Artifacts Available CBSoft 2025

An Exploratory Study on the Lifecycle of Code Clones During 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 early identification of several issues during software development. Overall, code review consists of inspecting code before merging it into the codebase. Code clones are code fragments that are copied and reused across different (or the same) codebases, often with minor changes. Developers must be aware of code clones in their projects, as issues in a cloned fragment may cause adjustments in all related clones, which can significantly impact the project's maintainability. Nevertheless, there is still a gap in research addressing the presence and behavior of code clones during code review. By leveraging the CROP dataset (with over 28k code reviews and 80k revisions) and the Siamese clone detector, we identified 27,656 relevant code clones that underwent code review in 6 different software systems. A manual validation of a representative sample indicated a predominance of Type-I (46.74%) and Type-III (45.3%) clones. Based on the clones' lifecycle within the review, we categorized the reviews into Single and Recurring, according to how the clones are introduced and/or removed during the review process. We identified 224 reviews for which clones appear in a single review (Single) and 1,258 reviews for which clones appear in multiple revisions (Recurring). Additionally, 236 code reviews lie at the intersection of Recurring and Single code reviews. To deepen the analysis, we introduced two metrics, *Duration* and *Distance*, to assess how clones are introduced or removed during the review. We observed that, on average, clones are often introduced at the beginning of the code review, commonly surviving the review process and being merged into the codebase.

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 [8, 26]. This practice is common among large companies such as Microsoft [2], Google [21], and Facebook [4]. Through MCR, developers are able to analyze code modifications before they are integrated into the codebase in a faster and more intuitive manner. Unlike traditional code inspections, which is 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 [34], code smells [17], security vulnerabilities [25], architectural changes [14], among others.

Code clones are fragments of code that are copied and reused on other (or the same) codebase, often with minor modifications [1, 20]. Code cloning increases productivity in implementing new features and is often introduced intentionally into the codebase [31]. 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 [5, 10, 19]. Therefore, it is essential that developers are aware of the copies that exist in their codebase. Although some clones can be spotted manually, it is more common that code clones are detected through the use of static analysis tools such as code clone detectors [35].

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 moved to the superclass to eliminate redundancy and improve code maintainability. This example represents a successful code clone mitigation, as the code clone was quickly identified, and a corrective action was proposed through the use of an abstract class.

In review number 101346 of the same project, we observed a similar scenario. Most of the code from the *createButtonsForButtonBar* method in the *SaveAsDialog* class was copied to the *CustomizePerspectiveDialog* class with a few added changes. However, in this case, there was no discussion regarding the code duplication. The copied snippet was introduced in the

```
protected void createButtonsForButtonBar(
  Composite parent) {
  okButton = createButton(parent,
  IDialogConstants.OK_ID,
  IDialogConstants.OK_LABEL,
  true);
  createButton(parent,
  IDialogConstants.CANCEL_ID,
  IDialogConstants.CANCEL_LABEL,
  false);
}
```

```
protected void createButtonsForButtonBar(
  Composite parent) {
  Button okButton = createButton(parent,
  IDialogConstants.OK_ID,
  WorkbenchMessages.
  CustomizePerspectiveDialog_okButtonLabel,
  true);
  okButton.setFocus();
  createButton(parent,
  IDialogConstants.CANCEL_ID,
  IDialogConstants.CANCEL_LABEL,
  false);
}
```

Figure 1: Original (left) and cloned (right) versions of method *createButtonsForButtonBar*

first revision, lasting until the fifth revision, and was finally merged into the codebase. Figure 1 depicts the original *createButtonsForButtonBar* method and the cloned version. These examples demonstrate that while some clones are detected and addressed during MCR, others remain undetected.

This paper presents an empirical study on the lifecycle of code clones in MCR to understand their presence and behavior during code review. Our goal is to provide insights for developers, tool builders, and researchers on code cloning in this context.

By leveraging the CROP dataset [13] and the Siamese [18] clone detector, we identified 27,656 relevant code clones in 6 different software systems. Next, we took a statistically significant sample of these relevant clones and performed a manual validation. As a result, 96.7% of the identified clones were validated as true clones. In addition, we also manually classified the clones into Type-I (i.e., exact clones with only formatting differences), Type-II (i.e., Clones differing only in names and data types), and Type-III (i.e., clones with added/removed/changed statements). We identified 46.74% and 45.3% of Type-I and Type-III clones, respectively.

Following that, the code reviews with more than one revision and the presence of code clones were categorized as *Single* or *Recurring*. In our data, we identified 224 reviews for which clones appear in a single revision (*Single*) and 1,258 reviews for which clones appear in multiple revisions (*Recurring*). Additionally, 236 code reviews lie at the intersection of *Recurring* and *Single* reviews. From the code reviews classified in these categories, we carried out an analysis of the clones' lifecycle, analyzing the moment in which they appear and disappear in the code review.

Finally, two metrics are proposed to assess the persistence and introduction of code clones in code reviews. The *Duration* metric measures the number of revisions the clone remained during the code review. Additionally, *Distance* measures at which revision the first occurrence of the clone occurs in the code review. Overall, the results from *Duration* and *Distance* suggest that clones usually persist throughout much of the code review and are mostly introduced early in the process.

The main contributions of this paper are listed as follows:

- To the best of our knowledge, this is the first empirical study of code cloning during code review.
- A dataset [28] of manually validated code clones linked to code reviews.
- Two novel metrics (*Duration* and *Distance*) to measure clone persistence and introduction timing.

2 Background

2.1 Modern Code Review

Modern Code Review (MCR) was introduced by Cohen [3] and popularized by Bacchelli and Bird [2]. It is a software development practice where team members review code changes submitted by a developer to identify defects, improve quality, and promote knowledge sharing. Reviewers typically look for bugs, performance issues, design flaws, and style violations [16, 17, 27].

This practice is widely adopted by major companies such as Microsoft [2], Facebook [4], and Google [21]. MCR is iterative by nature, as after the initial review of the code changes, reviewers provide feedback through comments. The code owner then adjusts the code based on the comments and resubmits a revised version of the code for inspection. Thus, code review may go through multiple revisions, which correspond to intermediate versions of the code that are refined over time [2]. These revisions serve as temporary versions that are progressively adjusted until the review is approved and the code changes are merged into the codebase.

MCR is flexible due to the lack of strict guidelines and is supported by tools like Gerrit and GitHub, which enable asynchronous reviews with inline comments. Integration with version control and CI/CD pipelines further streamlines the process within agile workflows [36].

2.2 Code Clones

Code Clones are pairs of code fragments that are considered similar based on a defined notion of similarity. This similarity can be assessed at different levels of programming abstraction (e.g., statement, block, method, class, package, or component) and within one or across multiple projects. In the study by Zakeri-Nasrabadi et al. [35], code clones are classified into four types: (1) Type-I, in which the code fragments are identical, except for variations in white spaces and comments; (2) Type-II, in which the code fragments maintain the same structure, but may differ in in identifiers' names, data types, white spaces, and comments; (3) Type-III, where the code fragments may include changes in identifiers, variable names, data types, and comments, and parts of the code may be deleted, updated, or newly added; and (4) Type-IV, in which the code fragments differ in the text but implement the same functionality.

Clones may be intentionally added to meet specific requirements or developer preferences. Several studies propose methods for identifying different types of code clones by analyzing code repositories [1, 22, 35], often combining multiple detection approaches. The literature [1, 35] highlights common approaches for clone detection. The textual approach uses text processing to find Type-I and Type-II clones. The token-based (lexical) approach [7, 18] efficiently detects Type-III clones, while also identifying Type-I and Type-II clones by breaking code into symbol sequences for comparison. Tree-based methods use abstract syntax trees, while graph-based methods rely on structures like program dependence graphs [32]. The semantic approach, usually based on machine-learning-based techniques, is more appropriate for detecting Type-IV clones, which are undecidable in general. There are other approaches such as metric-based, image-based, test-based, or hybrid.

In this study, we use the token-based Siamese tool [18] to detect Java code clones in code review. Siamese was chosen for its scalability and ability to identify Type-I to Type-III clones, even with minor code changes, in large codebases.

3 Experimental Setup

To conduct the empirical study proposed in this study, we selected a clone detector and a code review dataset. Both of which are presented in this section.

3.1 Siamese Clone Detector

Siamese [18] is a scalable and rank-based clone detector designed to enhance both accuracy and efficiency. It combines multiple techniques, including multiple code representations, query reduction (QR), and a customized similarity ranking function to detect clones effectively. The detection process consists of two main phases: indexing and retrieval. In the indexing phase, the tool processes the source code corpus by generating four code representations: (i) the original source code (r_0), (ii) n-gram tokens without renaming (r_1), (iii) n-gram tokens with renamed identifiers, literals, and type tokens (r_2), and (iv) n-gram tokens with all tokens renamed (r_3).

These representations are obtained through code tokenization, a step in which the source code is broken down into smaller syntactic elements such as keywords, operators, and literals. After tokenization, Siamese constructs n-grams, i.e., sequences of n consecutive tokens, that help capture structural patterns and local context. This method enhances the tool's ability to detect the presence of small code variations like identifier renaming or command reordering. Additionally, by using n-grams, Siamese increases the robustness of clone search and can identify both syntactically and semantically similar code fragments.

In the *retrieval phase*, a code snippet is used as a query to search for potential clones within the indexed corpus. The query undergoes the same preprocessing as the corpus, including tokenization and the generation of the four code representations. Then, Siamese applies *query reduction* to each representation, removing common or non-informative tokens and n-grams. This reduction process favors rare and more distinctive tokens, aiming to increase retrieval precision. The aggressiveness of this filtering is controlled by the *qr threshold* parameter: lower threshold values result in a more selective reduction.

Refined queries retrieve similar code segments, which are then ranked and filtered by a similarity threshold (*simThreshold*) to identify clones. Using a two-phase approach with token-based representations and ranking, Siamese enables scalable and effective clone search in large codebases.

In this study, we configured Siamese with the following parameters: an *n-gram size* of 4, a *QR threshold* of 10, and a *boosting* factor also set to 4. The minimum clone size was set to 6 tokens, and the similarity thresholds were defined as {50%, 60%, 70%, 80%}. We adopted the parameter values from the original Siamese study [18]. These parameters control important aspects of clone detection, including the granularity of code representation (*n-gram size*), the aggressiveness of token filtering (*QR threshold*), the weighting of relevant matches (*boosting*), the minimum number of tokens required to consider a clone (*minCloneSize*), and the similarity thresholds used to filter final results.

Although the optimal configuration of these parameters may vary depending on the dataset characteristics, we adopted the default settings proposed in the original Siamese study, as they have proven effective across a wide range of scenarios. Furthermore, we manually validated a sample of the detected clones and observed high precision, which supports the adequacy of these default parameters for the goals of our study.

3.2 The CROP Dataset

CROP is a dataset that stores different code revisions by linking review data with full code snapshots taken at the moment of the review [13]. The dataset records code changes made during the code review process, including their respective revisions.

CROP retrieves snapshots of each revision from projects hosted on Gerrit. The dataset was built using commits to store the snapshots of project revisions in repositories traceable via

Table 1: Number of code reviews and revisions extracted from CROP

Projects	Code Reviews	Revisions	Collaborators
jgit	9,676	27,559	304
platform.ui	8,565	26,728	787
egit	7,121	17,196	180
couchbase-java-client	1,313	3,918	28
couchbase-jvm-core	1,179	3,249	23
spymemcached	563	1,456	48
Total	**28,417**	**80,106**	**1,370**

Git. Moreover, CROP labels snapshots as *before* when they refer to the state of the project prior to a revision's submission, and *after* when they refer to the state following the submission. This setup enables a deep analysis of the code review process since it provides a complete record of the revisions performed.

In this study, we used only the Java projects available in CROP due to Siamese's scope. Table 1 shows the projects included in this study, along with the number of reviews and revisions considered. For this study, we considered 28,417 and 80,106 reviews and revisions, respectively. Additionally, we included the number of contributors to provide more context on the scale and collaborative nature of the studied projects, in total we had 1,370 collaborators for all projects.

4 Methodology

In this empirical study, we conducted an exploration of code clones using Siamese [18] on six Java projects extracted from CROP [13] to investigate the lifecycle of code clones during code review. We aim to ask the following research questions:

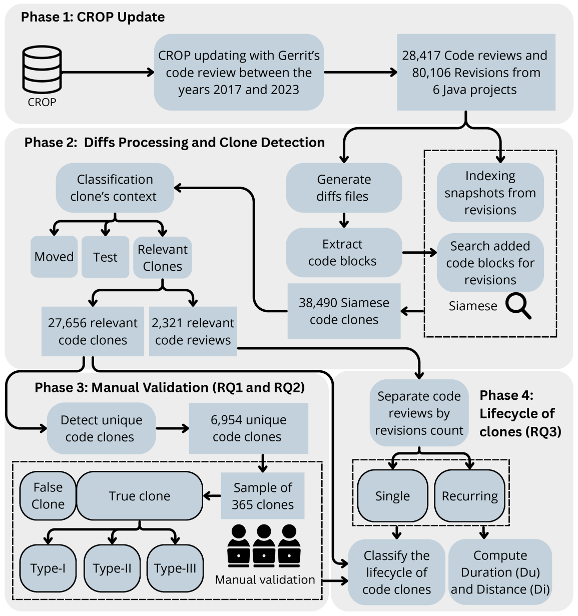
- *RQ1: Does code cloning occur during code review?*
- *RQ2: What types of code clones emerge during code review?*
- *RQ3: What is the lifecycle of code clones during code review?*

To answer the proposed RQs, we carried out a set of processes divided into four phases, as illustrated in Figure 2.

4.1 Phase 1: CROP Update

The original CROP dataset includes code review data up until 2017 [13]. Hence, we first performed an update to include code reviews from the years 2017 to 2023. The update was applied exclusively to the Java projects in CROP, through modifications to the original source code [12]. The following steps were carried out: **Language update** and **Adaptation to new Gerrit API**.

Language update: The mining code, originally written in Python 2.7, was migrated to Python 3.10. This update aimed to ensure compatibility with modern libraries and tools. **Adaptation to new Gerrit API:** CROP, developed in 2017, was designed to work with the structure and response formats of the Gerrit API as it existed at the time. However, Gerrit has since undergone updates that significantly changed its API, including modifications to URLs, query parameters, and data structure. The modifications consisted of updating HTTP requests and processing the responses returned by the API.



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
27,656 relevant code clones
2,321 relevant code reviews
38,490 Siamese code clones
Siamese
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 2: Empirical methodology employed in this study.

After updating CROP's source code, as described above, we ran the mining process to collect the new code review data. The updated mining process took approximately three months to execute. Table 1 showcases the code review data considered in this study. The updated CROP dataset, alongside all other artifacts created in this study, is available in our replication package [28].

4.2 Phase 2: Diffs Processing and Clone Detection

The goal of this phase is to identify code clones across revisions. To achieve this, we performed the same clone detection process for each of the 80,106 revisions extracted from CROP. For each revision, CROP provides the *before* and *after* snapshots of the codebase, representing the state of the code as it was right before and right after the revision, respectively. Hence, a *diff* between the *after* and *before* snapshots contains only the code changes made by the developers in the respective revision. This is necessary to avoid the Rebasing problem when working with code review data [15]. In short, to find potential clones introduced in a certain revision, we need to i) identify the newly added code, and ii) use Siamese to find clones between the new code and the existing code (*before* snapshot). This process is detailed next.

First, we take the revision's *before* snapshot and index it into Siamese. Internally, Siamese parses the entire codebase, preparing the code snippets to be compared against a certain query, as detailed in Section 3.1. Next, we generate a *diff* between the *after* and *before* snapshots, representing the set of changes performed in the revision. Once we generate the *diff* file for the revision, we extract the corresponding code blocks. We organize the code blocks into added and removed code blocks. Since we want to identify clones that are introduced, we discard the removed code blocks. Finally, we also discard all added code blocks smaller than six lines of code This is the minimum snippet size commonly used to detect code clones [9, 23, 30].

In the next step, we use the added code blocks longer than six lines as queries into Siamese. This way, if Siamese returns a match, it indicates that an added code block is a potential clone of an existing snippet in the codebase. This process is repeated for all added code blocks of all revisions considered in this study. This procedure took approximately two weeks to execute. In total, considering the 80,106 revisions, Siamese reported 38,490 code clones.

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. To address this, we cross-checked all reported clones with their original *diff* files and discarded 2,282 clones that came from moved snippets.

Siamese detects clones across the entire codebase, but not all are relevant to this study. Since cloning is common in test code [29], we excluded 8,275 clones from test files and directories. Additionally, 277 clones moved within test files were also discarded.

All clones not from moved or test code were considered relevant, resulting in 27,656 relevant clones, which represent approximately 72% of those reported by Siamese. These clones appeared in 2,321 code reviews. This set of clones and reviews was used in Phases 3 and 4 of our methodology, as described next.

4.3 Phase 3: Manual Validation

For this phase, initially, we filtered the 27,656 relevant clones identified in our methodology, as presented in Section 4.2. Since clone detection is performed independently for each revision, the same code snippet may be cloned multiple times within the same revision, across consecutive revisions, and across consecutive reviews. Hence, a manual validation that includes duplicated clones would not be representative.

To address this, we applied a filtering step to identify *unique clones*, i.e., unique code snippets that do not repeat within the set of detected relevant clones. Thus, for each relevant clone, we generated a hash value based on the code content. This approach allowed us to group identical clone instances together and reduce the set to a collection of entirely distinct clones. As a result, we obtained 6,954 *unique clones*.

Next, we selected a statistically representative sample from the set of *unique clones*. The sample was calculated using a 95% confidence level and a 5% margin of error, resulting in a set of 365 clones. We adopted uniform stratified sampling, where the distribution of sampled clones was proportional to the number of *unique clones* in each project. This ensured the sample was not dominated by clones from specific projects.

Table 2: Distribution of the total number of unique clones and the proportional selection of blocks for the sample and pilot per project.

Project	Population	Sample	Pilot
jgit	1031	54	5
platform.ui	4092	215	21
egit	998	52	5
couchbase-java-client	347	18	2
couchbase-jvm-core	192	11	1
spymemcached	294	15	2
Total	6954	365	36

With the sample selected, we conducted a manual analysis of the relevant clones detected by Siamese. Although Siamese showed high accuracy in detecting code clones in its original study [18], clone detectors can still produce false positives [30], which may affect the reliability of our empirical study. Furthermore, even though Siamese is capable of detecting Type-I to Type-III clones, it does not report the clone type when a clone match is found.

The manual analysis aimed to validate whether the sampled clones were true clones or false positives. In case of a true clone, the clone type (Type-I, Type-II or Type-III) was also indicated. The validation was conducted by three researchers with experience in code clone analysis. First, two researchers independently analyzed each clone. In cases of disagreement, either in terms of clone validity or type, a third researcher was consulted to make the final decision.

To ensure consistency and agreement among the evaluators, we conducted a pilot study with 37 clones (approximately 10% of the minimum required sample size). The clones included in the pilot study were not part of the representative sample.

Table 2 presents the total number of *unique clones* per project, along with the number of clones selected for the sample and pilot study.

4.4 Phase 4: Lifecycle of Clones

After our manual analysis of the clones presented in Section 4.3, we proceeded with the analysis of the lifecycle of the relevant code clones that appear during code review, i.e., how long the clones stay in the code review process, with the goal of answering RQ3 introduced in Section 4. The following describes the steps performed for code review classification according to the clones' behavior.

We initially used the 2,321 relevant code reviews and the 27,656 relevant code clones presented in Section 4.2. For the lifecycle analysis, we selected only the code reviews that contained more than one revision.

Table 3: 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.

As a result, we collected 1,718 relevant code reviews with more than one revision. We used these code reviews and categorized them by analyzing the number of times a clone appears across revisions. Code reviews in which a clone appears in only one revision were categorized as *Single*, while those in which clones appear in more than one revision were categorized as *Recurring*. We identified 224 and 1,258 code reviews as *Single* and *Recurring*, respectively. Additionally, a code review may include more than one clone. In cases where one clone appeared in only one revision and another persisted through multiple revisions, the code review is classified as belonging to the intersection of *Single* and *Recurring*. In this scenario, 236 code reviews were identified.

After categorizing the code reviews into *Single* and *Recurring*, we analyzed the clones in each category and identified their lifecycle, i.e., we determined when clones appeared and disappeared during a code review. Table 3 presents the categories used to assess the clone lifecycle. The *Single* category includes cases where clones appear in only one specific stage of the review process: at the beginning (*Early Stage*), middle (*Mid Stage*), or end (*Late Stage*).

The *Recurring* category captures more complex behaviors, including clones that span multiple stages, such as from the beginning to the middle (*Emerging Cycle*), from the middle, but not persisting to the end (*Central Cycle*), from the middle to the end (*Closing Cycle*), or throughout the entire review process (*Full Cycle*). Additionally, *Unstable Cycle* refers to reviews in which clones appear, disappear, and later reappear throughout revisions.

After categorizing the clone lifecycle, we conducted an investigation into the length of the period that clones remain in a code review and also the moment at which they appear. For this, we introduce two new metrics: *Duration* (Du) and *Distance* (Di), as described next.

Let $R = \{r_1, r_2, \dots, r_n\}$ be the ordered set of n revisions in a review. Consider a clone c that appears in a subset of revisions $R_c \subseteq R$. The *Duration* of clone c is defined as the proportion of revisions in which c is present.

The *Distance* metric evaluates the relative position of the first appearance of clone c within the revision sequence. Let i be the index in the revision sequence at which clone c is first introduced during the code review. The *Duration* and *Distance* metrics are defined in Equation 1.

$$Di(c) = \begin{cases} 0 & \text{if } i = 1 \\ \frac{i}{|R|} & \text{if } i > 1 \end{cases} \qquad Du(c) \quad = \frac{|R_c|}{|R|} \tag{1}$$

Thus, the values of Du and Di range from 0 to 1. Higher values of $Du(c)$ indicate that clone c persists throughout the review, while lower values of $Di(c)$ indicate clones that appeared earlier in the revision process. Together, these metrics provide insight into the lifecycle and the moment of introduction of clones in reviews composed of multiple revisions. We disregarded code reviews with only a single revision, as these do not allow us to understand the *Duration* and *Distance* of a clone within a code review.

4.4.1 Duration and Distance Computation. To illustrate the computation of the *Duration* and *Distance* metrics, we use review number 22961 from our motivating example. This review comprises four revisions, denoted as $R = \{r_1, r_2, r_3, r_4\}$. The clone under consideration was introduced in the first revision (r_1) and removed in the second revision (r_2), thus appearing only in r_1. Accordingly, the subset of revisions where the clone is present is $R_c = \{r_1\}$.

To compute the *Duration* of this clone, we consider the proportion of revisions in which it appeared: $Du(c) = \frac{|R_c|}{|R|} = \frac{1}{4} = 0.25$. The *Distance* is calculated based on the revision in which the clone is introduced: since the clone appears in the initial revision ($i = 1$), we apply the case $Di(c) = 0$.

These values demonstrate a short-lived clone that appeared early in the review process and was promptly removed, illustrating an effective review practice that addresses code duplication at an early stage.

5 Results

5.1 RQ1: Does code cloning occur during code review?

This research question serves as a sanity check for this empirical study. To investigate code cloning in code reviews, we first verify the presence of code duplication in MCR. Thus, RQ1 is addressed through manual analysis of a representative sample of 365 relevant clones, as detailed in Section 4.3.

Table 4: Results of the manual validation of clones reported by Siamese

Project	True Clones	False Positives
jgit	51	3
platform.ui	213	2
egit	45	7
couchbase-java-client	18	0
couchbase-jvm-core	10	0
spymemcached	15	0
Total	353	12

Table 4 presents the number of true clones and false positives identified in the manual validation. We can observe that 353 clones were validated as true clones, representing a precision rate of about 96.7%. This is in line with the precision reported in the original Siamese publication [18]. Hence, this result shows that Siamese is a suitable tool for this study.

With this level of precision, when extrapolating to the entire population, it is estimated that at least 26,743 true relevant clones were detected by Siamese. These clones were detected in a total of 2,321 code reviews. Hence, the average number of clones per review in which cloning occurs is 12. These results indicate that code cloning occurs during MCR.

Although *jgit* has the highest number of code reviews (9,676), as shown in Table 1, it is *platform.ui* that has the most true clones detected (213). This suggests that the quantity of clones does not directly correlate with the number of code reviews or revisions, but rather depends on other factors.

Additionally, *platform.ui* focuses on GUI components, suggesting that GUI projects may be more prone to code duplication due to repetitive layouts, widgets, and event handling. Developers often replicate similar UI patterns, such as dialogs or menus, across modules. This points to a new research direction: investigating whether visual interface systems generate more clones than other domains.

Answer to RQ1: Code cloning occurs during code review. In our study, we manually validated 365 clones and found 353 to be true positives. Considering this high precision, we estimate that approximately 26,743 of the 27,656 relevant clones identified are true clones.

5.2 RQ2: What types of code clones emerge during code review?

To address RQ2, we used the 353 true clones observed in the sample of 365 *unique clones*. Table 5 presents the number of clone types identified per project. We observed that 46.74% were Type-I clones, 7.9% were Type-II, and 45.3% were Type-III. These findings indicate that the most frequently occurring clone types during code review are Type-I and Type-III.

We observed that *platform.ui* has many Type-I clones, likely due to its focus on UI components where developers often duplicate code structures. The larger contributor base in *platform.ui* may also increase reuse of familiar solutions, resulting in more exact duplicates. In contrast, *jgit* shows more Type-III clones, reflecting different project characteristics.

Table 5: The number of clone types detected per project.

Project	Type-I	Type-II	Type-III
jgit	7	5	39
platform.ui	123	5	85
egit	17	11	17
couchbase-java-client	5	0	13
couchbase-jvm-core	3	4	4
spymemcached	10	3	2
Total	165	28	160

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. [24], 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.

Answer to RQ2: It is observed that the most frequent clone types are Type-I and Type-III, together representing more than 92% of the manually investigated clone samples.

5.3 RQ3: What is the lifecycle of code clones during code review?

We answer RQ3 by analyzing the clones and the relevant code reviews identified in Section 4.2.

Although our manual analysis was based on a representative 95% sample of the clone population, the high precision achieved by the Siamese clone detector gives us sufficient confidence to extend our lifecycle analysis to the broader population of clones identified in our dataset. We believe this approach offers a comprehensive perspective on how code clones behave during the review process. This is particularly relevant for syntactically identical clones that, despite their structural similarity, may follow distinct lifecycle patterns across different code reviews.

Table 6 presents the number of code reviews per project classified as *Single*, *Recurring*, and the intersection between these categories. In total, 224 and 1,258 code reviews were identified as *Single* and *Recurring*, respectively. The results show that most projects have *Recurring* code reviews. Without considering the code reviews intercessions, approximately 84.89% of the code reviews contain clones that persist across multiple revisions.

By categorizing code reviews as either *Single* or *Recurring*, we were able to evaluate their respective lifecycles. Thus, Table 7 shows the distribution of clone lifecycle stages within

Single code reviews. A total of 287 clones were observed in the *Early Stage*, 140 in the *Mid Stage*, and 131 in the *Late Stage*. With 51.44%, the majority of clones tend to appear in the first revision. The chance of a clone appearing in the middle or at the end of the code review is similar, at 25.09% and 23.48%, respectively. In other words, if a clone is not recurrent, it will not be submitted to the final project in 76.53% of cases.

In sequence, Table 8 presents the distribution of clone lifecycle stages within *Recurring* code reviews. We identified 224 clones in the *Emerging Cycle*, 134 in the *Central Cycle*, and 426 in the *Closing Cycle*. Additionally, 967 clones completed the *Full Cycle*, and 31 were categorized as undergoing an *Unstable Cycle*. Clones that span the entire code review (*Full Cycle*) represent the majority, accounting for 54.3% of the cases. Meanwhile, clones found primarily toward the final stages of the review (*Closing Cycle*) constitute the second-largest group, with 24.50%. Clones emerging or concentrating in the earlier stages of the review (*Emerging Cycle* and *Central Cycle*) together account for 21.01%. These data show that if the clone is recurrent, in 80.07% of cases the clone was submitted to the final project. Finally, clones with unstable behavior during the review are relatively rare, representing only 1.78% of the total.

After analyzing the life cycle of the clones, we used the 353 true clones identified in the sample presented in Section 4.3. We analyzed the distribution of clone types across their respective life cycle stages. Table 9 presents the results of this analysis. A joint analysis of Tables 7, 8, and 9 reveals a notable consistency between the population data and the sample. The *Full Cycle* and *Early Stage* categories remain the most frequent

Table 6: Distribution of code reviews with multiple revisions by project, classified as *Single, Recurring* and *Intersection*

Project	Single	Recurrings	Intersection
jgit	74	312	61
platform.ui	60	391	67
egit	73	355	64
couchbase-java-client	7	92	21
couchbase-jvm-core	7	85	13
spymemcached	3	23	10
Total	224	1,258	236

Table 7: Distribution of Clone Life Cycle Stages in *Single* Code Reviews

Project	Early Stage	Mid Stage	Late Stage
jgit	77	44	45
platform.ui	74	43	38
egit	93	36	36
couchbase-java-client	21	7	3
couchbase-jvm-core	14	5	2
spymemcached	8	5	7
Total	287	140	131

in both sets, reinforcing the robustness of the identified patterns. This proportionality also gave us greater confidence to extend the analysis to the entire clone population, using the proposed metrics to evaluate the clone lifecycle in the code review process. The clones that appear in *Single* code reviews are mostly Type-III clones, representing 64.52% of the cases. In *Recurring* code reviews, *Full Cycle* clones are the most prevalent, especially among Type-I (131 clones), followed by Type-III (94) and Type-II (19), indicating that most clones persist throughout the review. *Emerging Cycle* behavior is more common in Type-III clones (15), suggesting greater activity of syntactically dissimilar clones early in the review. *Central Cycle* clones are less frequent but again concentrated among Type-III clones (9). For *Closing Cycle*, Type-I leads with 10 instances, showing that exact duplicates often gain relevance toward the review's end. Overall, Type-I clones exhibit greater stability, while Type-III clones show more dynamic behavior across review stages.

Figure 3 compares the *Duration* and *Distance* of clones in *Single* and *Recurring* code reviews. In *Single* reviews, *Mid Stage* clones have the shortest and most variable duration, while *Late Stage* clones persist longer and appear farther from insertion. In *Recurring* reviews, *Central Cycle*, *Closing*, and especially *Full Cycle* clones have greater duration and distance, indicating more stability. *Unstable Cycle* clones always show zero duration, and *Full Cycle* have a fixed duration of one. *Distance* is zero for *Early Stage*, *Emerging Cycle*, and *Full Cycle* clones, and one for *Late Stage* clones.

The results presented in Table 10 indicate that, on average, clones persist throughout a significant portion of the review process, as reflected by an overall mean *Duration* of 0.71. Moreover, the low overall mean *Distance* of 0.17 suggests that clones are typically introduced at early stages of the review.

Among the analyzed projects, *platform.ui* shows the highest mean *Duration* of 0.85 and the earliest mean *Distance* of 0.09, indicating that clones are introduced early and tend to persist throughout the review process. This persistence suggests challenges in promptly resolving duplication issues, despite a code review culture.

In contrast, *spymemcached* exhibits the lowest mean *Duration* of 0.30 and the latest mean *Distance* of 0.37, indicating that clones are introduced relatively early and persist for a short period, possibly reflecting a more structured review process.

Answer to RQ3: We found that 54.3% of clones that appear in *Full Cycle* category. In general, clones survive the review process (0.71 average *Duration*) and emerge early (0.17 average *Distance*).

6 Discussion

In this section, we discuss our findings, limitations, and potential future directions of this study.

Our qualitative analysis showed that Type-I and Type-III clones are predominant during code review. Despite modern practices and tools, code duplication is still frequent. Detecting

Table 8: Distribution of Clone Lifecycle Stages in *Recurring* Code Reviews

Project	Emerging Cycle	Central Cycle	Closing Cycle	Full Cycle	Unstable Cycle
jgit	67	50	105	214	13
platform.ui	79	43	138	300	8
egit	58	27	116	275	8
couchbase-java-client	8	5	34	86	1
couchbase-jvm-core	4	5	24	71	1
spymemcached	8	4	9	21	0
Total	224	134	426	967	31

Table 9: Distribution of Clone Lifecycle Stages by Clone Type

	Single			Recurring				
Type	Early Stage	Mid Stage	Late Stage	Emerging Cycle	Central Cycle	Closing Cycle	Full Cycle	Unstable Cycle
Type-I	4	2	0	8	1	10	131	0
Type-II	3	1	1	0	2	2	19	0
Type-III	12	6	2	15	9	9	94	1
Total	19	9	3	23	12	21	244	1

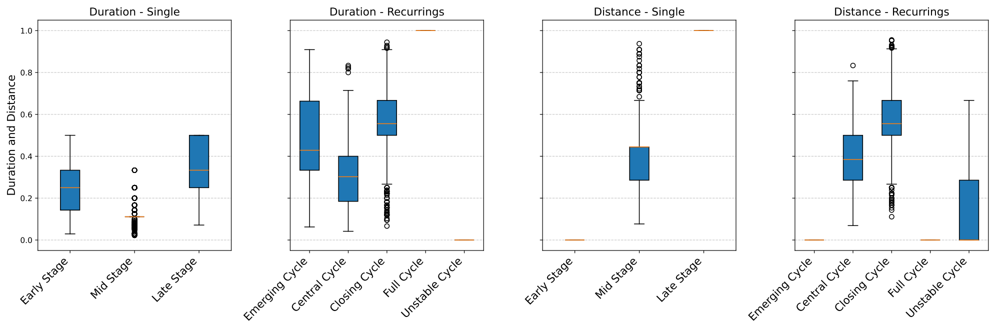


Figure 3: Boxplots illustrating *Duration* and *Distance* metrics across the lifecycle stages of *Single* and *Recurring* code review clones.

Table 10: Mean Duration and Distance of the Clones.

Project	Duration	Distance
jgit	0,59	0,24
platform.ui	0,85	0,09
egit	0,67	0,22
couchbase-java-client	0,73	0,18
couchbase-jvm-core	0,77	0,17
spymemcached	0,30	0,37
All Projects	0,71	0,17

clones early is important, as they often persist after merging, complicating maintenance and debugging. Two main factors contribute to this.

(1) Lack of Perception: Developers and reviewers may simply not perceive the presence of duplicated code. Without clear indications or tool support, these clones can easily go unnoticed during code review. As a result, developers might be unaware that a duplicated fragment exists elsewhere in the codebase. This unawareness becomes especially critical when a bug is introduced, as developers may fix the issue in one location but overlook its cloned counterpart, potentially leaving the system vulnerable to failures.

(2) Subtle Differences in Type-III Clones: Detecting clones manually is particularly difficult when the duplicated code has slight structural or syntactic variations. Type-III clones, by nature, often contain subtle edits that obscure their similarity. These differences imply a reduced likelihood of identification during manual reviews, increasing the chance that such clones will merged into the final project without due consideration.

7 Threats to the Validity

We discuss the threats to our study's validity according to Wohlin et al. [33] guidelines on experimentation.

Conclusion and Internal threats: During code review, moved code blocks can be misidentified as clones, since only added blocks are considered. To address this, we analyze the removed blocks. However, small refactorings still hinder detection. As future work, we propose using Siamese to detect moved blocks with minor changes. The analysis considered only code blocks with at least six lines, excluding comments and other non-code elements. Despite challenges distinguishing comments from operators like /, we removed lines containing only comments. Isolated lines like closing braces () were kept, as they represent micro clones that may affect development [11].

A possible threat to validity is that the clone lifecycle analysis used the entire dataset, while manual validation covered only a sample. However, the high accuracy in manual inspection, where 353 out of 365 clones were confirmed as true positives, and similar lifecycle distributions between the sample and full dataset give us confidence in our analysis. Still, minor differences could exist if all data were manually validated.

Construct threats: Clone type classification can be imprecise, as Siamese returns whole methods even if only parts are added, making manual validation challenging. To reduce ambiguity, two researchers independently analyzed each case, with a third resolving disagreements. All researchers have experience in code clone analysis.

External threats: The dataset used CROP [13] which is specifically built from code review data hosted on the open-source platform Gerrit. Although Gerrit is widely adopted by large open-source communities and companies, its review workflows and usage patterns may differ from those observed in other platforms, such as GitHub, GitLab, or Phabricator. Furthermore, other threat stems from the clone detection tool employed in this study. Siamese [18] is specifically designed to work with code written in Java. As such, our analysis and findings are restricted to Java projects.

8 Related Work

This section discusses related empirical studies focusing on other aspects of code quality in code review.

Uchôa et al. [27] analyze the evolution of design degradation during reviews and across revisions, using review discussions and 16 metrics to assess its relation to code review practices. Approximately 14,971 code reviews from 7 software projects are analyzed. Paixão et al. [16] present an empirical study that inspects and classifies developers' intentions regarding refactorings performed during code review. In this study, 1,780 code changes across 6 open-source applications are analyzed, and developers' intentions are categorized into 7 groups.

Pascarella et al. [17] examined the impact of code review on code smells and convention violations across 21,000 reviews. Assessing severity with code metrics, they found that, in 16,442 Eclipse reviews and 1,268 manually analyzed cases, only a minority of convention violations were addressed after review comments.

Jiang et al. [6] analyze how changes made to pull requests affect the code review process in pull request-based development projects. By examining 104,307 pull requests from 9 GitHub projects, the authors observed that approximately 34.56% of the pull requests analyzed had modifications after their initial submission.

Although prior studies address aspects of code quality like refactoring, design degradation, and code smells, none specifically examine the lifecycle of code clones during code review. To fill this gap, this study propose two novel metrics (*Duration* and *Distance*) to quantify clone persistence and timing within the review process, offering new insights not explored in previous work.

9 Conclusion

In this paper, we conducted an empirical study on the lifecycle of code clones during the modern code review process. We leveraged the Siamese clone detector to analyze six open-source projects from the CROP dataset, for which we detected a total of **38,490** code clones. We analyzed **27,656** relevant code clones and their corresponding code reviews.

After removing duplicate clones from our set of relevant clones, we identified approximately **6,954** unique clones and performed a manual analysis on them. We found that **96.7%** of the clones were true positives, with the majority being classified as Type-I (46.74%) and Type-III (45.3%) clones.

We analyzed the lifecycle of relevant clones in code reviews with multiple revisions, calculating *Duration* and *Distance* metrics. With an average *Duration* of 0.71, we found that most clones tend to survive the code review process and are merged into the project. Furthermore, we observed that most clones emerge early in the code review, as the average *Distance* is 0.17.

As future work, we plan to qualitatively analyze discussion threads from code reviews containing relevant clones. Our goal is to understand whether and how developers address code clones during reviews, specifically, if clones are explicitly mentioned, what concerns or actions are raised, and which strategies or practices are used when reviewers acknowledge them. This analysis may provide deeper insights into developer awareness, communication, and decision-making regarding code cloning in practice.

ARTIFACT AVAILABILITY

All artifacts, including datasets, scripts, and documentation from this study are available in our replication package [28].

ACKNOWLEDGMENTS

This work received partial funding from CNPq-Brazil, Universal grant 404406/2023-8, and support from FUNCAP (BP6-00241-00276.01.00/25).

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] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 712–721.

[3] Jason Cohen. 2010. Modern code review. *Making Software: What Really Works, and Why We Believe It* (2010), 329–336.

[4] Dror G Feitelson, Eitan Frachtenberg, and Kent L Beck. 2013. Development and deployment at facebook. *IEEE Internet Computing* 17, 4 (2013), 8–17.

[5] Judith F Islam, Manishankar Mondal, and Chanchal K Roy. 2016. Bug replication in code clones: An empirical study. In *2016 IEEE 23Rd international conference on software analysis, evolution, and reengineering (SANER)*, Vol. 1. IEEE, 68–78.

[6] Jing Jiang, Jiangfeng Lv, Jiateng Zheng, and Li Zhang. 2021. How developers modify pull requests in code review. *IEEE Transactions on Reliability* 71, 3 (2021), 1325–1339.

[7] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE transactions on software engineering* 28, 7 (2002), 654–670.

[8] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2016. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering* 21 (2016), 2146–2189.

[9] Manishankar Mondai, Chanchal K Roy, and Kevin A Schneider. 2018. Micro-clones in evolving software. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 50–60.

[10] Manishankar Mondal, Banani Roy, Chanchal K Roy, and Kevin A Schneider. 2019. Investigating context adaptation bugs in code clones. In *2019 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 157–168.

[11] Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. 2018. Micro-clones in Evolving Software. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 70–80. doi:10.1109/SANER.2018.8330196

[12] Matheus Paixao, Jens Krinke, Donggyun Han, and Mark Harman. [n. d.]. Codebase Repository of CROP. https://github.com/crop-repo/crop

[13] 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.

[14] Matheus Paixao, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman. 2019. The impact of code review on architectural changes. *IEEE Transactions on Software Engineering* 47, 5 (2019), 1041–1059.

[15] Matheus Paixao and Paulo Henrique Maia. 2019. Rebasing in code review considered harmful: A large-scale empirical investigation. In *2019 19th international working conference on source code analysis and manipulation (SCAM)*. IEEE, 45–55.

[16] Matheus Paixão, Anderson Uchôa, Ana Carla Bibiano, Daniel Oliveira, Alessandro Garcia, Jens Krinke, and Emilio Arvonio. 2020. Behind the intents: An in-depth empirical study on software refactoring in modern code review. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 125–136.

[17] Luca Pascarella, Davide Spadini, Fabio Palomba, and Alberto Bacchelli. 2019. On the effect of code review on code smells. *arXiv preprint arXiv:1912.10098* (2019).

[18] 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.

[19] 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.

[20] 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.

[21] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: a case study at google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice*. 181–190.

[22] G Shobha, Ajay Rana, Vineet Kansal, and Sarvesh Tanwar. 2021. Code clone detection—a systematic review. *Emerging Technologies in Data Mining and Information Security: Proceedings of IEMIS 2020, Volume 2* (2021), 645–655.

[23] Denis Sousa, Matheus Paixao, Chaiyong Ragkhitwetsagul, and Italo Uchoa. 2024. Code Clone Configuration as a Multi-Objective Search Problem. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 503–509.

[24] 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.

[25] Christopher Thompson and David Wagner. 2017. A large-scale study of modern code review and security in open source projects. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*. 83–92.

[26] 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.

[27] Anderson Uchôa, Caio Barbosa, Willian Oizumi, Publio Blenílio, Rafael Lima, Alessandro Garcia, and Carla Bezerra. 2020. How does modern code review impact software design degradation? an in-depth empirical study. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 511–522.

[28] Italo Uchoa, Denis Sousa, Matheus Paixão, Pedro Maia, Anderson Uchôa, and Chaiyong Ragkhitwetsagul. 2025. Replication Package for the paper: 'An Exploratory Study on the Lifecycle of Code Clones during Code Review". https://zenodo.org/records/16746835

[29] Brent van Bladel and Serge Demeyer. 2021. A comparative study of test code clones and production code clones. *Journal of Systems and Software* 176 (2021), 110940.

[30] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. 2013. Searching for better configurations: a rigorous approach to clone evaluation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 455–465.

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

[32] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 261–271.

[33] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[34] 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), 204–218.

[35] 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

[36] Fiorella Zampetti, Gabriele Bavota, Gerardo Canfora, and Massimiliano Di Penta. 2019. A study on the interplay between pull request review and continuous integration builds. In *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 38–48.