Exploring Code Clone Behavior in Modern Code Review

Italo Uchoa, Denis Sousa, Matheus Paixao, Pedro Maia
State University of Ceará
Fortaleza, Brazil
{italo.uchoa,denis.sousa,pedro.ulisses}@aluno.uece.br,matheus.paixao@uece.br

Anderson Uchôa
Federal University of Ceará
Itapajé, Brazil
andersonuchoa@ufc.br

Chaiyong Ragkhitwetsagul
Mahidol University
Nakhon Pathom, Thailand
chaiyong.rag@mahidol.edu

ABSTRACT

The Modern Code Review (MCR) process is iterative and asynchronous, enabling the early identification of issues during development. One of the main challenges in this context is the presence of code clones, fragments of code copied with small modifications that hinder maintainability. In this study, we analyzed 80k revisions from the CROP dataset using the Siamese detector, identifying 27,656 relevant clones across six systems. A manual validation indicated a predominance of Type-I (46.7%) and Type-III (45.3%) clones. We also identified 224 reviews in which clones appeared in a single revision (*Single*), 1,258 reviews in which clones appeared across multiple revisions (*Recurring*), and 236 reviews at the intersection of both categories. To deepen the analysis, we introduced two metrics, *Duration* and *Distance*, to assess how clones are introduced or removed during the review. This paper presents an expanded abstract of the study "An Exploratory Study on the lifecycle of Code Clones During Code Review", published at SBES 2025.

KEYWORDS

Modern Code Review, Code Clones, Empirical Study

1 Introduction

Modern Code Review (MCR) has enabled developers to build higher-quality systems by providing benefits related to maintenance and knowledge transfer [7]. Unlike traditional code inspections, which are a manual practice of reviewing code, MCR is fully tool-based. Platforms such as Gerrit and GitHub enable reviews to be performed asynchronously and dynamically. Furthermore, it occurs in an iterative manner, with each iteration being referred to as a revision. In this way, development teams can prevent future issues such as the introduction of bugs [10].

Code clones are fragments of code that are copied and reused in other (or the same) codebase, often with minor modifications [1, 5]. Code cloning increases productivity in implementing new features and is often introduced intentionally into the codebase [9]. However, if an error is identified in the code clone segment, it means that all clones must be corrected, which hinders the maintainability of the system [4]. Therefore, it is essential that developers are aware of the copies that exist in their codebase.

Although MCR is designed to help developers manage code cloning, no studies have examined the link between code cloning and code review. As a motivating example, we present a detailed analysis of the code review numbered 22961 from the *eclipse.platform.ui* project on the Gerrit platform. In this review, the developer added a method called *getStyleOverride* within the *StackRenderer* class. However, this method was copied from the *WBWRenderer* class, and one of the reviewers identified the duplicated code during the first revision and suggested a refactoring:

> *'I like this but perhaps we should be moving the getStyleOverride' logic into AbstractPartRenderer (since it doesn't use SWT) so that we don't keep making copies of it."*

As a result, the cloned fragment was refactored into the superclass, eliminating redundancy and improving maintainability. This case exemplifies successful clone mitigation. However, in another review (101346), a similar duplication was introduced without discussion, illustrating that while some clones are addressed during MCR, others remain unnoticed.

This expanded abstract presents the work previously published as [8]. Using the CROP dataset [2] and the Siamese [3] clone detector, we identified 27,656 relevant clones across six software systems, of which 96.7% were validated as true positives. These clones were categorized into Type-I (46.7%), Type-II (7.9%), and Type-III (45.3%). Moreover, code reviews were classified as *Single* (224 reviews with clones in a single revision) or *Recurring* (1,258 reviews with clones across multiple revisions), with 236 reviews overlapping both categories.

To analyze clone persistence, we introduced two metrics: *Duration*, measuring the proportion of revisions a clone remains, and *Distance*, indicating its first appearance. Results show that clones are often introduced early and frequently persist through the review process.

The contributions are: (i) the first empirical study on code cloning during code review, (ii) a dataset of manually validated clones linked to reviews, and (iii) two novel metrics (*Duration* and *Distance*) to assess clone persistence and introduction timing.

2 Methodology

The methodology of this study was guided by three research questions: (RQ1) Does code cloning occur during code review? (RQ2) What types of code clones emerge during code review? (RQ3) What is the lifecycle of code clones during code review? To answer the proposed RQs, we carried out a set of processes divided into four phases, as illustrated in Figure 1.

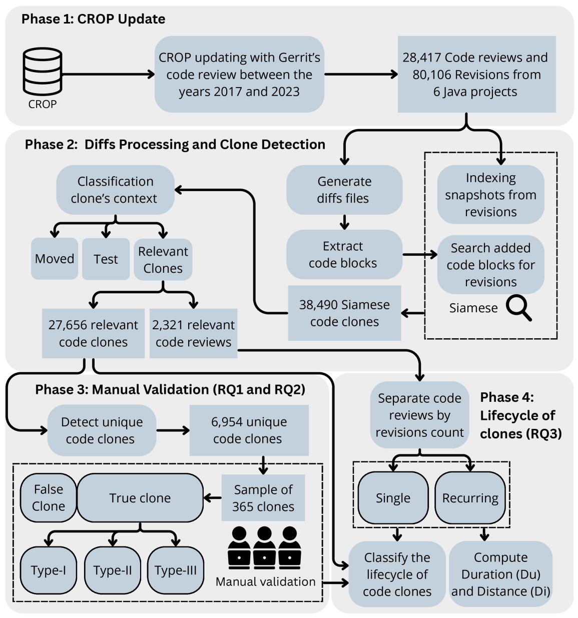
Phase 1: CROP Update
CROP
CROP updating with Gerrit's code review between the years 2017 and 2023
28,417 Code reviews and 80,106 Revisions from 6 Java projects
Phase 2: Diffs Processing and Clone Detection
Classification clone's context
Generate diffs files
Indexing snapshots from revisions
Moved
Test
Relevant Clones
Extract code blocks
Search added code blocks for revisions
38,490 Siamese code clones
Siamese
27,656 relevant code clones
2,321 relevant code reviews
Phase 3: Manual Validation (RQ1 and RQ2)
Detect unique code clones
6,954 unique code clones
False Clone
True clone
Sample of 365 clones
Type-I
Type-II
Type-III
Manual validation
Separate code reviews by revisions count
Phase 4: Lifecycle of clones (RQ3)
Single
Recurring
Classify the lifecycle of code clones
Compute Duration (Du) and Distance (Di)

Figure 1: Empirical methodology employed in this study.

In **Phase 1 (CROP Update)**, the CROP dataset was updated to include reviews conducted between 2017 and 2023 in six Java projects. For this, the original mining code was adapted, migrating from Python 2.7 to Python 3.10 and adjusted to the new versions of the Gerrit API. As a result of this process, we extracted a total of 28,417 code reviews and 80,106 revisions across six Java projects, forming the basis for the subsequent phases of our study.

In **Phase 2 (Diff Processing and Clone Detection)**, each revision, which refers to the code versions recorded during the code review, was analyzed by generating diffs between before and after snapshots. From this, only added code blocks (with at least six lines) were selected for duplication search using the Siamese detector. Initially, Siamese identified 38,490 code clones. However, we excluded cases that were not true duplicates, such as clones in test files and moved snippets. In our preliminary analysis, we found that some clones reported by Siamese were actually code snippets moved within the codebase, not true duplicates. Because diff files represent moved code as separate additions and removals, these cases appeared as new code. After this filtering, we obtained 27,656 relevant clones distributed across 2,321 code reviews.

In **Phase 3 (Manual Validation)**, a stratified sampling process was applied to the relevant clones, eliminating repetitions and reducing them to a set of 6,954 unique clones. From this set, a statistically representative sample of 365 clones was selected for manual analysis. Three researchers validated the presence and type of each clone, classifying them as Type-I (identical copies with only formatting differences), Type-II (copies with minor changes such as variable names or data types), and Type-III (copies with structural modifications but preserving the same functionality), with a pilot study conducted to ensure consistency among evaluators. This phase confirmed the presence of clones, which motivated the exploration of RQ3.

In **Phase 4 (Clone Lifecycle Analysis)**, 1,718 code reviews containing more than one revision were investigated. These were categorized as Single, when clones appear only in one revision, and Recurring, when they appear in multiple revisions. In addition, two metrics were proposed: *Duration* and *Distance*. The *Duration* metric measures the proportion of revisions in which a clone was present, i.e., it is calculated by dividing the number of revisions where the clone appears by the total number of revisions in the review. The *Distance* metric indicates the moment of the first appearance of the clone in the process, defined as the relative position of the first revision in which the clone emerges in relation to the total revisions. Both metrics range from 0 to 1. Furthermore, clones were classified according to Table 1.

3 Results and Discussions

RQ1 – Does code cloning occur during code review? The results showed that the presence of clones is common during the code review process. From the 80,106 revisions analyzed, Siamese identified 27,656 relevant clones distributed across 2,321 code reviews. With this and the manual analysis, it was shown that 96.7% of the sample were indeed clones, and when extrapolating this result to all clones found by the Siamese tool, approximately 26,743 can be considered true clones. This extrapolation was also supported by the fact that the classification of clones in both the total population and the sample were similar. These results confirm that code cloning is not only introduced in the initial development but also emerges recurrently during the review process.

RQ2 – What types of code clones emerge during code review? In the manual validation, 96.7% of the sampled clones were confirmed as true clones and classified into three main types: Type-I (46.74%), which are identical clones with only formatting differences; Type-II (7.95%), with small changes in names or data types; and Type-III (45.3%), which involve structural modifications but maintain the same functionality. This predominance of Type-I and Type-III indicates that both literal duplications and structural variations frequently appear in reviews. The low proportion of Type-II clones (7.9%) indicates that small edits to copied code are uncommon within the same project. As shown by Svajlenko et al. [6], who analyzed millions of clones in 25,000 Java projects, Type-I and Type-III clones are far more frequent, with only 0.06% of benchmark clones classified as Type-II. This suggests developers typically either reuse code exactly or make substantial changes, rarely just renaming elements.

RQ3 – What is the lifecycle of code clones during code review? According to Table 2, the lifecycle analysis of clones in 1,718 reviews with multiple revisions revealed different behaviors. In reviews classified as Single, clones generally appeared and disappeared before the code change was merged

Table 1: Categories used to assess a clone's lifecycle

Type	Category	Description
Single	Early Stage	Code reviews where clones appeared only at the beginning.
	Mid Stage	Code reviews where clones appeared only in the middle.
	Late Stage	Code reviews where clones appeared only at the end.
Recurring	Emerging Cycle	Code reviews where clones appeared from the beginning to the middle.
	Central Cycle	Code reviews where clones appeared repeatedly during the middle.
	Closing Cycle	Code reviews where clones appeared from the middle to the end.
	Full Cycle	Code reviews where clones appeared throughout all revisions.
	Unstable Cycle	Code reviews where clones appeared, disappeared, and reappeared.

Table 2: Distribution of Clone Life Cycle Categories in Code Reviews

Category	Count
Single - Early Stage	287
Single - Mid Stage	140
Single - Late Stage	131
Recurring - Emerging Cycle	224
Recurring - Central Cycle	134
Recurring - Closing Cycle	426
Recurring - Full Cycle	967
Recurring - Unstable Cycle	31

into the project itself (final stage of the code review). In Recurring, however, more complex patterns were observed, such as clones spanning several stages (Emerging Cycle, Closing Cycle, Full Cycle), and most of them lasted until the end of the review, meaning the clone was merged into the project. Moreover, the results of the proposed metrics showed that clones tend to emerge in the early revisions and often persist until the code is merged into the main codebase, as *Distance* had an average of 0.17 and *Duration* had an average of 0.71.

The qualitative analysis showed that Type-I and Type-III clones are predominant during code reviews. Despite the use of modern practices and support tools, code duplication still occurs frequently. Detecting these clones early is essential, as many persist even after the code is merged, increasing the complexity of maintenance and debugging.

We identified two main factors contributing to this scenario. The first is the lack of perception by developers and reviewers, who often do not recognize the existence of duplicated fragments in the codebase, especially in the absence of indicators or tool support. This can pose risks when issues are fixed in only one instance of the clone, leaving other versions susceptible to errors. The second factor is related to the subtle differences in Type-III clones, which make their manual identification more difficult. These structural or syntactic variations reduce the likelihood of detection during reviews, increasing the chances that such clones will be incorporated into the project without proper attention.

4 Conclusion

In this research, we conducted an empirical study on the lifecycle of code clones during the modern code review process. We used the Siamese clone detector to analyze six open-source projects from the CROP dataset, detecting a total of 38,490 code clones. After filtering relevant and duplicate clones, we identified approximately 6,954 unique clones, of which 96.7% were true positives, mostly Type-I (46.74%) and Type-III (45.3%). We analyzed the lifecycle of these clones across multiple revisions using the *Duration* and *Distance* metrics and observed that most clones tend to survive the review process and be merged into the project. Moreover, clones usually appear early in the reviews, with an average *Distance* of 0.17.

REFERENCES

[1] Qurat Ul Ain, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. 2019. A systematic review on code clone detection. *IEEE access* 7 (2019), 86121–86144.

[2] Matheus Paixao, Jens Krinke, Donggyun Han, and Mark Harman. 2018. CROP: Linking Code Reviews to Source Code Changes. In *Proceedings of the IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)* (Gothenburg, Sweden). IEEE, 46–49.

[3] 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.

[4] Chaiyong Ragkhitwetsagul, Jens Krinke, Matheus Paixao, Giuseppe Bianco, and Rocco Oliveto. 2019. Toxic code snippets on stack overflow. *IEEE Transactions on Software Engineering* 47, 3 (2019), 560–581.

[5] Chanchal K Roy, Minhaz F Zibran, and Rainer Koschke. 2014. The vision of software clone management: Past, present, and future (keynote paper). In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 18–33.

[6] Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 476–480.

[7] Anderson Uchôa, Caio Barbosa, Daniel Coutinho, Willian Oizumi, Wesley KG Assunçao, Silvia Regina Vergilio, Juliana Alves Pereira, Anderson Oliveira, and Alessandro Garcia. 2021. Predicting design impactful changes in modern code review: A large-scale empirical study. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 471–482.

[8] Italo Uchoa, Denis Sousa, Matheus Paixão, Pedro Ulisses, Anderson Uchoa, and Chaiyong Ragkhitwetsagul. 2025. An Exploratory Study on the Lifecycle of Code Clones During Code Review. In *Brazilian Symposium on Software Engineering (SBES)*. Recife, Brazil.

[9] Wei Wang and Michael Godfrey. 2014. Investigating intentional clone refactoring. *Electronic Communications of the EASST* 63 (2014).

[10] Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. 2016. Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment? *Information and software technology* 74 (2016).