

A Mixed-Methods Study on the Lifecycle of Code Clones in Code Review

Italo Uchoa [State University of Ceará |italo.uchoa@aluno.uece.br]

Denis Sousa [State University of Ceará |denis.sousa@aluno.uece.br]

Pedro Maia [State University of Ceará |pedro.ulisses@aluno.uece.br]

Anderson Uchôa [Federal University of Ceará and North Carolina State University |andersonuchoa@ufc.br; agoncal2@ncsu.edu]

Chaiyong Ragkhitwetsagul [Mahidol University |chaiyong.rag@mahidol.edu]

Sávio Freire [Federal Institute of Ceará and State University of Ceará |savio.freire@ifce.edu.br]

Matheus Paixao [State University of Ceará |matheus.paixao@uece.br]

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 can propagate to all related clones, significantly impacting the project's maintainability. Nevertheless, there remains a gap in research on the presence and behavior of code clones during code review. By leveraging the CROP dataset (over 28k code reviews and 80k revisions) and the Siamese clone detector, we identified 27,656 relevant code clones that underwent code review across 6 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, depending on whether 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). 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. To extend the lifecycle analysis, we also conducted a qualitative analysis of a stratified sample of code reviews, investigating whether developers discuss code clones during review and the reasons why clones are removed or retained. We observed that reviews with explicit discussions about code clones are more likely to occur when the clone is not merged. In particular, among the 315 sampled reviews, 10 of the 11 reviews discussing clones in the context of refactoring (90.9%) involved cases where the clone was not merged. In contrast, 5 of the 7 reviews with *Unjustified / Unclear Cloning* discussions (71.4%) involved clones that were merged. Given the limited number of clone-related discussions, these observations should be interpreted as exploratory rather than conclusive evidence.

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 (McIntosh et al., 2016; Uchôa et al., 2021). This practice is common among large companies such as Microsoft (Bacchelli and Bird, 2013), Google (Sadowski et al., 2018), and Facebook (Feitelson et al., 2013). 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 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 (Yu et al., 2016), code smells (Pascarella et al., 2019), security vulnerabilities (Thompson and Wagner, 2017), architectural changes (Paixao et al., 2021), among others.

Code clones are fragments of code that are copied and reused on other (or the same) codebase, often with minor modifications (Roy et al., 2014; Ain et al., 2019). Code cloning increases productivity in implementing new features and is often introduced intentionally into the codebase (Wang and Godfrey, 2014). However, if an error is identified in the code clone segment, it means that all clones must be corrected, which hinders the system's maintainability (Islam et al., 2016; Mondal et al., 2019; Ragkhitwetsagul et al., 2021). 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 (Zakeri-Nasrabadi et al., 2023).

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. In this review, the developer added a method called `getStyleOverride` within the `StackRenderer` class. However, this method was

```
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` in project `eclipse.platform.ui`

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 method pull-up refactoring operation was performed (Fowler and Beck, 1999).

In review number 101346 of the same project, we identified 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 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, combining quantitative and qualitative analyses to understand their presence and behavior during code review. Our goal is to provide insights for developers, tool builders, and researchers not only on how clones evolve during reviews, but also on the underlying reasons and justifications that influence whether clones are removed or retained in the codebase.

By leveraging the CROP dataset (Paixao et al., 2018) and the Siamese (Ragkhitwetsagul and Krinke, 2019) 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 at which they appear and disappear in the code review.

Furthermore, two metrics are proposed to assess the persistence and introduction of code clones in code reviews. The *Duration* metric measures the number of revisions a clone remains during the code review. Additionally, *Distance* measures the revision in which the first occurrence of the clone appears. Overall, the results from *Duration* and *Distance* suggest that clones tend to be introduced early in the review process and often persist throughout multiple revisions, frequently being merged into the codebase.

Finally, we conducted a comprehensive qualitative analysis of 1,718 code reviews containing more than one revision. In this step, we investigated whether developers explicitly discuss code clones during the review process and aimed to understand the motivations behind clone removal or retention. To this end, we categorized these discussions into different themes, capturing the main focus of what developers addressed. From a manually analyzed sample of 315 reviews, we identified 32 cases containing explicit discussions about code clones. Most of these discussions (68.8%) occurred in reviews where the clone was ultimately not merged into the codebase.

Our findings also reveal distinct behavioral patterns. Reviews in which clones are not merged tend to include discussions focused on refactoring opportunities, with 90.9% of the refactoring-related discussions occurring in reviews where the clone was not merged. In contrast, reviews where clones are merged often lack explicit discussion or justification, where 71.4% of the discussions that did not provide justification for cloning were associated with reviews where the clone was merged.

The main contributions of this paper are listed as follows:

- A dataset (Uchoa et al., 2026) of manually validated

code clones linked to code reviews.

- An empirical study to analyze the lifecycle of code clones during code review, including proposed categories and two novel metrics (*Duration* and *Distance*) to measure clone persistence and introduction timing.
- A qualitative analysis of developer discussions about code clones, suggesting why clones are removed or retained during code review.
- A replication package (Uchoa et al., 2026) containing all artifacts necessary to reproduce this study, including the code for quantitative and qualitative analyses, as well as the complete set of results.

This study extends our previous work by Uchoa et al. (2025b), which first introduced an exploratory analysis of the lifecycle of code clones during code review. An extended abstract of this work was published in the *CTIC-SBQS 2025 Undergraduate Research Competition* (Uchoa et al., 2025a), where it received an award. The present manuscript substantially expands the original study by introducing two new research questions that investigate whether developers explicitly discuss code clones during code review and the reasons why clones are removed or retained. To answer these new research questions, we introduce a qualitative analysis of developer discussions, supported by additional manual analyses and a broader discussion of the findings.

2 Background

2.1 Modern Code Review

Modern Code Review (MCR) was introduced by Cohen (2010) and popularized by Bacchelli and Bird (2013). 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 (Pascarella et al., 2019; Paixão et al., 2020; Uchoa et al., 2020).

This practice is widely adopted by major companies such as Microsoft (Bacchelli and Bird, 2013), Facebook (Feitelson et al., 2013), and Google (Sadowski et al., 2018). 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 (Bacchelli and Bird, 2013). 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 Gerrit¹ and GitHub², which enable asynchronous reviews with inline comments. Integration with version control and CI/CD pipelines further streamlines the process within agile workflows (Zampetti et al., 2019).

¹<https://www.gerritcodereview.com/>

²<https://docs.github.com/en/pull-requests>

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. (2023), 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 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 (Ain et al., 2019; Shobha et al., 2021; Zakeri-Nasrabadi et al., 2023), often combining multiple detection approaches. The literature (Ain et al., 2019; Zakeri-Nasrabadi et al., 2023) 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 (Kamiya et al., 2002; Ragkhitwetsagul and Krinke, 2019) 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 (Wang et al., 2020). 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 (Ragkhitwetsagul and Krinke, 2019) 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 (Zakeri-Nasrabadi et al., 2023).

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 (Ragkhitwetsagul and Krinke, 2019) 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. Next, 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 (Ragkhitwetsagul and Krinke, 2019). 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 (Paixao et al., 2018). 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 Git. Moreover, CROP labels snapshots as *before* when they refer to the state of the project prior to a revision's

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

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 Research Methodology

In this empirical study, we investigated code clones using Siamese (Ragkhitwetsagul and Krinke, 2019) to six projects drawn from CROP (Paixao et al., 2018), with the objective of classifying code reviews according to the lifecycle behavior of their code clones. We seek to address the following research questions (RQs):

- *RQ1: Does code cloning occur during code review?*
This question establishes whether code cloning is sufficiently present during Modern Code Review to justify a deeper investigation of its characteristics, lifecycle, and developer practices.
- *RQ2: What types of code clones emerge during code review?*
After confirming the presence of code clones, this question characterizes the predominant clone types, providing a better understanding of the nature of code duplication observed during code review.
- *RQ3: What is the lifecycle of code clones during code review?*
This question investigates how code clones evolve throughout the review process, including when they are introduced, how long they persist, and whether they are ultimately retained or removed.
- *RQ4: Do developers discuss code cloning during code review?*
Beyond understanding clone evolution, this question examines whether developers explicitly identify and discuss code cloning during the review process, providing insights into reviewers' awareness of code duplication.
- *RQ5: What are the reasons that lead code clones to be removed or retained during code review?*
Finally, this question investigates the motivations underlying developers' decisions, helping explain why some

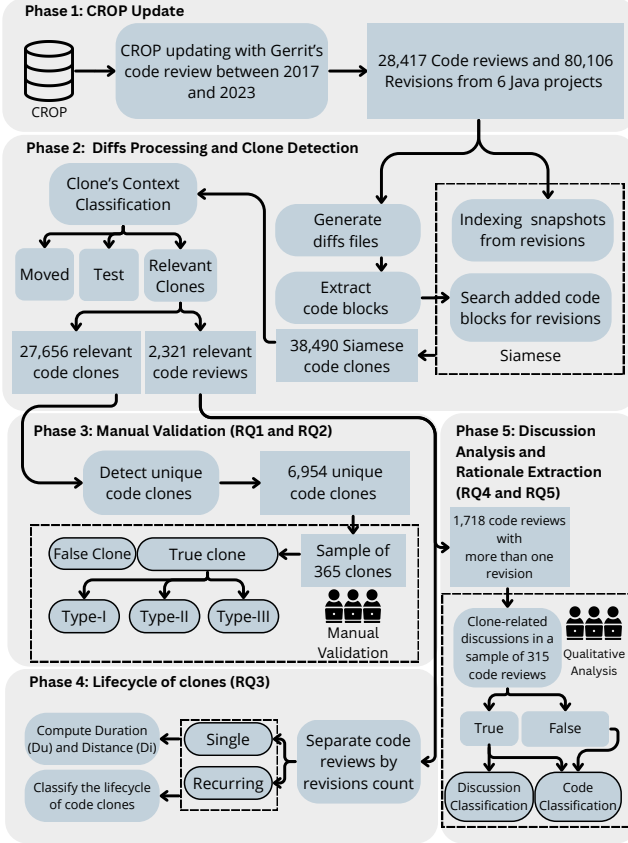


Figure 2. Empirical methodology employed in this study.

clones are removed whereas others are ultimately retained in the codebase.

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

4.1 Phase 1: CROP Update

The original CROP dataset includes code review data up until 2017 (Paixao et al., 2018). 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. 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.

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 (Uchoa et al., 2026).

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 (Paixao and Maia, 2019). 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 (Wang et al., 2013; Mondai et al., 2018; Sousa et al., 2024).

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 (van Bladel and Demeyer, 2021), 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 identified multiple times within the same revision, across consecutive revisions, and even across different reviews. Therefore, including duplicated clones in the manual validation would not be representative, as it could lead to repeated analysis of the same examples, potentially biasing the results by overrepresenting certain clones and reducing the diversity of the evaluated sample.

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.

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 (Ragkhitwetsagul and Krinke, 2019), clone detectors can still produce false positives (Wang et al., 2013), 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 authors 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 36 clones (approximately 10% of the minimum required sample size). The clones included in the pilot study were not part of the representative sample. During the pilot study, both reviewers agreed that all candidate pairs were valid clones, obtaining observed agreement (Po) and expected agreement (Pe) are equal to 100%, making Cohen's Kappa undefined due to the absence of rating variability. For clone type classification, the pilot achieved Po = 69.44%, Pe = 48.61%, and $\kappa = 0.40$, motivating the refinement of the annotation guidelines and the involvement of a third reviewer to resolve disagreements during the subsequent analyses.

For the main manual validation, clone validation achieved

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	1,031	54	5
platform.ui	4,092	215	21
egit	998	52	5
couchbase-java-client	347	18	2
couchbase-jvm-core	192	11	1
spymemcached	294	15	2
Total	6,954	365	36

Po = 95.07% and Pe = 95.00%. We report Po and Pe instead of Cohen's Kappa because this binary task presented a highly imbalanced class distribution with very few disagreements, making κ less informative. All disagreements were subsequently reviewed and resolved by a third reviewer.

The manual classification of clone types achieved Po = 61.38%, Pe = 44.19%, and $\kappa \approx 0.31$, indicating a fair level of agreement between the reviewers. Most disagreements occurred when distinguishing between Type-I and Type-III clones because the clone detection tool presents the entire method containing the detected clone to provide contextual information, requiring reviewers to manually determine the exact clone boundaries. Consequently, one reviewer occasionally considered only the identical duplicated statements (Type-I), whereas the other considered adjacent modified statements within the surrounding method context (Type-III). All disagreements were subsequently reviewed and resolved by a more experienced third reviewer with expertise in both the clone detection tool and clone type classification.

Table 2 presents the total number of *unique clones* per project, along with the number of clones selected for the sample and pilot study. The results of the manual validation answer RQ1 and RQ2.

The manual validation confirmed that 353 of the 365 sampled clones were true positives. Since this validation indicates that the vast majority of the relevant clones identified by Siamese correspond to actual code clones, the subsequent lifecycle analysis (Phase 4) is performed on the complete set of 27,656 relevant clone instances rather than on the manually validated sample. In contrast, the set of 6,954 unique clones is used exclusively for sampling and manual validation (RQ1 and RQ2), while the full set of relevant clone instances is retained for the large-scale lifecycle analysis (RQ3).

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. The goal of this phase is answering RQ3 introduced in Section 4. This analysis builds upon the manual validation presented in Phase 3, which confirmed the reliability of the detected clones. Based on this validation, the lifecycle analysis was conducted on the complete set of relevant clone instances identified in Phase 2, addressing RQ3 independently of the qualitative analysis presented in Phase 5. The following de-

scribes 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. Although the lifecycle of each clone is analyzed individually, the unit of analysis reported throughout this study is the code review. Specifically, every clone contained in a review is tracked across its revisions, and the review is subsequently assigned to one or more lifecycle categories according to the observed behavior of its clones. For the lifecycle analysis, we selected only the code reviews that contained more than one revision.

As a result, we collected 1,718 relevant code reviews with more than one revision. For each selected code review, we analyzed the lifecycle of every contained code clone by tracking the revisions in which that clone appeared throughout the review process. Based on these clone-level lifecycles, the corresponding code review was assigned to one or more lifecycle categories.

All code reviews analyzed in this study were ultimately merged into the project's main branch, as they originate from the CROP dataset. Therefore, throughout this study, the terms *merged* and *unmerged* refer exclusively to the lifecycle of the code clone within a review, not to the outcome of the review itself. A clone is considered *merged* if it remains present in the last revision of the review and is therefore incorporated into the final version of the codebase. Conversely, a clone is considered *unmerged* if it disappears before the last revision.

A code review was classified as *Single* if it contained at least one clone that appeared in only a single revision. Likewise, it was classified as *Recurring* if it contained at least one clone that persisted across multiple revisions. We identified 224 and 1,258 code reviews as *Single* and *Recurring*, respectively. A code review may contain multiple relevant clones, and different clones within the same review may exhibit different lifecycle behaviors. Consequently, the *Single* and *Recurring* categories are not mutually exclusive. For example, if one clone appears in only a single revision while another persists across multiple revisions, the same review is assigned to both categories. Therefore, the totals reported for these categories may exceed the total number of analyzed code reviews. In our dataset, 236 code reviews belong to both categories.

After assigning code reviews to the *Single* and *Recurring* groups based on the lifecycle of their constituent clones, we further characterized the observed clone lifecycles using the categories presented in Table 3. 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.

4.4.1 Duration and Distance Computation

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.

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

The special case $Di(c) = 0$ when $i = 1$ was introduced to ensure that clones present from the first revision are assigned zero distance, consistently representing their introduction at the beginning of the review. Using the expression $\frac{i}{|R|}$ for all cases would assign a positive value even when a clone first appears in the initial revision, which does not match the intended interpretation of the metric.

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.

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.

4.5 Phase 5: Discussion Analysis and Rationale Extraction

In this phase, we extended our analysis to investigate how code cloning is discussed during code review and the rationale behind cloning decisions. We started from the 2,321

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.

relevant code reviews identified in Phase 2 and filtered only those containing more than one revision, resulting in 1,718 reviews. This filtering step ensured consistency with the lifecycle analysis and enabled a comparison between code clones that were retained and those that were removed. Unlike Phase 3, which manually validated a representative sample of unique code clones, this phase relies on a different stratified sample of code reviews with more than one revision. For each sampled review, we extracted the non-automatic review discussions and conducted the qualitative analysis exclusively on those relevant discussions related to code cloning, thereby providing the basis for answering RQ4 and RQ5.

Next, we conducted a qualitative analysis based on a statistically representative sample of these 1,718 reviews. Using a 95% confidence level and a 5% margin of error, we obtained a stratified sample of 315 code reviews. For each selected review, we analyzed both the code changes and the associated discussion.

Initially, we verified whether the discussions within each code review explicitly addressed code cloning. This step allowed us to quantify how often cloning is discussed and to relate these discussions to whether the corresponding code changes were retained or removed, thus addressing RQ4. The analysis was performed independently by two researchers, i.e., investigators. In cases of disagreement, a third researcher acted as a tie-breaker, following the same protocol adopted in Section 4.3.

After identifying reviews that contained clone-related discussions, the two primary investigators first held a calibration meeting to establish a common understanding of what should be considered evidence of reasoning about code cloning. Next, they independently extracted the discussion excerpts they considered most representative of clone-related reasoning, focusing on text that explained why duplicated code was removed, retained, or otherwise discussed during the review process.

The independently extracted excerpts were then compared, and disagreements were subsequently resolved through a consensus meeting between the two investigators, resulting in a single agreed set of discussion excerpts. Based on this final set, the same investigators collaboratively performed an iterative thematic coding process, grouping excerpts according to recurring themes observed across the discussions. The categories were refined during the coding process until consensus was reached, resulting in the discussion behaviors presented in Table 4. This step supports answering RQ5.

Additionally, for all analyzed reviews, including those without explicit clone-related discussions, we performed a manual classification of the duplicated code according to a predefined taxonomy derived from Kapser and Godfrey (2008). The four categories are: *Forking*, *Templating*, *Customization*, and *Exact Match*. These categories are shortly described in Table 4. The full definition of each category alongside examples can be found in the original study and we computed the Observed Agreement (Po), Expected Agreement (Pe), and Cohen's Kappa (κ). For binary classification tasks with highly imbalanced class distributions, we report Po and Pe because Cohen's Kappa may become less informative under these conditions.

The identification of clone-related discussions achieved high inter-rater agreement, with Po = 89.84% and Pe = 83.32%. We report Po and Pe instead of Cohen's Kappa because this binary task presented a highly imbalanced class distribution with very few disagreements, making κ less informative. All disagreements were subsequently reviewed and resolved by a third reviewer.

The qualitative coding of the clone-related discussion excerpts achieved high inter-rater agreement, with Po = 87.94%, Pe = 33.64%, and $\kappa = 0.82$, indicating almost perfect agreement between the reviewers. The remaining disagreements were subsequently resolved through consensus between the two primary reviewers before the final categories were established.

Furthermore, we consider a fifth category for cases in which the duplicated code does not fit any of the predefined patterns. The classification process followed the same protocol adopted in Section 4.3 and at the beginning of this phase: two researchers independently classified each instance, and in cases of disagreement, a third researcher acted as a tie-breaker. This procedure ensured consistency and allowed us to identify patterns and motivations behind why code clones are maintained or removed during code review, thereby supporting the analysis of RQ5.

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 need to verify the presence of code duplication in

Table 4. Clone-related discussion categories and code clone patterns identified through thematic coding in the qualitative analysis

Type	Category	Definition
Discussion Behaviors	Refactoring for Reuse	Clones are removed through abstraction (e.g., methods or classes).
	Direct Clone Removal	Clones are removed without further discussion or redesign.
	Justified Cloning	Clones are retained with explicit technical or practical justification.
	Temporary Cloning	Clones are introduced as a short-term solution with planned refactoring.
Clone Patterns (Kapsner and Godfrey (2008))	Unjustified Cloning	Clones are introduced without explicit reasoning or justification.
	Forking	Code is copied to enable independent evolution, often diverging structurally over time.
	Templating	Code is reused as a template with minor parameter changes, typically resembling boilerplate.
	Customization	Code is copied and significantly adapted to meet new requirements or constraints.
	Exact Match	Code is duplicated with minimal or no changes, preserving structure and behavior.

Table 5. 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

MCR. Thus, RQ1 is addressed through manual analysis of a representative sample of 365 relevant clones, as detailed in Section 4.3.

Table 5 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 (Ragkhitwetsagul and Krinke, 2019). 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.

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.

Table 6. 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

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 6 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.

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. (2014), 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 simply renaming elements.

Answer to RQ2: We 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 7 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' intersections, approximately 84.89% of the code reviews contain clones that persist across multiple revisions.

Table 7. 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

By categorizing code reviews as either *Single* or *Recurring*, we were able to evaluate their respective lifecycles. Thus, Table 8 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 merged in 76.53% of cases. Specifically, the *Single* code reviews shown in Table 7 give rise to 558 observed *Single* lifecycle cases, of which 131 (23.48%) correspond to clones that remained until the last revision and were therefore merged into the final version of the code, whereas the remaining 427 cases (76.53%) correspond to clones that disappeared before the last revision and were not merged. Although the unit of classification is the code review, a single review may contain multiple clones exhibiting different *Single* lifecycle behaviors. Consequently, the same review may contribute to more than one lifecycle category, causing the total number of lifecycle cases to exceed the number of *Single* code reviews.

Table 8. 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

Table 9 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 a clone is recurrent, it is merged into the final version of the code in 78.17% of cases. Specifically, the *Recurring* code reviews shown in Table 7 give rise to 1,782 observed *Recurring* lifecycle cases, of which 1,393 (78.17%) correspond to *Closing Cycle* or *Full Cycle*, meaning that the clone remained present in the last revision and was therefore merged into the final version of the code. The remaining 389 cases (21.83%), corresponding to *Emerging Cycle*, *Central Cycle*, and *Unstable Cycle*, refer to clones that disappeared before the last revision and were therefore not merged. Although the unit of classification is the code review, a single review may contain multiple clones exhibiting different *Recurring* lifecycle behaviors. Consequently, the same review may contribute to more than one lifecycle category, causing the total number of lifecycle cases to exceed the number of *Recurring* code reviews. Finally, clones with unstable behavior during the review are relatively rare, representing only 1.78% of the total.

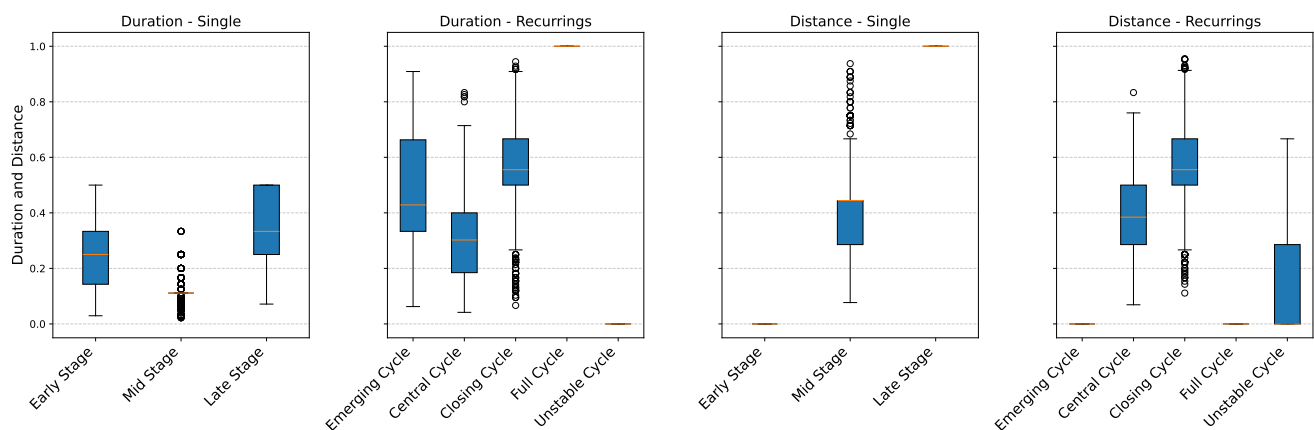
After analyzing the lifecycle of the clones, we looked at the 353 true clones identified in the sample presented in Section 4.3. We analyzed the distribution of clone types across their respective lifecycle stages. Table 10 presents the results of this analysis. A joint analysis of Tables 8, 9, and 10 reveals a notable consistency between the population data and the sample. The *Full Cycle* and *Early Stage* categories remain the most frequent 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 clones) and Type-II (19 clones), indicating that most clones persist throughout the review. *Emerging Cycle* behavior is more common

Table 9. 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 10. Distribution of Clone Lifecycle Stages by Clone Type

Type	Single			Recurring				
	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.

in Type-III clones (15 clones), 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 clones). 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 were assigned a *Duration* of zero due to their intermittent behavior of appearing, disappearing, and reappearing throughout the review, making a continuous duration undefined. Consequently, they were excluded from the overall *Duration* analysis. *Full Cycle* clones 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 11 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.

Table 11. 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

Table 12. Distribution of discussion categories across reviews with merged and non-merged clones

Category	Merged	Unmerged	Total
Direct Removal	2	3	5
Justified Cloning	1	5	6
Refactoring for Reuse	1	10	11
Temporary Cloning	1	2	3
Unjustified / Unclear Cloning	5	2	7

Answer to RQ3: Most clone lifecycle cases belong to long-lived categories, with *Full Cycle* accounting for 54.3% of the cases, indicating that clones often persist throughout the entire review. In general, clones survive the review process (0.71 average *Duration*) and emerge early (0.17 average *Distance*).

5.4 RQ4: Do developers discuss code cloning during code review?

To answer RQ4, we conducted a qualitative manual analysis focused on the discussions associated with code reviews, as detailed in Section 4.5. From the 315 sampled code reviews, we identified 32 reviews (10.16%) that explicitly contained discussions related to code cloning.

During the independent extraction of clone-related discussion excerpts, the two investigators disagreed on 15 of the 32 identified clone-related discussions. These disagreements were subsequently resolved through a consensus meeting before the thematic coding process was conducted, ensuring that the final categorization was based on a single agreed set of discussion excerpts. Based on this agreed set, the discussions were then categorized into five groups using thematic coding. To better understand how discussions relate to review outcomes, we compared the distribution of these categories across reviews with merged and non-merged clones, as presented in Table 12.

Although clone-related discussions represent a minority of the analyzed reviews, only 32 out of the 315 sampled reviews contained explicit discussions about code cloning. Among these, 22 discussions involved clones that were not merged (68.8%). This observed distribution suggests that clone-related discussions may be more common in reviews where duplicated code is ultimately removed from the codebase. Given the limited number of clone-related discussions, these findings should be interpreted as exploratory evidence rather than definitive conclusions.

Regarding the types of discussions observed, reviews with

Table 13. Distribution of cloned code categories across reviews with merged and non-merged clones

Category	Merged	Unmerged	Total
Customization	62	77	139
Exact Match	57	47	104
Forking	4	11	15
Templating	31	21	52

clones that were removed are more frequently associated with *Refactoring for Reuse* (10 out of 11 cases—90.9%) and *Justified Cloning* (5 out of 6 cases—83.3%). In contrast, reviews with clones that were retained show a higher concentration of *Unjustified / Unclear Cloning*, with 5 out of 7 cases (71.4%) in this category.

These exploratory results suggest that clone-related discussions tend to appear more often in reviews where clones are not merged, and that the discussion content may vary according to the review outcome.

Answer to RQ4: Code cloning is discussed during code review, although in a minority of cases (32 out of 315 reviews). When code cloning is discussed, it appears more often in reviews where clones are removed, suggesting a tendency towards rejection or revision of changes.

5.5 RQ5: What are the reasons that lead code clones to be removed or retained during code review?

To answer RQ5, we combined two perspectives: (i) the qualitative classification of the 32 clone-related discussions, and (ii) the classification of duplicated code patterns across all 315 sampled reviews, as described in Section 4.5.

First, from the code classification perspective, all identified clones were categorized according to the taxonomy derived from Kapser and Godfrey (2008). Table 13 presents the distribution of code reviews according to the predefined categories. The reviews are further grouped based on whether the corresponding clones were ultimately retained or removed. Additionally, in 5 cases, the duplicated code could not be classified into any of the predefined categories.

The results indicate that, when considering code reviews with clones that were not merged into the codebase (i.e., instances in the *unmerged* group), there is a higher proportion of *Customization* and *Forking* patterns. Specifically, *Customization* accounts for 77 out of 139 instances (55.4%), representing more than half of the cases in this category. Similarly, *Forking* appears in 11 out of 15 instances (73.3%), indicating a strong concentration of this pattern among clones that are not merged. These types of clones typically involve more extensive modifications or structural divergence, suggesting more complex or experimental changes that may require further refinement and are therefore less likely to be accepted.

Conversely, for code reviews with clones that were merged into the codebase (i.e., instances in the *merged*

group), we observe a higher concentration of *Exact Match* and *Templating* patterns. In particular, *Exact Match* corresponds to 57 out of 104 instances (54.8%), while *Templating* accounts for 31 out of 52 instances (59.6%). These patterns reflect more standardized and straightforward forms of code reuse, typically involving minimal or well-understood modifications, which may better align with existing practices and facilitate their acceptance during code review.

From a discussion perspective, the qualitative analysis helps explain how cloning decisions are made during code review. As shown in Table 12, 10 of the 11 observed *Refactoring for Reuse* discussions (90.9%) and 5 of the 6 *Justified Cloning* discussions (83.3%) occurred in reviews where the clone was not merged. Although these counts are small, they suggest that, when duplication is explicitly discussed in these contexts, it is often critically examined and may lead to refactoring decisions or rejection.

In contrast, 5 of the 7 observed *Unjustified / Unclear Cloning* discussions (71.4%) occurred in reviews where the clone was merged. While this observation is based on a limited number of cases, it suggests that accepted clones may sometimes be introduced without extensive discussion or explicit justification. When combined with the prevalence of *Exact Match* and *Templating* patterns observed earlier, this indicates that simpler and more standardized forms of duplication are more likely to be accepted with minimal scrutiny.

Taken together, these exploratory findings suggest that both the structural characteristics of the cloned code and the presence (or absence) of discussion may influence whether a clone is ultimately removed or retained in the codebase. Although discussions about code clones are relatively rare (32 out of 315 reviews), 22 of these 32 discussions (68.8%) were associated with clones that were ultimately not merged. Given the limited number of clone-related discussions, these observations should be interpreted as initial evidence rather than definitive conclusions.

Answer to RQ5: Code clones are more likely to be merged when they involve straightforward reuse patterns, such as *Exact Match* and *Templating*. In contrast, clones related to *Customization* and *Forking*, which involve more complex or divergent changes, are more likely to be discussed and removed.

6 Discussion

In this section, we discuss the main findings of our study and their implications.

Our qualitative analyses revealed that Type-I and Type-III clones are predominant during code review. Despite modern development practices and tool support, code duplication remains frequent. Early detection of clones is essential, as they often persist throughout the review process and are eventually retained in the codebase, increasing maintenance effort and the risk of propagating defects. Two main factors contribute to this behavior.

(1) Lack of Perception: Developers and reviewers may simply not perceive the presence of duplicated code. With-

out clear indications or tool support, these clones can easily go unnoticed during code review. This observation is reinforced by our second qualitative analysis, in which only 32 out of 315 manually analyzed reviews contained explicit discussions about code clones. Moreover, when clones are explicitly discussed, they are often removed or refactored. Although this observation is based on only 32 clone-related discussions, it provides initial evidence that increased awareness during code review may contribute to mitigating unnecessary duplication. 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 be retained in the final project without due consideration.

Beyond these factors, our exploratory results also reveal distinct behavioral patterns between clones that are retained and those that are removed. Clones that are removed tend to be associated with more complex reuse patterns, such as *Customization* and *Forking*, which often involve structural modifications and are more likely to trigger discussion and critical evaluation. In contrast, retained clones are predominantly linked to simpler patterns, such as *Exact Match* and *Templating*, reflecting more straightforward and standardized reuse. From a discussion perspective, removed clones are frequently associated with refactoring or justified cloning decisions, whereas retained clones are often introduced without explicit justification. Together, these exploratory findings suggest that simpler and more familiar forms of duplication may be more easily accepted during code review, while more complex or visible cases of cloning appear more likely to be scrutinized and, ultimately, rejected. Although our manual validation showed high precision (96.7%), the lifecycle analysis depends on the clones detected by Siamese. The original evaluation of Siamese reported 99% recall for Type-I, Type-II and Type-III clones, and 88% recall for Moderately Type-III clones, indicating high effectiveness for the clone types considered in this study. Nevertheless, recall decreases for more challenging clone instances, which may result in some clones not being included in our analysis (Raghitwet-sagul and Krinke, 2019).

This behavior aligns with prior findings in the context of software architecture, where discussions during code review have been shown to correlate with higher-quality architectural changes (Paixao et al., 2021). Similarly, our results indicate that when developers explicitly discuss code clones, they are more likely to take corrective actions, such as refactoring or removal. Furthermore, prior research suggests that as software evolves, its design tends to degrade over time (Uchôa et al., 2020). Although this degradation is not specifically tied to code cloning, our findings point to a related phenomenon in the context of code review. In particular, the absence of discussions about code clones may contribute to the accumulation of duplicated code, representing

a form of design degradation that emerges during the review process.

7 Threats to the Validity

We discuss the threats to our study's validity according to Wohlin et al. (2012) guidelines on experimentation.

Conclusion and Internal Validity: 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 in 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 (Mondai et al., 2018).

A further threat to validity arises from the fact that the clone lifecycle analysis was conducted on the entire dataset, whereas the manual validation relied on a sample. Nevertheless, the high accuracy observed during manual inspection, where 353 out of 365 clones were confirmed as true positives, combined with the similar lifecycle distributions between the sample and the full dataset, increases our confidence in the results. Even so, minor differences might still exist if the entire dataset were manually validated.

An additional concern relates to the qualitative analysis used to identify the reasons why code clones are removed or retained during code review. Since qualitative coding is inherently subjective and may introduce inconsistencies, two researchers independently analyzed the discussions and subsequently met to compare and consolidate their findings. Any disagreements were resolved collaboratively with the support of a third researcher with extensive experience in qualitative research in Software Engineering.

Construct Validity: 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 Validity: The dataset used CROP (Paixao et al., 2018), 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, another threat stems from the clone detection tool employed in this study. Siamese (Ragkhitwetsagul and Krinke, 2019) is specifically designed to work with code written in Java. As such, our analysis and findings are restricted to Java projects and may not be generalized to other programming languages.

8 Related Work

This section presents empirical studies that investigate different aspects of code quality and developer behavior during the code review process.

Uchôa et al. (2020) investigate the evolution of design degradation throughout code reviews and across their revisions. By analyzing review discussions and leveraging 16 software metrics, the authors explore how design degradation manifests and evolves during the review process. Their study is based on a large dataset of approximately 14,971 code reviews from seven software projects, providing evidence on how review practices influence design quality over time.

Similarly, Paixão et al. (2020) focus on understanding developers' intentions when performing refactorings during code review. The authors analyze 1,780 code changes across six open-source systems and classify refactoring intentions into seven distinct categories. Their findings highlight that refactoring is often driven by multiple factors, including maintainability improvements and design adjustments identified during the review process.

In another line of work, Pascarella et al. (2019) examine the effectiveness of code review in addressing code smells and coding convention violations. Their study spans over 21,000 code reviews, including 16,442 reviews from the Eclipse ecosystem and a manually analyzed subset of 1,268 cases. The results indicate that only a limited portion of detected issues are actually resolved after review comments, suggesting that code review does not always lead to improvements in code quality.

Focusing on developer behavior in pull request-based workflows, Jiang et al. (2021) analyze how modifications made after the initial submission of pull requests affect the review process. By studying 104,307 pull requests from nine GitHub projects, they report that approximately 34.56% of pull requests undergo changes after submission, emphasizing the iterative nature of modern code review.

Despite these efforts, existing studies primarily focus on aspects such as refactoring, design degradation, code smells, and developer activity during reviews. To the best of our knowledge, no prior work has specifically investigated the lifecycle of code clones within the code review process. To address this gap, our study introduces two novel metrics, *Duration* and *Distance*, which quantify how long clones persist and when they are introduced during code review. In addition, we perform a qualitative analysis of developer discussions to better understand the factors influencing whether clones are ultimately merged or removed, providing deeper insights into the decision-making process surrounding code cloning during code review.

9 Conclusion

In this paper, we conduct an empirical study on the lifecycle of code clones during the modern code review process. We leverage 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 analyze **27,656** rel-

evant code clones and their corresponding code reviews.

After removing duplicate clones from our set of relevant clones, we identify approximately **6,954** unique clones and performed a manual validation 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 also conduct qualitative analyses focusing on developer discussions in code reviews. From a manually analyzed sample of 315 reviews, we identified that only **10.16%** (32 out of 315) contained explicit discussions about code clones. Among these cases, clones were predominantly removed, accounting for **68.8%** of the discussions. These findings indicate that while explicit discussions about cloning are relatively rare, they are strongly associated with clone removal, highlighting the importance of developer awareness and communication during the review process.

We further analyze the lifecycle of relevant clones in code reviews with multiple revisions, calculating the *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 retained in 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 extend this study by exploring datasets beyond CROP, evaluating other clone detection tools in addition to Siamese that support multiple programming languages, and analyzing different code review platforms beyond Gerrit, such as GitHub. These directions aim to improve the generalizability and robustness of our findings.

ARTIFACT AVAILABILITY

All artifacts, including datasets, scripts, and documentation from this study, are available in our replication package (Uchoa et al., 2026).

ACKNOWLEDGEMENTS

This work was partially supported by the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), Brazil [Grant 404406/2023-8, 201708/2025-6, 403304/2025-3] and by Fundação Cearense de Apoio ao Desenvolvimento Científico e Tecnológico (FUNCAP), Brazil [Grant BP6-00241-00276.01.00/25]. **Generative AI Use Statement.** During the preparation of this manuscript, the authors used ChatGPT (OpenAI, GPT-5.5) to assist with language editing, writing refinement, and the translation of selected passages. All AI-assisted content was carefully reviewed, validated, and revised by the authors. All analyses, methodological decisions, interpretations, and conclusions were developed exclusively by the authors, who assume full responsibility for the final content of the manuscript.

References

- Ain, Q. U., Butt, W. H., Anwar, M. W., Azam, F., and Maqbool, B. (2019). A systematic review on code clone detection. *IEEE access*, 7:86121–86144.
- Bacchelli, A. and Bird, C. (2013). Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE.
- Cohen, J. (2010). Modern code review. *Making Software: What Really Works, and Why We Believe It*, pages 329–336.
- Feitelson, D. G., Frachtenberg, E., and Beck, K. L. (2013). Development and deployment at facebook. *IEEE Internet Computing*, 17(4):8–17.
- Fowler, M. and Beck, K. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.
- Islam, J. F., Mondal, M., and Roy, C. K. (2016). Bug replication in code clones: An empirical study. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 1, pages 68–78. IEEE.
- Jiang, J., Lv, J., Zheng, J., and Zhang, L. (2021). How developers modify pull requests in code review. *IEEE Transactions on Reliability*, 71(3):1325–1339.
- Kamiya, T., Kusumoto, S., and Inoue, K. (2002). Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE transactions on software engineering*, 28(7):654–670.
- Kapser, C. J. and Godfrey, M. W. (2008). “cloning considered harmful” considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692.
- McIntosh, S., Kamei, Y., Adams, B., and Hassan, A. E. (2016). An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21:2146–2189.
- Mondal, M., Roy, C. K., and Schneider, K. A. (2018). Micro-clones in evolving software. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 50–60.
- Mondal, M., Roy, B., Roy, C. K., and Schneider, K. A. (2019). Investigating context adaptation bugs in code clones. In *2019 IEEE International conference on software maintenance and evolution (ICSME)*, pages 157–168. IEEE.
- Paixao, M., Krinke, J., Han, D., and Harman, M. (2018). CROP: Linking Code Reviews to Source Code Changes. In *Proceedings of the IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 46–49. Association for Computing Machinery.
- Paixao, M., Krinke, J., Han, D., Ragkhitwetsagul, C., and Harman, M. (2021). The impact of code review on architectural changes. *IEEE Transactions on Software Engineering*, 47(5):1041–1059.
- Paixao, M. and Maia, P. H. (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)*, pages

- 45–55. IEEE.
- Paixão, M., Uchoa, A., Bibiano, A. C., Oliveira, D., Garcia, A., Krinke, J., and Arvonio, E. (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*, pages 125–136.
- Pascarella, L., Spadini, D., Palomba, F., and Bacchelli, A. (2019). On the effect of code review on code smells. *arXiv preprint arXiv:1912.10098*.
- Ragkhitswatsagul, C. and Krinke, J. (2019). Siamese: scalable and incremental code clone search via multiple code representations. *Empirical Software Engineering*, 24(4):2236–2284.
- Ragkhitswatsagul, C., Krinke, J., Paixao, M., Bianco, G., and Oliveto, R. (2021). Toxic code snippets on stack overflow. *IEEE Transactions on Software Engineering*, 47(3):560–581.
- Roy, C. K., Zibran, M. F., and Koschke, R. (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)*, pages 18–33. IEEE.
- Sadowski, C., Söderberg, E., Church, L., Sipko, M., and Bacchelli, A. (2018). Modern code review: a case study at google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice*, pages 181–190.
- Shobha, G., Rana, A., Kansal, V., and Tanwar, S. (2021). Code clone detection—a systematic review. *Emerging Technologies in Data Mining and Information Security: Proceedings of IEMIS 2020, Volume 2*, pages 645–655.
- Sousa, D., Paixao, M., Ragkhitswatsagul, C., and Uchoa, I. (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*, pages 503–509.
- Svajlenko, J., Islam, J. F., Keivanloo, I., Roy, C. K., and Mia, M. M. (2014). Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480. IEEE.
- Thompson, C. and Wagner, D. (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*, pages 83–92.
- Uchoa, A., Barbosa, C., Coutinho, D., Oizumi, W., Assunção, W. K., Vergilio, S. R., Pereira, J. A., Oliveira, A., and Garcia, A. (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)*, pages 471–482. IEEE.
- Uchoa, A., Barbosa, C., Oizumi, W., Blenílio, P., Lima, R., Garcia, A., and Bezerra, C. (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)*, pages 511–522. IEEE.
- Uchoa, I., Sousa, D., Maia, P., Uchoa, A., Ragkhitswatsagul, C., Freire, S., and Paixao, M. (2026). Replication Package for the paper: “A Mixed-Methods Study on the Lifecycle of Code Clones in Code Review”. <https://zenodo.org/records/19774835>.
- Uchoa, I., Sousa, D., Paixao, M., Maia, P., Uchoa, A., and Ragkhitswatsagul, C. (2025a). Exploring code clone behavior in modern code review. pages 606–608.
- Uchoa, I., Sousa, D., Paixao, M., Ulisses, P., Uchoa, A., and Ragkhitswatsagul, C. (2025b). An exploratory study on the lifecycle of code clones during code review. In *Brazilian Symposium on Software Engineering (SBES)*, Recife, Brazil.
- van Bladel, B. and Demeyer, S. (2021). A comparative study of test code clones and production code clones. *Journal of Systems and Software*, 176:110940.
- Wang, T., Harman, M., Jia, Y., and Krinke, J. (2013). Searching for better configurations: a rigorous approach to clone evaluation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 455–465.
- Wang, W. and Godfrey, M. (2014). Investigating intentional clone refactoring. *Electronic Communications of the EASST*, 63.
- Wang, W., Li, G., Ma, B., Xia, X., and Jin, Z. (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)*, pages 261–271. IEEE.
- Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media.
- Yu, Y., Wang, H., Yin, G., and Wang, T. (2016). Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? *Information and software technology*, 74:204–218.
- Zakeri-Nasrabadi, M., Parsa, S., Ramezani, M., Roy, C., and Ekhtiarzadeh, M. (2023). A systematic literature review on source code similarity measurement and clone detection: techniques, applications, and challenges. *Journal of Systems and Software*, 204:111796.
- Zampetti, F., Bavota, G., Canfora, G., and Di Penta, M. (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)*, pages 38–48. IEEE.