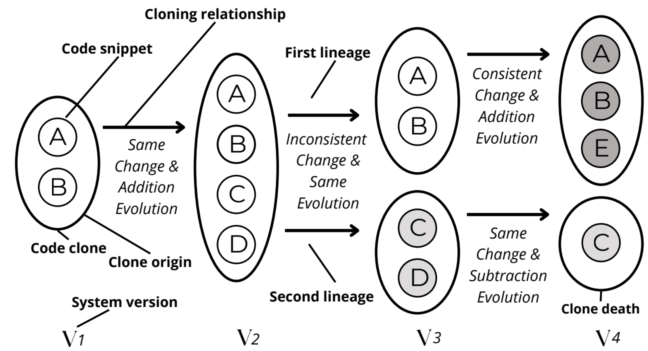


Figure 1: Example of a Code Clone Genealogy (CCG).

3 OmniCCG

This section provides an overview of OmniCCG, describing the user settings, its architecture, how to install and execute it.

3.1 User Settings

To execute OmniCCG, the user must initially provide a Git repository and settings to extract the genealogy. Table 1 summarizes the settings that can be specified in OmniCCG. These settings allow the user to customize how the commit history should be navigated, determining how the genealogy will be assembled. The user can choose the commit from which the genealogy should be built. They can select the first commit made in the repository to build the genealogy, provide the hash of a specific commit, or use the first commit made N days ago. In addition, the user can make other customizations, such as selecting only commits that were merged into the repository, or setting a fixed leap of N commits, selecting one commit at every N leap to build the genealogy.

3.2 OmniCCG Architecture

Figure 2 presents the architecture of OmniCCG. Initially, OmniCCG extracts a commit history using the `GitPython`[1] library. For each commit obtained, OmniCCG performs code clone detection on the system, storing in memory the code clones obtained in each version.

To detect code clones, OmniCCG currently provides native integration with the Nicad [8] and Simian [1] clone detectors. Both detectors are executed with fixed settings. For NiCad, we adopt the settings used by Van et al. [30], whereas for Simian we use the default settings. Both detectors are configured to work only with Java code. If the user does not want to use one of these detectors,

[1] https://gitpython.readthedocs.io

Table 1: OmniCCG Settings

Settings	Description
Commit starting point	
From the First Commit	The genealogy is built from the repository's first commit.
From a Specific Commit	The genealogy is built from the hash commit provided by the user.
Days Prior	The genealogy is built from the first commit made within the last *N* days chosen by the user.
Customizations for commit extraction	
Merge Commits	The genealogy is built using only merge commits.
Fixed Leaps	The genealogy is built with a fixed leap of *N* commits, selecting one commit at every *N* leap.

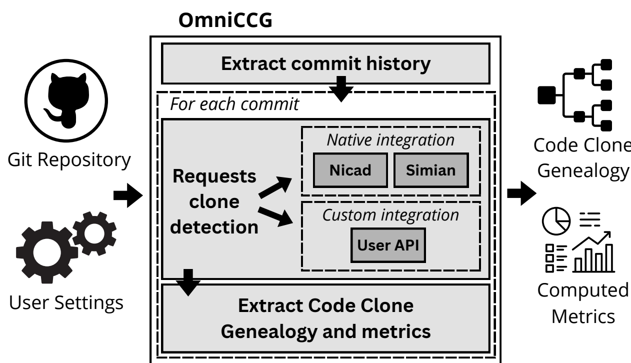


Figure 2: OmniCCG Architecture.

they can use the *Custom integration*. To do this, the user needs to implement an API that has the target Git repository locally cloned, providing and endpoint that accepts a commit hash in the *query string*. OmniCCG makes an HTTP request using the `requests`[2] library to the following route:

```
GET /<user-api>/clone-detection/?sha=<hash>
```

When the API receives an HTTP call to this endpoint, a git checkout must be performed for the submitted commit. Next, clone detection must be performed using the user's preferred detector.

After the clone detector finishes, the developer's API must return the code clones following the predefined XML schema presented in Figure 3. This XML structure is a simplified version of the XML produced by NiCad. We use NiCad's structure as a basis because it is the most widely used clone detector in recent code clone genealogy studies [4, 11, 30]. Different code clone detector output clones in a different structure, which may be CSV, XML, JSON, YAML, or others. Therefore, the user must implement a parser within the API so that the obtained clones are returned in the correct structure. If NiCad is used by the developer, parsing the results is unnecessary.

The clone genealogy and metrics were implemented based on the code provided by van Bladel et al. [30], which, in turn, was based on the works presented by Kim et al. [17] and Bakota et al. [5].

To build the code clone genealogy, it is necessary to track, chronologically, how the same code clones appear across different versions

[2]https://pypi.org/project/requests

Figure 3: XML schema for clone detection API.

```xml
<clones>
    <class>
        <source file="/folder/ExampleOne.java"
                startline="4" endline="25"></source>
        <source file="/folder/ExampleTwo.java"
                startline="70" endline="86"></source>
    </class>
    ...
</clones>
```

of the repository. To identify that a specific code clone is present in two distinct versions, we analyze the method name and its path. When analyzing two subsequent versions of the system, we capture the method name and path of each code snippet of a specific clone in both versions. Next, we take all the clones from the second version and compare the method name and path of each code snippet of the clones from the first version. If the path and method name match between versions, it indicates that code snippets remained clones. If there was no match, it represents the clone appeared in the second version.

For each version, we identified *Change patterns* and *Evolution patterns*. To determine the *Change Pattern*, we capture the code content of each code snippet and perform text normalization by removing comments and whitespace. Using Python's `hashlib`[3] library, we create a hash for each code snippet. Thus, we compare the hashes of the normalized code snippets in the two consecutive versions. If all the hashes have changed, it indicates a *Consistent pattern*. If some of the hashes have changed but not all, it indicates an *Inconsistent* pattern. To determine the *Evolution pattern*, we count the number of code snippets in two consecutive versions. If the number of code snippets remains the same, it indicates a *Same* pattern. If it increases, it indicates an *Addition* pattern. Finally, if the number of code snippets decreases, it indicates a *Subtraction* pattern.

After extracting the genealogy, OmniCCG captures common genealogy metrics [17, 24, 30]. One example of a captured metric is *clone density*, also referred to as *clone percentage* [6]. This metric is defined as:

$$\text{CloneDensity} = \frac{\text{LOC}_c}{\text{LOC}_{tot}} \times 100 \quad (1)$$

LOC_c denotes the number of cloned lines of code and LOC_{tot} is the total number of lines of code in the system. This metric is useful for tracking the evolution of clones, enabling the visualization of how duplication evolves throughout development, and facilitating the investigation of anomalies on code clones.

OmniCCG captures others metrics, such as *k-volatile* and relative ratio of the *Evolution patterns* and *Change patterns*. For a full reference of these metrics, we refer to their original publications [17, 30].

4 Installation and usage

OmniCCG can be used through its online web application [27]. The first screen is a home page where the user must provide a the Git repository. Next, the user is directed to a settings page where they

[3]https://docs.python.org/3/library/hashlib.html

can specify from which commit the genealogy will be extracted and choose the clone detector, selecting either NiCad or Simian. If the user wants to use their preferred clone detector, they must provide the endpoint of its API on the platform. They can also choose to extract only merge commits and whether the history will be built with leaps. Once configured on the platform, the genealogy will be extracted and displayed. On this page, the user can analyze the lineages, *Change patterns*, and *Evolution patterns* of code clones. It is also possible to navigate to a metrics dashboard page, allowing the user to have a holistic view of how the clones have evolved across the repository's history.

OmniCCG can also be used through a console application. For this, one should download the tool source code from our replication package [28], and install it using pip (`pip install -e .`). To extract a genealogy, the user can run the command:

```
omniccg --git-repo <git-project-url> --from-first-commit
--merge-commit --clone-detector nicad
```

This command extracts the genealogy of `<git-project-url>` from the first commit, using only merge commits and detecting clones with Nicad. The replication package [28] provides more details on configuring and executing OmniCCG.

5 Preliminary Evaluation

For this study, we performed a preliminary evaluation of OmniCCG to showcase its main functionalities. We selected Java repositories with more than 5,000 stars, more than 10,000 lines of code (LOC), more than 30 contributors, and more than 500 pull requests. These constraints help ensure that the selected repositories are popular and substantially large. Using SEART GitHub search [9], we apply these constraints and identify four Java projects: *ai-alibaba* [2], *pkl* [3], *BookLore* [10], and *PeerBanHelper* [7].

For each project, OmniCCG was executed with the same settings, extracting the genealogy from the first commit at 200 commit leaps, using the NiCad detector. The shortest execution time was for *BookLore* (32 seconds), which has 6 extracted commits, whereas *ai-alibaba* had the longest execution time (152 seconds), with 15 extracted commits.

In our manual analysis, we examined whether all occurrences of the *Consistent*, *Inconsistent*, *Addition*, and *Subtraction* patterns indeed occurred in the clone genealogy of the projects. All pattern identifications made by OmniCCG were correct. Table 2 presents the total number of *Change patterns* and *Evolution patterns* identified by OmniCCG for each project.

Table 2: Number of Change Patterns and Evolution Patterns identified by OmniCCG during evaluation.

			Change Patterns		Evolution Patterns	
Project	**Commits**	**Total Patterns**	**Consistent**	**Inconsistent**	**Addition**	**Subtraction**
booklore	6	90	1	1	1	0
pkl	4	93	39	2	14	13
ai-alibaba	15	3379	18	9	8	4
PeerBanHelper	27	2390	53	14	7	6
Total			**111**	**26**	**30**	**23**

6 Related Work

This section discusses the most recent studies that provide genealogy extractors in their replication packages.

Hu et al. [13] use code clone genealogy to evaluate clone "harmfulness" by building genealogies from Git blob histories and detecting clones with SAGA [18]. The replication package currently contains only a README file.

van Bladel et al. [30] use clone genealogy to compare clone evolution in production code and test code and to inform refactoring and maintenance estimates. They build the clone genealogy using NiCad [8] and iClones [12]. The pipeline's code is available in the replication package but it is not functional.

Ehsan et al. [11] train classifiers and regressors across multiple systems to rank clones by estimated failure risk over time. Using NiCad [8] and iClones [12], they rebuild genealogy to map changes to failures and prioritize riskier clones. The replication package is available, but the genealogy extractor is not functional.

Assi et al. [4] use code clone genealogy to analyze clone evolution and code reuse across Deep Learning frameworks. They apply NiCad [8] for intra-framework clone detection and SourcererCC [26] for inter-framework analysis. Although a replication package is available, it provides only the results, not the genealogy extractor source code.

In short, current clone genealogy studies still lack an easily runnable and reproducible genealogy extractor. Most replication packages lack a functional, well-documented, and executable genealogy extractor. OmniCCG simplifies clone genealogy extraction and visualization through a modern web interface and a fully containerized Docker-based replication package that enhances reproducibility.

7 Conclusion

This paper presented OmniCCG, a code clone genealogy extractor that is agnostic to clone detectors. Given a Git repository and user settings, OmniCCG extracts the entire clone genealogy from the repository. The tool is agnostic because it can be integrated with any clone detector.

Limitations: Currently, when using the built-in clone detectors, the user is limited to using the fixed settings of Nicad and Simian. Both detectors are configured to find Java clones. Furthermore, the tool does not offer the possibility of extracting a genealogy using multiple clone detectors at the same time.

Future work: In the future, beyond addressing known limitations, we plan to extend OmniCCG's scope. New metrics will be captured to help recommend which code clones should be refactored [32]. Information about developers' actions (additions, removals, and modifications) during code cloning will also be collected, as this topic has been little explored in the literature [33]. We also aim to integrate the tool with continuous integration pipelines. In addition, we plan to support the simultaneous analysis of clone genealogies across multiple software repositories

ACKNOWLEDGMENTS

This work received partial funding from CNPq-Brazil, Universal grant 404406/2023-8

References

[1] Quandary Peak Research, Inc. 2024. *Simian Similarity Analyzer*. Quandary Peak Research, Inc. https://simian.quandarypeak.com/docs/

[2] Alibaba. 2023. spring-ai-alibaba. https://github.com/alibaba/spring-ai-alibaba. Accessed: 2025-10-24.

[3] Apple. 2023. pkl. https://github.com/apple/pkl. Accessed: 2025-10-24.

[4] Maram Assi, Safwat Hassan, and Ying Zou. 2024. Unraveling code clone dynamics in deep learning frameworks. *arXiv preprint arXiv:2404.17046* (2024).

[5] Tibor Bakota, Rudolf Ferenc, and Tibor Gyimothy. 2007. Clone smells in software evolution. In *2007 IEEE International Conference on Software Maintenance*. IEEE, 24–33.

[6] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 368–377.

[7] PBH-BTN Community. 2023. PeerBanHelper. https://github.com/PBH-BTN/PeerBanHelper. Accessed: 2025-10-24.

[8] James R Cordy and Chanchal K Roy. 2011. The NiCad clone detector. In *2011 IEEE 19th international conference on program comprehension*. IEEE, 219–220.

[9] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling Projects in GitHub for MSR Studies. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*. IEEE, 560–564.

[10] BookLore App Developers. 2023. BookLore. https://github.com/booklore-app/BookLore. Accessed: 2025-10-24.

[11] Osama Ehsan, Foutse Khomh, Ying Zou, and Dong Qiu. 2023. Ranking code clones to support maintenance activities. *Empirical Software Engineering* 28, 3 (2023), 70.

[12] Nils Göde and Rainer Koschke. 2009. Incremental clone detection. In *2009 13th European conference on software maintenance and reengineering*. IEEE, 219–228.

[13] Bin Hu, Yijian Wu, Xin Peng, Jun Sun, Nanjie Zhan, and Jun Wu. 2021. Assessing code clone harmfulness: Indicators, factors, and counter measures. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 225–236.

[14] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. 2007. Context-based detection of clone-related bugs. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 55–64.

[15] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. 2009. Do code clones matter?. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 485–495.

[16] Miryung Kim and David Notkin. 2005. Using a clone genealogy extractor for understanding and supporting evolution of code clones. *ACM SIGSOFT Software Engineering Notes* 30, 4 (2005), 1–5.

[17] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. 2005. An empirical study of code clone genealogies. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. 187–196.

[18] Guanhua Li, Yijian Wu, Chanchal K Roy, Jun Sun, Xin Peng, Nanjie Zhan, Bin Hu, and Jingyi Ma. 2020. SAGA: efficient and large-scale detection of near-miss clones with GPU acceleration. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 272–283.

[19] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner: A tool for finding copy-paste and related bugs in operating system code.. In *OSdi*, Vol. 4. 289–302.

[20] Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. 2014. An insight into the dispersion of changes in cloned and non-cloned code: A genealogy based empirical study. *Science of Computer Programming* 95 (2014), 445–468.

[21] Chaiyong Ragkhitwetsagul and Jens Krinke. 2019. Siamese: scalable and incremental code clone search via multiple code representations. *Empirical Software Engineering* 24, 4 (2019), 2236–2284.

[22] Baishakhi Ray, Miryung Kim, Suzette Person, and Neha Rungta. 2013. Detecting and characterizing semantic inconsistencies in ported code. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 367–377.

[23] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. 2009. Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. 117–128.

[24] Ripon K Saha, Muhammad Asaduzzaman, Minhaz F Zibran, Chanchal K Roy, and Kevin A Schneider. 2010. Evaluating code clone genealogies at release level: An empirical study. In *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*. IEEE, 87–96.

[25] Neha Saini, Sukhdip Singh, et al. 2018. Code clones: Detection and management. *Procedia computer science* 132 (2018), 718–727.

[26] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th international conference on software engineering*. 1157–1168.

[27] Denis Sousa, Matheus Paixao, Thiago Lima, Adriely Silva, Italo Uchoa, and Chaiyong Ragkhitwetsagul. 2025. *Online platform for the paper: 'OmniCCG: Agnostic Code Clone Genealogy Extractor"*. https://omnitool.store

[28] Denis Sousa, Matheus Paixao, Thiago Lima, Adriely Silva, Italo Uchoa, and Chaiyong Ragkhitwetsagul. 2026. *Replication Package for the paper: 'OmniCCG: Agnostic Code Clone Genealogy Extractor"*. https://zenodo.org/records/18344305

[29] Rajkumar Tekchandani, Rajesh Bhatia, and Maninder Singh. 2017. Code clone genealogy detection on e-health system using Hadoop. *Computers & Electrical Engineering* 61 (2017), 15–30.

[30] Brent van Bladel and Serge Demeyer. 2023. A comparative study of code clone genealogies in test code and production code. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 913–920.

[31] Yijian Wu, Yuan Chen, Xin Peng, Bin Hu, Xiaochen Wang, Baiqiang Fu, and Wenyun Zhao. 2025. CloneRipples: predicting change propagation between code clone instances by graph-based deep learning. *Empirical Software Engineering* 30, 1 (2025), 14.

[32] Ruru Yue, Zhe Gao, Na Meng, Yingfei Xiong, Xiaoyin Wang, and J David Morgenthaler. 2018. Automatic clone recommendation for refactoring based on the present and the past. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 115–126.

[33] Morteza Zakeri-Nasrabadi, Saeed Parsa, Mohammad Ramezani, Chanchal Roy, and Masoud Ekhtiarzadeh. 2023. A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges. *Journal of Systems and Software* 204 (10 2023), 111796. doi:10.1016/j.jss.2023.111796

[34] Minhaz F Zibran, Ripon K Saha, Chanchal K Roy, and Kevin A Schneider. 2013. Evaluating the conventional wisdom in clone removal: A genealogy-based empirical study. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. 1123–1130.