Received: 22 December 2017 | Revised: 1 August 2018 | Accepted: 3 September 2018

DOI: 10.1002/smr.2115

RESEARCH ARTICLE - EMPIRICAL

WILEY Journal of Software: Evolution and Process

An empirical study on how project context impacts on code cloning

Ricardo Perez-Castillo | **Mario Piattini**

Institute of Technology and Information Systems, University of Castilla-La Mancha, Ciudad Real, Spain

Correspondence
Ricardo Perez-Castillo, Institute of Technology and Information Systems, University of Castilla-La Mancha, Paseo de la Universidad 4, 13071 Ciudad Real, Spain.
Email: ricardo.pdelcastillo@uclm.es

Funding information
JCCM, Grant/Award Number: SBPLY/17/180501/000293; FEDER Fund; Spanish Ministry of Economy and Competitiveness, Grant/Award Numbers: TIN2015-63502-C3-1-R and TIN2015-70259-C2-1-R

Abstract

Code cloning can seriously affect software quality. Code clones are various fragments of syntactically or semantically equivalent code. Some authors argue that code clones have a negative impact on maintainability and understandability, since clones propagate defects and make it mandatory to pay attention to several copies. However, other authors believe clones are not necessarily bad, since self-admitted clones favor system stability and allow developers to move projects forward. Although some root causes and effects of cloning have been widely studied, there is not much relevant work analyzing how certain projects context factors impact on code cloning. This work presents an empirical validation of six open source projects by considering certain factors from Git repositories measured throughout a total of 70 releases for the 6 systems. The factors analyzed were the number of commits and committers per release, the average size of the commits and the size of the system in each release. The main conclusion obtained from the study is that, while the number of commits and committers and the system size do not significantly affect cloning, larger commits lead to a higher cloning ratio. These insights contribute to predicting and preventing code cloning, thus enabling a software quality improvement.

KEYWORDS

code cloning, development context, empirical study, git

1 | INTRODUCTION

The quality of code, and that of software in general, has been studied extensively, since it is recognized to be very significant as regards achieving competitiveness in the software industry.[1-3] One key factor that affects code quality is code cloning.[4,5] Code clones are two fragments of source code that are syntactically or semantically identical to each other.[6] According to Saini et al,[7] it is possible to identify four different types of cloning concerning relations between two code fragments and based on the nature of the similarity in their text or meaning.

- Type 1 (Exact clones): Two code fragments are exact copies of each other. This does not consider whitespaces, blanks, and comments.
- Type 2 (Renamed): Two code fragments are equal, with the exception of the names of variables, types, literals, and functions. This means that the abstract syntax tree is the same, regardless of the names of the code elements.
- Type 3 (Gapped clones): Two copied code fragments are similar, but modifications have been made to them, such as adding or removing statements, or the use of different identifiers, literals, types, whitespaces, layouts, and comments.
- Type 4 (Semantic clones): Two code fragments are semantically similar, regardless of their syntactic similarity.

 © 2018 John Wiley & Sons, Ltd.

According to Patil et al,[8] code cloning typically occurs as a consequence of reuse and programming approaches. The reuse approach is the simple activity of reusing by means of copy/paste, design, functionalities or logic, and may sometimes also occur because of common language constructs. The programming approach, meanwhile, comprises the accidental merging of two similar systems, system development with a generative programming approach, and an intentional delay in restructuring, ie, self-admitted technical debt.[9]

In the case of the copy/paste and reusing strategies, the addition of code clones can reduce the time and effort required by software developers at the outset, when these clones are introduced into the code base. If code clones are introduced afterwards, during system evolution, they can seriously affect maintainability, and clones specifically have a direct impact on modifiability/changeability and readability[10]:

- Code clones, and particularly those of a copy-paste nature, affect software quality, since they make it more difficult to maintain, update, or otherwise change the program. When an error is detected in one of the clones, the developer must find all the other copies and make parallel changes in order to be consistent and prevent bugs that degrade code quality.
- Code clones also harm software quality because they can make the understanding of a system more difficult. For example, the crucial difference between two nearly identical copies may be hidden by their overwhelming similarity.

Despite the fact that the impact of code cloning on software quality is broadly recognized by the research community, there is a long-standing debate regarding whether clones are really harmful or beneficial. Alternative approaches argue that cloned code does not necessarily impact on software quality in a negative manner and that, for example, code with a higher cloning ratio is more stable than non-cloned code.[11] The assumption of some sort of technical debt similarly sometimes leads to the admittance of certain code clones; this is not necessarily bad if developers know how to pay this debt back, thus enabling projects to move forward.[12] These two controversial viewpoints are important, since the results of code cloning studies provided in literature could be analyzed and interpreted in different ways.

The relevance of code cloning and its impact on software quality had led to the emergence of an intensive research effort on this topic during last two decades. Despite this effort, few works analyze factors that may lead to a greater cloning ratio and these studies, therefore, draw certain conclusion regarding how code cloning may be forecasted and prevented. For example, Harder[13] states that multiple distributed developers with certain communication deficiencies (such as those that may occur in some open source projects) can affect the propagation of code cloning.

As occurs in the aforementioned work, the goal of this paper is to analyze (through the use of an empirical study) how some factors in the development project context affect code cloning. This study specifically attempts to provide answers to the following research questions:

- How does the development effort in project releasing affect the evolution of code cloning?
- How does the size of commits in project releasing affect the evolution of code cloning?
- How does the size of the system affect the evolution of code cloning?

The main contribution of this paper is a case study conducted with six open source systems (OSSs) retrieved from GitHub, which analyzes the evolution of the factors related to the questions above as regards the code cloning fluctuation throughout the full history of 70 releases. After analyzing the results of the empirical validation, we discovered the factor that affects cloning to the greatest extent is the size of the commits. Additionally, this insight is tried to be explained through a qualitative research. The main implication for researchers and practitioners is that the empirical evidence provided by this study can be used to help predict and prevent code cloning in other projects.

The remainder of this paper is organized as follows: Section 2 presents work related to this research. Section 3 briefly introduces the mixed research method. Section 4 presents the design and planning of the case study according to the guidelines proposed by Runeson.[14] Section 5 shows the experimental results obtained after conducting the case study on six open source projects and its quantitative analysis. Section 6 presents the qualitative analysis related to the quantitative analysis' results. Finally, Section 7 sets out our conclusions and directions for future work.

2 | RELATED WORK

Code cloning is a wide area of research, and many works dealing with this issue have appeared in literature for at least the last two decades. These works can be classified in at least four lines, the first of which concerns studies that analyze the effect of code cloning on several quality code features, maintainability, or maintenance effort and cost. The second concerns how to detect different kinds of clones by proposing different techniques and tools. Having detected clones, the third research line concerns how to refactor code in order to reduce cloning, or at least manage code clones and consequently limit their harmful effects. Despite such a stunning research effort, comparatively fewer works analyze factors that can lead to a higher number of clones and, therefore, draw conclusion regarding how to predict and prevent code clones by dealing with these factors. Most of the recent key works in these research lines are presented in the following subsections.

2.1 | Clone effects

Mondal et al[4] provide an empirical study on the impacts of clones on software maintenance. This study deals with the unclear effects of code cloning on software maintenance. Some researchers argue that clones have negative impacts on software quality and maintenance, as cloning increases software maintenance cost. Furthermore, they claim that inconsistent changes to clones may introduce faults during evolution.[5] Nevertheless, other researchers argue that cloned code is more stable than non-cloned code.[11]

Mondal et al[15] also present an empirical study on how bugs are propagated through code cloning in different releases. According to the study carried out on thousands of commits of four open-source systems, up to 33% of clone fragments with some bug-fix changes can propagate bugs.

Similarly to that which occurs in our study, Forbes et al[16] searched for code clones in various open source project repositories. In addition to describing the code clones detected, this work provides *DoppelCode*, a tool with which to visualize code clones (type 1, type 2, and type 3) and their local and global impact on code, ie, the amount and similarity of clones found in the same module or in external, dependent modules.

2.2 | Clone detection

Several approaches and techniques can be employed to detect clones, such as that of Haque et al,[17] who propose a generic technique with which to detect code clones from several input source codes by segmenting the code into a number of sub-programs, modules, or functions. This technique can detect type 1 to 4 clones. Mondal et al[18] investigate which of the clone fragments detected have high possibilities of containing bugs in order to prioritize them for refactoring and tracking so as to help minimize future bug-fixing tasks. In a previous study, the same authors[19] provide a comparative study on the intensity and harmfulness of late propagation in near-miss code clones.

Alternatively, Kononenko et al[20] analyze the results of clone detection in compiled Java code as regards clones detected in source code. This study shows that source code and bytecode clone detection can produce significantly different results, especially in the case of large programs. Tiarks et al[21] analyze type-3 clones detected by means of state-of-the-art tools and investigate how syntactic differences in type-3 clones can be used to derive certain semantic abstractions, which can subsequently be used to determine whether the clone candidate suggested by the tool is a real type-3 clone from a human's viewpoint.

Although researchers have attempted to detect code clones for decades, some of the proposed approaches fail to scale to the size of the ever-growing systems code base. This lack of scalability prevents software developers from readily managing code clones and their related bugs. As a result, Kim et al[22] propose *VUDDY*, an approach for the scalable detection of vulnerable code clones. This work achieves a higher scalability by leveraging function-level granularity and a length-filtering technique that reduces the number of signature comparisons. In this respect, Patil et al[8] detect duplicate code in efficient manner by using decentralized computing and code reduction.

The detection of semantic clones (type 4) is the most difficult owing to the challenge of defining and implementing semantic similarity functions. Priyambadha and Rochimah[23] provide a semantic clone detection technique based on program dependence graphs. Similarly, Sheneamer and Kalita[6] consider abstract syntax trees and program dependency graphs. The innovative aspect of this approach is the representation of a pair of code fragments as a vector and the use of machine learning algorithms to detect clones.

Saha et al[24] develop a clone genealogy extractor and study different dimensions of how clone groups evolve with the evolution of the software systems. Like our paper, this work provides an in-depth empirical study with which to evaluate clone genealogies in evolving OSSs at the release level.

2.3 | Clone refactoring and tracking

Once code clones have been detected, there are two possible ways in which to deal with them, depending on the particular consideration. If code clones are considered to be harmful for maintainability, those clones are usually removed by means of code refactoring. If the code cloning impact is not, however, considered negative, the clones must be tracked and managed appropriately.

On one hand, Tsantalis et al[25] propose an approach with which to automatically evaluate whether a pair of clones can be refactored without causing side effects. This signifies that clones determined as refactorable can be refactored without causing any compile errors or test failures. Chen et al[26] present a technique that can be used to manage clone refactoring by matching some refactoring pattern templates with the code base. This technique makes it possible to summarize the refactoring changes of clones and detect the anomalous clone instances that are not consistently refactored.

On the other hand, Duala-Ekoko and Robillard[27] propose a technique with which to track clones in evolving software, which trusts in the concept of abstract clone region descriptors. This technique considers clone regions within methods in a way that is independent from the exact text of the clone region or its location in a file.

2.4 | Clone prediction and prevention

Recent studies on the evolution of code clones show that only some of the code clones change consistently during the evolution of the system. Wang et al[28] analyze how to accurately predict whether a code clone will undergo consistent changes. The work, therefore, provides useful recommendations to developers on leveraging the convenience of some code cloning operations while avoiding other code cloning operations in order to reduce a future consistency maintenance effort. A similar approach based on Bayesian networks is proposed by Zhang et al[29] and predicts clone consistency requirement at the time when changes have been made to a clone group.

Other studies use clone genealogy to predict fault propagation. Clone genealogy is the set of states and changes undergone by clone fragments over time that form an evolution history (owing to, eg, a changed clone fragment being left in an inconsistent state). For example,[30] examine clone genealogies to identify the fault-prone patterns of states and changes, thus enabling that faults to be predicted. Unlike our study, the intention of this research was to explore factors that are correlated with the fault-proneness of code clones rather than cloning itself.

Zibran et al[31] present a study on the evolution of near-miss clones at release level in medium to large open source software systems. The work investigates the evolution of both exact and near-miss clones and forecasts the number of clones in future releases of the software systems. Like our paper, the study analyses the evolution of code clones and their relationships with different factors, such as the programming language or paradigm and the program size.

Bladel et al[32] conducted a similar large-scale empirical study with open-source Java projects to investigate how the number of clones changes throughout software evolution, along with the tendency of individual developers to introduce clones. Although this study analyzes the fluctuation in cloning, it does not provide an in-depth analysis of the correlation with factors that can affect cloning.

Similarly to that which occurs in our work, Goon et al[33] analyze the code clone ratios throughout the entire development lifetime of open-source projects in order to comprehend code clone growth in software as regards development and potential developer habits which could affect the growth.

Despite all the aforementioned efforts made in these research lines, there is almost no research concerning the prevention of code cloning. This is because both developers and researchers believe that code cloning is inevitable as a direct consequence of human developer faults.[34] This is a consequence of copy-paste (because it is easier than generating code manually), in addition to the ignorance regarding the existence of similar source code pieces in huge unmanageable information systems that could be reused.

In order to prevent or limit code cloning, the first step should be to determine factors that affect code cloning in a development project. Our study investigates the code cloning relationship of some hypothesized factors concerning development teams and the nature of delivery by concentrating on the releasing history of various open source projects.

With regard to our study and how it relates to the four research lines, in the case of clone effects (first line), this study does not investigate side effects as regards considering cloning to be a root cause, but rather studies factors that contribute to higher cloning. In the case of the second and third research lines, this study does not attempt to provide innovative methods with which to detect or deal with code clones, but rather focuses on the prediction and prevention of code cloning (fourth line). Although the relationships among several factors concerning cloning have been widely studied in order to predict and avoid cloning, the novelty of this paper lies in the analysis of factors related to the nature of development projects, such as the number of developers and their participation, the average size of commits, and the total size of the system.

3 | RESEARCH METHOD

This research has been conducted following a mixed method, which combines quantitative and qualitative research methods. Mixed methods *"can help develop rich insights into various phenomena of interest that cannot be fully understood using only a quantitative or a qualitative method"*.[35] In mixed methods research, researchers employ both quantitative and qualitative data because they attempt to provide the best understanding of a research problem.[36] Specifically, our research method considers one quantitative method and one qualitative approach:

- **Quantitative research method**: It consists of a multi-case study with several analysis units extracted from the analysis of six systems throughout all their releasing history. Although many authors consider case studies as a qualitative research method, we focus on the quantitative analysis of the data collected.
- **Qualitative research method**: It is a qualitative description of the phenomena surrounding some results previously obtained in the case study. It can be understood as grounded theory method,[37] in which researchers attempt to derive a general, abstract theory of a process, action, or interaction grounded in the views of cases under study. The process of qualitative research is largely inductive while the quantitative is more deductive.

The design of mixed research methods can be categorized in four major types[36]: triangulation, embedded, explanatory, and exploratory. In this research, we follow the explanatory approach since we use qualitative data to help explain or elaborate quantitative results.

The quantitative research is disseminated in Sections 4 and 6, presenting the design and planning of the case study, and then the quantitative data analysis. The qualitative research is then presented in Section 7.

4 | CASE STUDY DESIGN AND PLANNING

This section presents a detailed case study of six OSSs to which the research introduced is applied. The case study was conducted according to the method with which to design, conduct, and report case studies proposed by Runeson et al.[14] The following subsections show the case study design and planning details, according to the items proposed in the aforementioned method: rationale and objective of the study, cases and units of analysis, theoretical framework, research questions, propositions and hypotheses, concepts and measures, data collection methods, data analysis methods, case selection, selection of data, case study protocol and data storage, quality control and assurance, and ethical considerations. In Section 6 we then go on to address the data analysis and interpretation, along with evaluating the validity of this work.

4.1 | Rationale and objective of the study

The rationale of the study is the limited published research concerning the development context, such as committing, releasing, and the evolution of the size of open source projects, which may affect the evolution of code cloning. The primary motivation for the study was particularly, from a practitioners' point of view, not to describe and create a theory or to provide an in-depth understanding of problems. The main rationale originated, rather, from its projected utility to predict and prevent code cloning in software development projects.

Keeping the aforementioned rationale for the case study in mind, the objective of the study is to determine how code cloning fluctuation in different releases in open source development projects is affected by some specific properties in the project context. The long-term expectation of the research is to improve the prediction and, therefore, the prevention of cloning. This objective is refined into a set of research questions, which are answered by means of the collection and analysis of the case study data presented in the following sections.

4.2 | Cases and units of analysis

The study was designed as a holistic multi-case study[38] because it focuses on six open source development projects. It then analyzes the correlation among the factors in all the different releases in each case. The analysis unit and, therefore, the independent variable is each code base on each different release branch. A release branch is simply a branch in the version control system, on which the code destined for this release can be isolated from mainline development.[39] The study consequently first considers the released version of the software product used to measure necessary code metrics (eg, LoC, lines of code) and second the respective release branch in order to obtain the version control system metrics (eg, number of commits).

4.3 | Theoretical framework

The theoretical framework of the study consists of the related work presented in Section 2, which shows other research analyzing the effects of cloning, detection, refactoring, and handling, along with the evolution of cloning during the system lifecycle so as to predict and prevent it. The limited theoretical development in the area of predicting and preventing code cloning signifies that it is difficult to apply theoretical generalization. However, the earlier studies influenced the design of this study.

4.4 | Research questions

RQ1. How does the development effort in project releasing affect the evolution of code cloning?

RQ2. How does the size of commits in project releasing affect the evolution of code cloning?

RQ3. How does the size of the system affect the evolution of code cloning?

The study defines three research questions: RQ1 to RQ3. RQ1 and RQ2 are based on the main hypothesis that the nature of commits can affect code cloning. Our initial theory is that the more commits a release branch has, the more code cloning is introduced. The other research hypothesis concerns RQ3, which attempts to explain the relationship between system size and the code cloning generated.

4.5 | Propositions and hypotheses

All the research questions defined concern the evolution of certain related measures during the project releasing lifecycle, and how these factors may have an impact on the fluctuation of code cloning. The hypotheses formulated as regards the research questions, therefore, consist of three pairs of null and alternative hypotheses.

$\mathbf{H}_{\mathbf{0}_{RQ1}}$: There is no significant difference in the cloning ratio for different numbers of commits and committers.

$\mathbf{H}_{\mathbf{1}_{RQ1}}$: $\neg H_{0_{RQ1}}$

With regard to the first pair for RQ1, the proposition is that a higher number of commits and committers in a release branch produces a greater cloning ratio. This proposition is based on the idea that the more developers there are contributing to the same release branch, the less communication there is between team members, thus making reuse difficult and leading to more code clones.[13] Although Harder's study analyzes how many developers are involved in the creation and maintenance of clones, RQ1 in our study extends this analysis to the numbers of commits and committers, which are two factors that do not necessarily have a linear relationship.

$\mathbf{H}_{\mathbf{0}_{RQ2}}$: There is no significant difference in the cloning ratio for different sizes of commits (ratio of additions and deletions per commit).

$\mathbf{H}_{\mathbf{1}_{RQ2}}$: $\neg H_{0_{RQ2}}$

In the second pair of hypotheses, for RQ2, the proposition is that the greater the average number of commits, the higher the number of clones produced. In a similar way to that which occurs in the previous proposition, code reuse becomes difficult because a higher number of conflicts may appear, signifying that the merging solutions conducted in the control version system are more difficult.

$\mathbf{H}_{\mathbf{0}_{RQ3}}$: There is no significant difference in the cloning ratio for larger code bases.

$\mathbf{H}_{\mathbf{1}_{RQ3}}$: $\neg H_{0_{RQ3}}$

The last pair of hypotheses, for RQ3, assumes that the more lines of codes, the greater the code cloning derived. The meaning of this proposition is that larger systems make it difficult to handle clones, hence more clones are produced.

4.6 | Concepts and measures

In order to answer the research questions, and keeping the aforementioned hypotheses in mind, several measures are considered, which are classified in five concepts. Table 1 summarizes all the variables by indicating for each: (1) the research question in which it is used (RQ1 to RQ3); (2) the concept to which the variable belongs (releasing, cloning, development effort, commit size, or system size); (3) the name of the variable; (4) whether the variable is dependent or independent; (5) the scale type (ie, interval, ratio, nominal, or ordinal) and, finally, (6) the range definition for all the possible values of the variable.

- **Releasing**. This concept refers to the storyline version of an open source development project.
  - *Release branch* is the independent variable representing the unit of analysis in the embedded case study. This represents the open source project status when a concrete version was released.
- **Cloning**. This concept refers to the code clones detected in the codebase of each release branch.
  - *Cloning Ratio(10)* is the proportion of lines of source code belonging to one of the clones detected (with at least 10 statements) as regards the total number of lines of source code in the systems that were analyzed.
  - *Cloning Ratio (20)* is the same measure, but considering a minimum of 20 statements during clone detection. It is, therefore, always the case that Clone Ratio (20) $\leq$ Clone Ratio (10).

TABLE 1 Concept and measure definitions

RQ	Concept	Measure	Type	Scale Type	Range or Definition
RQ*	Releasing	Release branch	Independent	Nominal	Release branch number X.Y.Z different for every system under study
	Cloning	Cloning ratio (10)	Dependent	Ratio	$x \in \mathbb{R}$, $x \in [0, 100]$
		Cloning ratio (20)	Dependent	Ratio	$x \in \mathbb{R}$, $x \in [0, 100]$
		Cloning ratio growth (10)	Dependent	Interval	$x \in \mathbb{R}$
		Cloning ratio growth (20)	Dependent	Interval	$x \in \mathbb{R}$
RQ1	Development effort	#commits	Dependent	Interval	$x \in \mathbb{N}$
		#committers	Dependent	Interval	$x \in \mathbb{N}$
RQ2	Commit size	Commit size ratio	Dependent	Interval	$x \in \mathbb{R}$
		Commit size category	Dependent	Ordinal	1 = small, 2 = medium, 3 = large, 4 = very-large
RQ3	System size	LoC	Dependent	Interval	$x \in \mathbb{N}$
		Analyzed LoC	Dependent	Interval	$x \in \mathbb{N}$
		Analyzed LoC growth	Dependent	Interval	$x \in \mathbb{R}$

- *Cloning Ratio Growth (10)*. Similarly to the cloning ratio, the study uses the difference between the cloning ratio and the previous release branch. This metric is used to evaluate the percentage of cloning growth on a concrete release branch and is, therefore, computed as the difference in the current cloning ratio minus the previous cloning ratio and is then divided by the previous cloning ratio. This considers this metric for cloning types with a minimum of 10 statements.
- *Cloning Ratio Growth (20)* measures the same concept as the previous measure, but by considering a minimum of 20 statements during clone detection.

- **Development effort**. This concept attempts to measure the amount of changes in a concrete released version
  - *#Commits* is the total number of source code changes registered in the version control system (Git, in this study) for a certain release branch.
  - *#Committers* is the total number of different developers who committed something on a certain release branch.
- **Commit size**. This concept refers to the size and complexity of the changes made to code by developers on a certain release branch.
  - *Commit Size Ratio* is computed as the number of additions or deletions divided by the total number of commits on a certain release branch.
  - *Commit Size Category* is computed on the basis of the quartiles of the previous variable. This is an ordinal variable that can take four values (*small*, *medium*, *large*, and *very large*).
- **System size**. This concept refers to the size of the system that is released on every repository branch.
  - *LoC* is the total number of lines of source code in the version released from the respective release branch.
  - *Analyzed LoC* is the total number of lines of source code that were considered during the clone detection phase. This variable is considered since some lines of source code may be ignored intentionally because, for example, they can be generated automatically and may, therefore, generate many clones. The cloning ratio is computed by considering only the number of lines of source code analyzed.
  - *Analyzed LoC growth* considers the increased percentage of LoC regarding the previous release.

4.7 | Data collection methods

In the case study, the data was primarily collected from two alternative data sources. The first data source consisted of the code base for all the projects that were retrieved from GitHub (an online version control system) together with all the information originating from the version control repository (eg, numbers of commits, committers, etc.). The second data source was the information regarding code clones, which was generated after applying a clone detection technique to the code base for each project. Both data sources were then integrated into a common case study database to be consumed during the data analysis phase.

4.8 | Data analysis methods

This case study combined both qualitative and quantitative data analyses. The technique employed in the qualitative analysis was *explanation building*, as depicted by Yin.[38] This technique identifies patterns based on cause-effect relationships and pursues underlying explanations. In this study, this technique made it possible to identify and explain a relationship between the tendency of the factors being analyzed and the cloning fluctuation throughout the releasing history. This qualitative analysis technique is based principally on descriptive statistics and some explanatory charts that relate different variables.

The quantitative analysis was then used in combination with that described above to confirm preliminary insights obtained by means of the qualitative analysis. The quantitative analysis technique used was the statistical correlation tests and, in particular, the bivariate Pearson correlation test. In statistics, the Pearson correlation coefficient, also referred to as Pearson's r or the bivariate correlation, is a measure of the linear correlation between two variables. This correlation factor has a value of between 1 and −1, where 1 is a total positive linear correlation, 0 is a nonlinear correlation, and −1 is a total negative (inverse) linear correlation. The correlation results made it possible to determine, with a certain statistical significance, whether a concrete factor affected code cloning.

Although the Pearson test is able to provide statistical evidence about correlations, it fails to provide information about the magnitude of the effect of one variable regarding another. The analysis of variance (ANOVA) test was, therefore, also used to check the differences between means in a cloning ratio. The ANOVA was used in conjunction with Tukey's HSD (honest significant difference) test in order to compare all possible pairs of means. The Tukey test is used in together with an ANOVA (post-hoc analysis) to find means that are significantly different from each other.

4.9 | Case selection

After designing the case study, it was necessary to select the cases that would be studied. Table 2 shows a summary of the criteria (C1 to C4) used to select the most appropriate cases. C1 ensured that the project consisted of the development of an OSS which could be freely analyzed and in which different programmers contributed to the code base. C2 was explicitly defined to guarantee that the codebase was tracked through the use of a control version system (specifically, Git) and that the study would, therefore, be capable of attaining information on commits and committers.

TABLE 2 Criteria for case selection

Id	Criterion for Case Selection
C1	It should be an open source system
C2	It must be available in GitHub
C3	It must have at least five release branches
C4	It must not be smaller than 50 KLoC

C3 was established to obtain only those projects that followed the release branch strategy to track the evolution of the code base and release different versions. Release branches allow maintenance to continue in parallel for examples 1.0.x and 1.1.x, and releases can be made independently from both lines (while new development work takes place either directly in the main trunk [master branch] or on short-lived feature branches that become merged into the main trunk as soon as they are ready).[39] This criterion, therefore, established that a minimum number of five release branches would be necessary to evaluate the aforementioned metrics during the evolution of the branches/releases. Finally, C4 was established so as to discard small systems, since it would not be possible to scale the results obtained results to larger systems, and the result could, therefore, have been negligible. The limit was established as 50 000 lines of code, and the last release branch was considered in order to compute this metric.

4.10 | Selection of data

After applying the aforementioned selection criteria, six open source projects from GitHub were selected. Table 3 shows the name, URL under https://github.com/, the date of the first release, thousands of lines of source code, and the number of release branches (corresponding with the number of released versions):

- **Spring Boot** (S1) makes it easy to create stand-alone, production-grade Spring-based applications that developers can run with very little Spring configuration. Spring is an application framework and the inversion of a control container for the Java platform, thus allowing Spring boot to create stand-alone Spring applications; it directly embeds Tomcat, Jetty, or Undertow, and there is, therefore, no need to deploy WAR files; it provides opinionated "starter" POMs with which to simplify Maven configurations; among other features.
- **Cassandra** (S2) is a free and open-source distributed NoSQL database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. Cassandra was initially developed by Facebook to power the Facebook inbox search feature and later became an Apache project.
- **Flink** (S3) is a stream processing framework developed by the Apache Software Foundation. The core of Apache Flink is a distributed streaming dataflow engine written in Java and Scala. Flink provides a high-throughput, low-latency streaming engine in addition to support for event-time processing and state management. Flink does not, however, provide its own data storage system, although it provides data source and sink connectors for systems such as Hadoop, Apache Kafka, HDFS, Apache Cassandra, and Amazon Kinesis, among others.
- **Groovy** (S4) is an object-oriented programming language for the Java platform. It is a dynamic language with features such as those of Python, Ruby, Perl, and Smalltalk. Most valid Java files are also valid Groovy files. Although the two languages are similar, Groovy code can be more compact, because it does not need all the elements required by Java. Groovy can be used as both a programming language and a scripting language for the Java Platform, is compiled for Java virtual machine bytecode and interoperates seamlessly with other Java code and libraries.
- **Tika** (S5) is a toolkit that detects and extracts metadata and text from over a thousand different file types (such as PPT, XLS, and PDF). These file types can be parsed through the use of a single interface, thus making Tika useful for search engine indexing, content analysis or translation, among others. The project originated to provide content identification and extraction when crawling, although Tika later was separated in order to make it more extensible and usable by content management systems, other Web crawlers, and information retrieval systems.
- **Apache Pig** (S6) is a platform whose purpose is to analyze large data sets that consist of a high-level language with which to express data analysis programs that run on Apache Hadoop. Hadoop is, in turn, a collection of open-source software utilities that facilitate the use of a

TABLE 3 Cases selected

Id	Project	URL	Since	KLoC	# Release Branches
S1	Spring boot	Spring-projects/spring-boot	Dec 2013	347	7
S2	Cassandra	Apache/cassandra	Feb 2011	564	9
S3	Flink	Apache/flink	Jan 2014	889	11
S4	Groovy	Apache/groovy	Apr 2009	297	11
S5	Tika	Apache/tika	May 2010	92	15
S6	Pig	Apache/pig	Oct 2010	396	17

network comprising many computers to solve problems involving massive amounts of data and computation. Pig abstracts the programming from the Java Hadoop idiom into a notation that makes Hadoop programming high level, similar to that of SQL for relational database management systems. Apache Pig was originally developed at Yahoo Research to provide an ad-hoc means of creating and executing Hadoop jobs in very large data sets. It subsequently became an Apache project.

4.11 | Case study protocol and data storage

After designing the case study and selecting the cases, the study was carried out following the steps depicted in Figure 1.

The procedure first focused on analyzing the Git repository (see left branch at the beginning of the process depicted in Figure 1). The Git repository was, therefore, cloned to enable it to be imported locally, and some Git commands were then executed using the Git bash tool.

- **Git shortlog**: this summarizes git log output in a format that is suitable for inclusion in release announcements. Each commit is grouped by author and title. Option *-n* (numbered) obtains the sort output according to the number of commits per author rather than the alphabetical order of the authors. Option *-s* (summary) suppresses the commit description and provides a commit count summary only. *git shortlog -sn branch/1.0x* is first executed using this command, followed by *git shortlog -sn branch/1.1.x ^branch/1.0.x*, etc., until every branch has been dealt with. After performing this command, the number of commits and committers per every release branch are obtained.
- **Git diff**: this shows changes between commits, commit and working tree, etc. The *–shortstat* option provides only the last line in the *--stat* format containing the total number of modified files, along with the number of lines added and deleted. *Git diff --shortstat branch/1.0x* is first executed using this command, followed by *git diff --shortstat branch/1.1.x ^branch/1.0.x*, etc., until every branch has been dealt with.

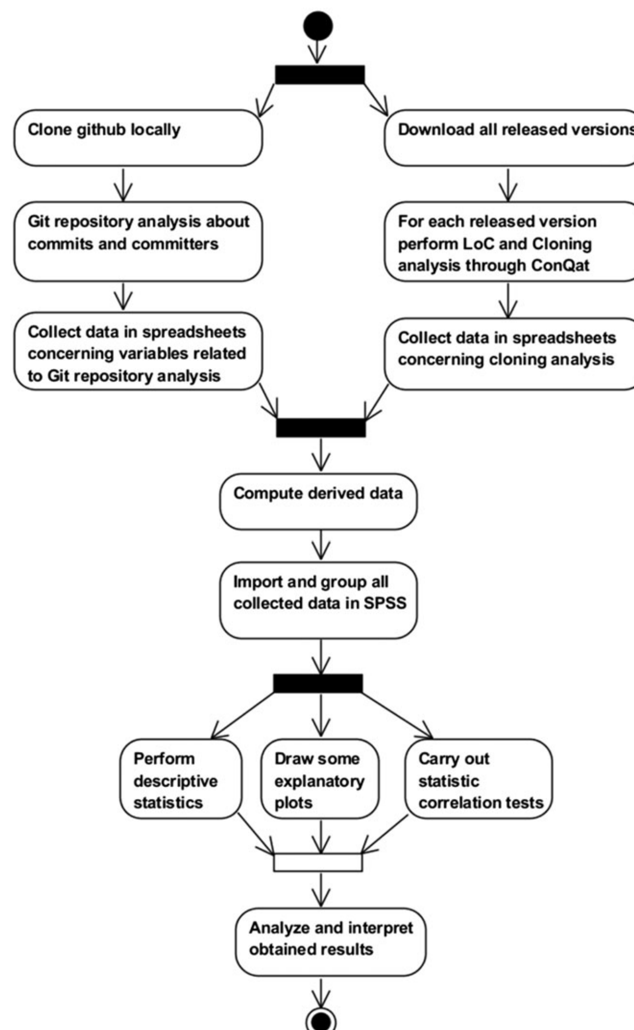

FIGURE 1 Case study procedure

After executing the aforementioned commands for each release branch, the textual outputs obtained are dealt with, and all the relevant data (see definition of variables in subsection 4.6) is added to spreadsheets for future analysis.

The code base is studied and cloning data are extracted in parallel to analyzing the Git repository (see right-hand branch at the beginning of the process depicted in Figure 1). First, all released versions (corresponding with every release branch) are downloaded from GitHub, after which the LoC computing and the code cloning analysis are performed. In this study, the clones were detected by using *ConQAT*,[40] a tool implementing a technique that is able to detect Type 1 to Type 3 clones.[41] The usage of this tool rather than others was because it has good benchmarking recall values for the detection of Type 2 and Type 3 clones.[42]

The cloning analysis can be parametrized with the *minimal number of statements* to consider copies as a clone. For example, a small value like 2 does not make sense, since it means that the copies of two statements are considered as clones. According to many studies, some copies of few lines of code correspond with common programming language constructs[8] or intentionally small clones used to gain code stability.[11] The study presented herein has, therefore, considered thresholds of 10 and 20 so as to take into consideration clones that share more than 10 statements, in addition to those with more than 20 statements. Please note that clones detected by using a parameter value of 20 are also included in clones detected using a parameter value of 10. An example of this is available online* and shows the cloning detection report for release branch *Pig 0.2* (with a minimum number of 10 statements) that was generated with *ConQAT*.

Another possible parameter allowed by *ConQAT* is the exclusion of certain code packages. In this study, packages were excluded through the use of the following patterns: **/test/**, **/integration-tests/**, **/generated/**, **/, and **/gen-java/**. These patterns discard code base focused on testing, along with code automatically generated with compilers and tools. Only the source supporting system functionality and committed by programmers is, therefore, considered.

After the aforementioned analysis had been carried out for every release branch, the generated html reports were checked, and relevant data was collected on spreadsheets (see definition of variables in subsection 4.7).

Since this case study collected a large, complex, diverse body of data, it was important to ensure that this data is stored in a structured manner in order to ensure ease of retrieval, completeness of data sets, and the ability to share data with co-researchers, in addition to supporting the development and maintenance of a chain of evidence. After the Git repository and code cloning had been analyzed, all the data collected on several spreadsheets were consequently merged and additionally derived data (cf. subsection 4.7), such as the ratio of added or deleted lines per commit, were also computed (see Figure 1). All the data that were relevant for the statistical analysis were then collected in SPSS, after which descriptive statistics, explanatory charts, and correlation tests were attained. Having generated all these artifacts, the analysis and interpretation of results were eventually conducted (cf. Section 6). The full experimental dataset can be accessed online at *http://alarcos.esi.uclm.es/per/rpdelcastillo/Ex_Cloning.htm*.

4.12 | Quality control and assurance

We established three methods with which to ensure that the quality of the case study would be considered during all the stages of the study:

- A draft of the case study design was reviewed by peers external to the empirical study.
- A pilot study was conducted in order to evaluate the case study design. This pilot study consisted of the analysis of a small single case which was carried out with Shiro, another open source development project in GitHub. All cloning detections tools and data collection were applied to ensure that all the desired data could be collected.
- During the execution of the case study protocol, the actual progress of the case study was reviewed against the planned progress to determine whether there were any significant differences. More specifically, an exhaustive review was carried out after concluding the data collection and storage steps for each of the six OSSs.

5 | QUANTITATIVE DATA ANALYSIS

After the execution of the case study, the values for the defined metrics were collected for 70 release branches throughout the six open source projects. The data analysis performed to answer each research question is presented in the following sections. In addition to the fact that the most relevant data collected are shown in the following subsection, the experimental data are available in their entirety online,† t hus hopefully enabling the research community to this experimental data for future replications or related empirical studies.

5.1 | RQ1. Commits and committers against cloning

The hypothesis for RQ1 is that both the number of commits and committers are correlated with a high amount of cloning. Figure 2 shows the evolution of cloning for each of the systems studied. Each plot shows the percentage of cloned lines of coder regarding the total number of lines

*https://alarcos.esi.uclm.es/per/rpdelcastillo/ex_cloning/example_cloning/[000007].html

†http://alarcos.esi.uclm.es/per/rpdelcastillo/Ex_Cloning.html

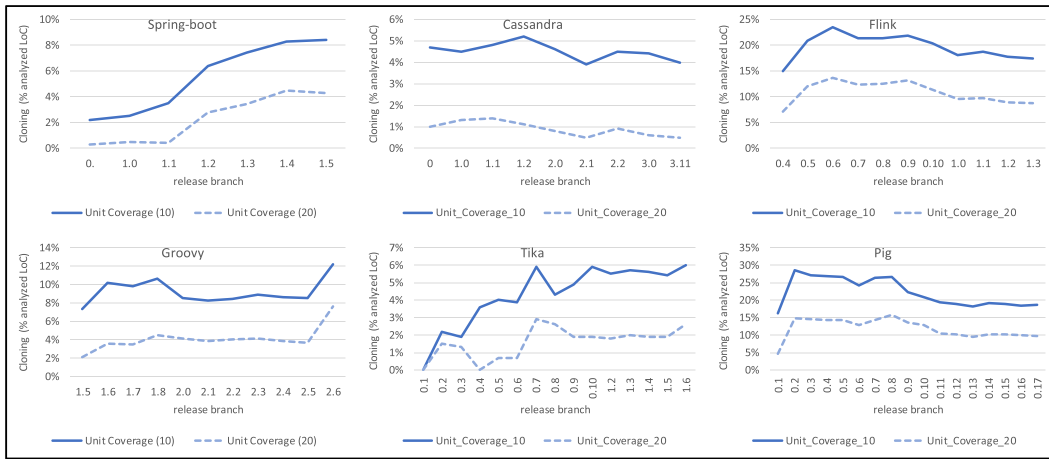

FIGURE 2 Evolution of cloning percentage regarding LoC analyzed

analyzed. Please note that some patterns were used to ignore test and generated code. Moreover, each plot presents two lines representing the clones with at least 10 LoC (continuous lines) and longer clones with at least 20 LoC (dashed lines). An important point is that all the plots have a different scale on their axes. The X-axis represents all the release branches that depend on each system, while the Y-axis represents the percentage of cloning, and the maximum of this scale depends on the maximum cloning ratio attained for each system.

After analyzing Figure 2, the preliminary insight is that all the systems follow a different trend. For example, systems like *Flink* and *Pig* had a higher cloning at the beginning (around 30%) and then followed a downward trend. The reduction in the cloning ratio may be for two main reason: (1) code refactoring, which removes many cloned fragments (ie, the numerator of the ratio is lower than before); or (2) several new lines of code are added to the current release without introducing further clones, signifying that the cloning ratio is reduced (the denominator is higher than before). The first reason is usually related to intentional actions carried out by the development team. This might, for example be because the developers realized the existence of the huge cloning ratio and attempted to revert it in the subsequent releases by means of refactoring actions. The second reason is sometimes related to non-intentional actions, ie, the new lines of source code were added by chance with a lower cloning ratio. This could, however, also be owing to the intentional actions of developers who, for example, agreed to avoid the introduction of further clones in the new parts while no complementary refactoring actions were planned to deal with the existing clones.

Other systems, such as *Spring-boot* and *Tika*, have slightly upward trends and have a low cloning ratio in the first releases (between 0 and 2%). Finally, the lines for *Cassandra* and *Groovy* have peaks but are certainly stable over time (between 5% and 10%). Only *Groovy* has a significant increase in the last release as regards the aforementioned stability.

Having analyzed the cloning trends in each system, the question now is whether one of the factors being studied (number of commits and committers) is the reason for these trends. Figures 3 and 4, respectively present the fluctuations of the commits and committers in each branch release. Note that the release branches do not have to be produced with the same duration. If we compare these figures with the peaks in Figure 2 , some of these peaks match while most of them do not.

The preliminary insight obtained by comparing the plots can be confirmed by means of the bivariate Pearson correlation test. Table 4 shows the results of the correlation tests for each system, in addition to considering the release branch entries for all the systems together. The highlighted cells represent cases in which a correlation exists. The lightly highlighted cells represent a confidence level of 95% whilst those that

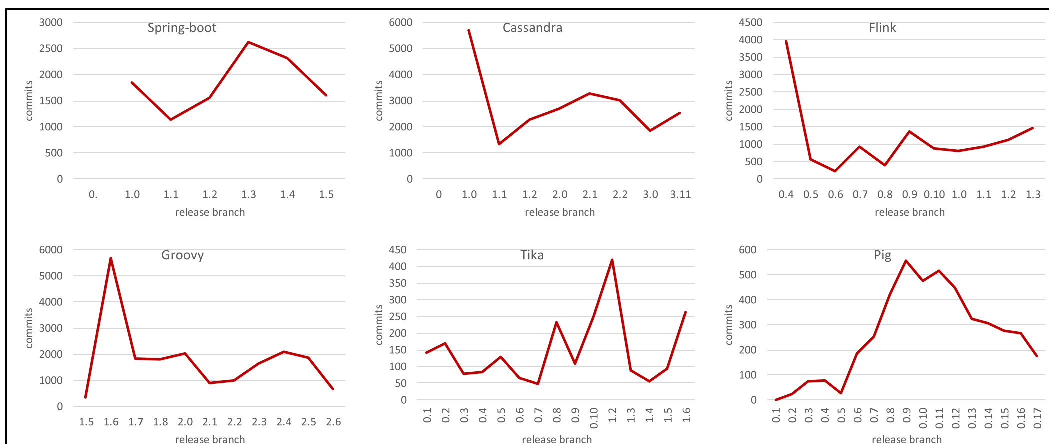

FIGURE 3 Evolution of number of commits

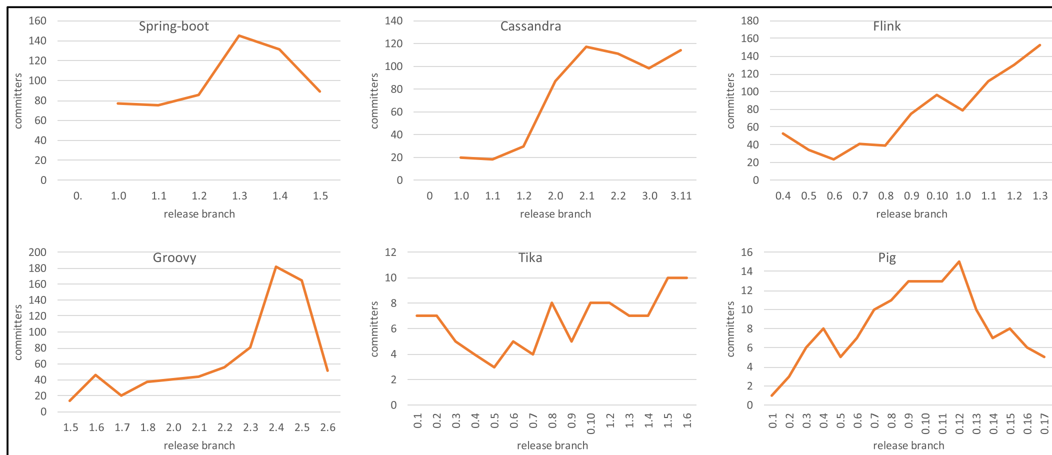

FIGURE 4 Evolution of number of committers

TABLE 4 Pearson correlation results for committers and commits when compared with cloning ratio

	Committers				Commits			
System	Cloning R. (10)	*P*-value	Cloning R. (20)	*P*-value	Cloning R. (10)	*P*-value	Cloning R. (20)	*P*-value
Spring-boot	0.653	0.160	0.653	0.160	0.471	0.345	0.526	0.345
Cassandra	−0.738	0.037	−0.922	0.001	−0.228	0.588	0.165	0.697
Flink	−0.849	0.002	−0.848	0.002	−0.585	0.076	−0.525	0.119
Groovy	−0.380	0.279	−0.160	0.659	0.083	0.819	−0.396	0.257
Tika	0.419	0.135	0.534	0.049	0.233	0.423	0.255	0.378
Pig	−0.342	0.195	−0.136	0.616	−0.557	0.025	−0.322	0.223
ALL	−0.187	0.138	−0.120	0.120	−0.348	0.005	−0.379	0.002

are heavily highlighted indicate a confidence level of 99%. These results indicate that only some of the systems (*Cassandra*, *Flink*, and *Tika*) have any kind of correlation between committers and the cloning ratio. Even more, most of these correlations are negative, signifying that there are inverse relationships between both variables. Only the cloning ratio (20) presents a positive correlation. With regard to commits, only *Pig* has a correlation regarding the cloning ratio in addition to whether all the analysis units are considered together. Anyway, these correlation factors are negative as well (see Table 4). Since most of these correlations are negative, it can be stated that the more committers or commits are involved, the lower the cloning ratio is. Suffice it to say that most of the comparisons do not have sufficient statistical significance. Nevertheless, there are both positive and negative Pearson correlation coefficients and with a wide variety of values between −1 and 1.

Upon considering these results, $H_{0_{RQ1}}$ cannot be rejected, which determines that the number of commits or committers have no effect on the cloning fluctuations during system evolution. The main implication of this research question is aligned with the insights of Goon et al[33]: every project most likely follows its own trends (see Figure 2). On the one hand, the trend noticed by Goon et al,[33] in which projects have an initial period of instability and fluctuations followed by a period of stability, is confirmed. Furthermore, there are other trends in which cloning is under control (with a minimal fluctuation) but grows slightly over time. These overall trends suggest some general workflow of code clone management.

As occurred with the results provided by Harder,[13] this study finds differences in cloning ratios for different numbers of commits and committers, although a clear positive linear correlation cannot be established.

5.2 | RQ2. Size of commits against cloning

The hypothesis for RQ2 is that there is a correlation between the size of the commits and the cloning ratio. The size of the commits is measured as the additions and deletions per commit for a concrete release branch. Figure 5 shows the evolution of the commit size ratio per system on each release branch. On the one hand, some of the peaks in the fluctuation of these plots match with the peaks in the cloning evolution plots shown in Figure 2. On the other, upon comparing the commit size and the number of commits evolution (see Figures 3 and 5), it will be observed that neither factor is directly correlated. This makes sense if we consider that the size of the commits depends on the development context. For example, in the case of the *Pig* system, it will be noted that there are fewer commits on the first release branches, although the size of these commits is significantly higher than in the subsequent releases. This could be explained by the fact that there were fewer committers at the beginning, and the development context basically consisted of the additions of new functionalities from scratch, and the commits were, therefore, made from time to time without undergoing many merging conflicts. In conclusion, the analysis of the number of commits and the commit size as two independent factors that can affect the cloning ratio in a different way would appear to be appropriate.

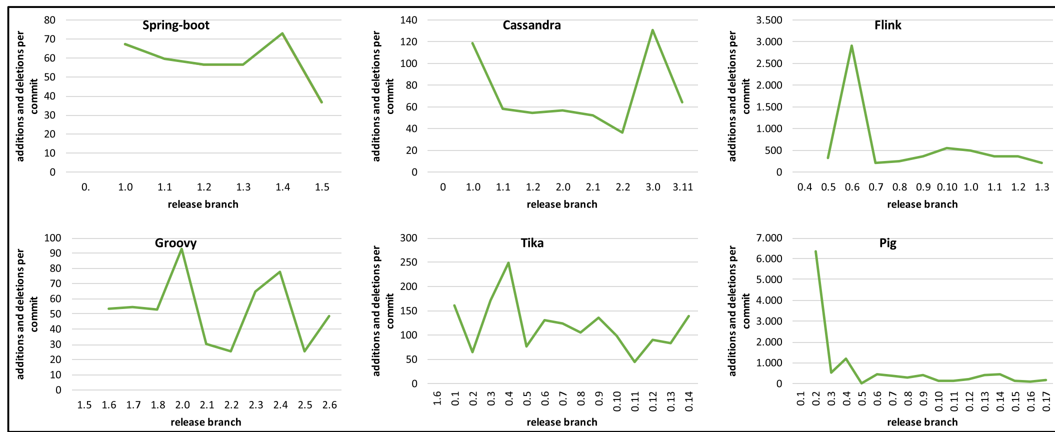

FIGURE 5 Evolution of commit size ratio

The hypothesized correlation between commit size and cloning ratio had to be confirmed by means of the bivariate Pearson correlation test. Table 5 shows the results for the bivariate Pearson correlation tests for commit size and cloning ratio and cloning growth. The results obtained for the data segmented by system are not significant (with the exception of the *Pig* and *Tika* system, see highlighted cells). However, the correlation test carried out by considering all analysis subunits for all systems (without data segmentation) provided a positive correlation for the four variables analyzed with a confidence level of 99%. The growth of cloning is particularly highly correlated with the size of commits with 0.630 (10) and 0.661 (20).

Upon considering these results, $H_{0_{RQ2}}$ can be rejected, and it is determined that the size of commits affects the cloning fluctuations during system evolution.

Despite the results obtained through the use of the Pearson correlation test, this test failed to provide statistical evidence regarding what the magnitude of this effect is. An ANOVA test was, therefore, conducted to check the differences between the means for the cloning ratio. This was done using the four categories (low, medium, large, very large) for the size of the commit variable, which were computed on the basis of quartiles. This uses these categories due to the ANOVA test requires an ordinal (category-based) variable to be used as "factor." For this reason, a new variable was created by considering these four categories by considering quartiles of the whole distribution of commit sizes. Thus, the commit size categories can be understood as a mean for knowing the commit size in a relative way for all the releases. Thus, the categories computed in this study correspond to the following numbers of lines of code.

- Small: LoC < 58
- Medium: 58 ≤ LoC < 124
- Large: 124 ≤ LoC < 250
- Very large: LoC ≥ 250

Table 6 presents the results for the ANOVA test, which shows significant effects for both cloning ratio (10) and cloning ratio (20). These results are aligned with those obtained in the Pearson test. However, the ANOVA test provides the magnitude of this effect (column R^2 in Table 6), signifying that at least 57.8% (10) and 57.1% (20) of the cloning ratio values can be explained by the size of the commits. Furthermore, it was possible to discover the differences in cloning ratio for each commit size category by using Tukey's post hoc test (see Table 7). This test showed significance for almost every pair of sizes (see highlighted cells). However, the most recurrent difference in these comparisons was the "very-large" size regarding each other. These results can also be graphically checked with the box plot presented in Figure 6.

TABLE 5 Pearson correlation results for commit size regarding cloning ratio

	Cloning R. (10)	*P*-Value	Cloning R. (20)	*P*-Value	Cloning R. Growth (10)	*P*-Value	Cloning R. Growth (20)	*P*-Value
Spring-boot	−0.361	0.483	−0.290	0.577	0.135	0.799	0.277	0.595
Cassandra	−0.080	0.851	0.038	0.929	−0.192	0.648	−0.043	0.920
Flink	0.553	0.097	0.443	0.200	0.291	0.415	0.206	0.567
Groovy	0.019	0.958	−0.010	0.977	−0.280	0.434	−0.100	0.783
Tika	−0.163	0.579	−0.296	0.304	0.538	0.047	0.170	0.562
Pig	0.476	0.063	0.353	0.180	0.892	0.000	0.916	0.000
ALL	0.431	0.000	0.402	0.001	0.630	0.000	0.661	0.000

TABLE 6 ANOVA test results for cloning ratio considering size of commit category as factor

	Types III Sum of Squares	R^2	F (ANOVA)	*P*-Value
Cloning ratio (10)	0.247	0.578	27.414	0.000
Cloning ratio (20)	0.092	0.571	26.583	0.000

TABLE 7 Tukey's port hoc test for cloning ratio considering size of commit category

Cloning	Size Commit Category		Mean Difference	Standard Error	*P*-Value
Cloning ratio (10)	Small	Medium	0.026	0.019	0.540
		Large	−0.033	0.019	0.337
		Very large	−0.137	0.019	0.000
	Medium	Small	−0.026	0.019	0.540
		Large	−0.059	0.019	0.018
		Very large	−0.163	0.019	0.000
	Large	Small	0.033	0.019	0.337
		Medium	0.059	0.019	0.018
		Very large	−0.105	0.019	0.000
	Very large	Small	0.137	0.019	0.000
		Medium	0.163	0.019	0.000
		Large	0.105	0.019	0.000
Cloning ratio (20)	Small	Medium	0.013	0.012	0.706
		Large	−0.024	0.012	0.194
		Very large	−0.086	0.012	0.000
	Medium	Small	−0.013	0.012	0.706
		Large	−0.037	0.012	0.016
		Very large	−0.099	0.012	0.000
	Large	Small	0.024	0.012	0.194
		Medium	0.037	0.012	0.016
		Very large	−0.062	0.012	0.000
	Very large	Small	0.086	0.012	0.000
		Medium	0.099	0.012	0.000
		Large	0.062	0.012	0.000

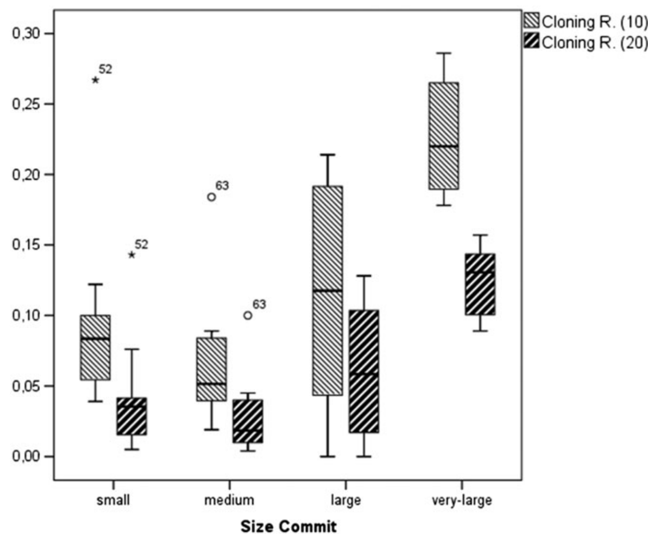

FIGURE 6 Box plot for cloning ratio and commit size categories

5.3 | RQ3. Size of the system against cloning

The hypothesis for RQ3 is that there is a correlation between the size of the system and the cloning ratio. Figure 7 shows the evolution of the number of LoC, along with the number of LoC analyzed after ignoring testing and generated code. It can be observed that there is a growing upward trend. This evolution is to be expected, since new functionality is added in each release. However, sometimes there are some stabilization releases in which LoC is the almost the same, or even decreases, as occurs in release 0.13 of *Pig*. A common explanation for this is that certain releases focus on refactoring and improving some quality aspects rather than simply adding new functionality.

The statistical correlation analysis for this research question focuses first on how the LoC analyzed are correlated with the cloning ratio (ie, absolute values), and second on the growth of the LoC analyzed in comparison to the growth of the cloning ratio on each release branch. Table 8 shows the bivariate Pearson correlation results for these cases. Upon considering absolute values (see left-hand side of Table 8), it will be observed that there are correlations in most of the systems with a confidence level of 99% (see highlighted cells). Surprisingly, there are both positive and negative correlations. For example, *Spring-boot* and *Tika* (for cloning ratio (10)) have strong positive correlations, ie, 0.962 and 0.789, respectively.

Flink, *Pig*, and *Cassandra* (only in the case of larger clones) simultaneously have a significant negative correlation factor. If the growth values are analyzed (see right-hand side of Table 8), then all correlation values are positive, with the exception of *Cassandra*. However, most of the results that consider growth variables are not statistically significant.

Upon taking into account the statistically significant results, $H_{0_{RQ3}}$ can be rejected, and it can be stated that size has some effect on the cloning evolution. Unfortunately, the direction of the correlation cannot be determined without further empirical validations.

The results obtained can be explained by the existence of different development contexts. The first possible context is that some release iterations are devoted to refactoring or improving certain quality issues in code rather than adding new functionality. The reduction in the cloning ratio (after it was detected in previous releases) might, therefore, have been defined as one of the goals in a concrete release. As a result, the cloning ratio decreases regardless of the LoC. For example, *Flink* and *Pig* have growing downward trends after huge cloning ratios in the first releases (see Figure 2), signifying that the negative correlation makes sense.

Another alternative development context that may explain the obtained results consists of projects in which cloning is moderately low and only a slight increase in cloning occurs upon each release (see cloning evolution for *Spring-boot*, *Groovy*, and *Tika* in Figure 2). Consequently, (1)

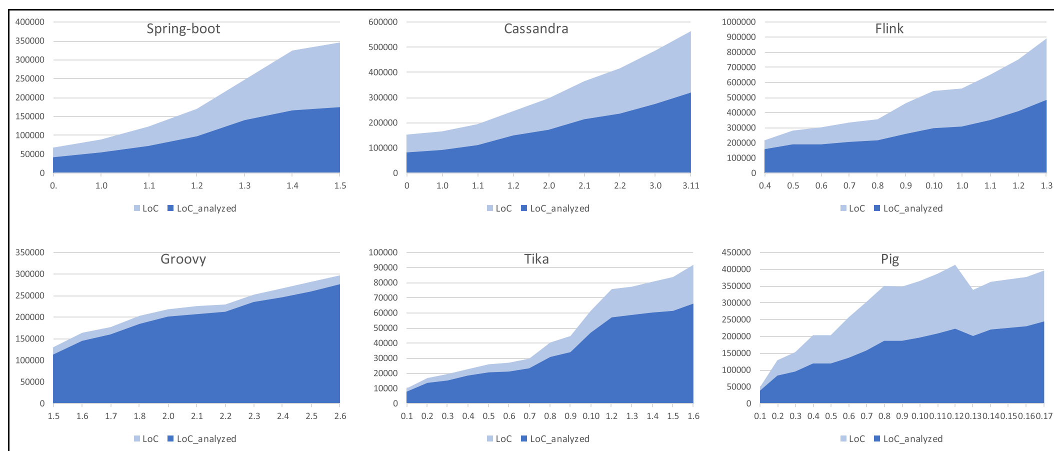

FIGURE 7 Evolution of system size and number of lines of code analyzed

TABLE 8 Pearson correlation results for LoC and cloning ratio analyzed

	LoC Analyzed				LoC Growth Analyzed			
	Cloning ratio (10)	*P*-value	Cloning ratio (20)	*P*-value	Cloning ratio growth (10)	*P*-value	Cloning ratio growth (20)	*P*-value
Spring-boot	0.962	0.002	0.962	0.002	0.502	0.310	0.346	0.501
Cassandra	-0.617	0.103	-0.874	0.005	-0.001	0.998	-0.624	0.098
Flink	-0.869	0.001	-0.877	0.001	0.327	0.356	0.356	0.313
Groovy	0.007	0.985	0.545	0.103	0.429	0.216	0.241	0.503
Tika	0.789	0.001	0.490	0.075	0.218	0.454	0.136	0.643
Pig	-0.903	0.000	-0.792	0.000	0.907	0.000	0.932	0.000
ALL	0.391	0.001	0.377	0.002	0.648	0.000	0.641	0.000

the developers were perhaps not worried about cloning, and it was, therefore, ignored, or (2) the developers were perhaps conscious of the cloning, but its treatment had to be deferred in order to move the project forward (ie, some technical debt was deliberately added). This context, in which cloning evolution has a growing upward trend, could explain the positive correlation factors.

In all events, the mixture of positive and negative correlations is in line with the long-standing debate on whether or not clones are beneficial.[4]

5.4 | Validity evaluation

This section shows the threats to the validity of this case study in order to denote the trustworthiness of the results. According to the scheme presented in Runeson et al,[14] four aspects of the validity and threats to validity can be distinguished: construct validity, internal validity, external validity, and reliability.

5.4.1 | Construct validity

With regard to the *construct validity*, the measures proposed were appropriate as regards measuring the variables and answering the research questions. However, the effect of other factors could be analyzed. For instance, team diversity measured by time zone and the committers' country, developer expertise, and so forth. These diversity factors were analyzed in.[43]

Furthermore, another threat must be mentioned. Clone detection was parametrized with two arbitrary values (10 and 20) for the minimum number of statements of clones. The goal was to analyze different sizes of clones. However, these two arbitrarily fixed values may be rather relative.

One possible explanation in RQ2 (correlation between commit size and cloning) was that larger commits lead to more severe merging conflicts in code repositories, and that this in turn leads to higher cloning. Keeping this in mind, the number of merging conflicts may be taken into account in future studies.

Finally, the study could have considered qualitative aspects related to the evolution of clones across the system releases. This study has not considered what happens to a given clone from one release to the next. The number of equivalent clones that was reduced or eliminated could, therefore, also be measured. In order to mitigate this, Section 7 provides a qualitative analysis on the basis of individual clone evolution through all the different releases.

5.4.2 | Internal validity

There is no large population that makes it possible to obtain statistically representative results. However, a clear trend for the proposed measures was identifiable in this multi-case study with 6 systems and 70 subunits of analysis. In the future, the results of these cases will be contrasted with the results obtained for the same study of other open source projects by means of meta-analysis.

A major threat to the *internal validity* concerns *ConQAT*, the tool used to detect clones. The cloning measurements obtained could be different when using alternative tools. Additionally, Type 4 clones were outside the scope of this case study. In the future, it will be necessary to consider semantic clones, together with the use of other tools. Moreover, the testing and generated code was excluded during the clone detection phase without any insights into how this kind of source code can affect the cloning ratio.

Finally, in order to analyze the evolution and fluctuation of the variables used, they were measured for each release branch. However, alternative means of discretizing these variables could be used. It would, for example, be possible to take measures every fixed period of time, or to consider a continuous measure by taking values for every single commit.

5.4.3 | External validity

External validity concerns the generalization of the results. In this case study, the results obtained could be generalized to open source projects in GitHub using Java and following a release branching strategy. Since all the cases selected were coded in Java, systems written in other programming languages must be considered for a broader generalization, and particularly other programming languages that do not support object orientation. Additionally, in order to achieve a better generalization, it is necessary to analyze systems shared in different repositories as a part of GitHub and even business projects that are not open source.

5.4.4 | Reliability

This aspect is concerned with the extent to which the data and the analysis are dependent on the specific researchers. If the same study is conducted by other researchers, then the results should be the same since we provide a web page showing the full experimental data set (both, raw and derived data), along with the statistical test result. The source code of all the open source projects studied can be accessed through GitHub in order to replicate the study.

6 | QUALITATIVE ANALYSIS AND INTERPRETATION

Apart from the quantitative analysis conducted through the case study, other qualitative aspects can be considered to get a better explanation or reinforce insights previously extracted. In this sense, this qualitative analysis focuses on the evolution of clones across the system releases. Thus, the research question behind this analysis is: *what happen to clones from a release to the next one?*

This qualitative analysis can help to verify and prove some of the hypothesized explanations we provided for some of the results obtained in the case study. This analysis could be also useful to better understand the reasons of the different cloning evolution patterns that were distinguished by means of the previous quantitative analysis.

6.1 | Cloning evolution patterns

The qualitative analysis was designed to consider at least one system for each cloning evolution pattern detected before. Thus, we can distinguish:

- Out of the control: systems that present a lower cloning ratio in first releases (around 5%), and then these experiment upward trends. An example of this pattern is *Spring-boot*.
- Toward control: systems having higher cloning ratio at the beginning (around 30%) and then these present a downward trend. This might have been owing to the fact that the developers realized there was a huge cloning ratio and attempted to reduce it in the subsequent releases. An example of this pattern is *Pig*.
- Under control: systems releases presenting some peaks, but these remain stable over time without exceeding a certain threshold. These systems could have some upward trends in first releases. However, after that, cloning ratio is stabilized around 5%. An example of this pattern is *Tika*.

In summary, the three systems analyzed are Spring-boot, Pig, and Tika. The scope of this analysis is also limited to the track of clones of size 10. Bigger clones (size 20 code units) are less numerous, and many releases do not have clones of such a size.

6.2 | Cloning tracking analysis

It should be realized that the implementation and usage of clone tracking mechanisms to determine equivalent clones and track them are not easy and are treated in specific research streams (Duala-Ekoko et al, 2007,[30]). Although there are some tools that apparently track clones in this way, it is extremely difficult to track actual equivalent clones between releases. A certain clone regarding the previous release could be extended, reduced, preserved, removed, divided, merged, etc. An equivalent definition for each of these operations should has to be carefully defined, otherwise the equivalence function could lead to wrong results. This is the main reason for which tracking clones is not easy.

Other approaches mark equivalent clones at the line of code level. This means that lines of source code are tagged to show whether or not they are part of a clone, and these lines are then tracked throughout the releases. This approach facilitates the clone tracking and is the one followed in this qualitative analysis.

Following the mentioned approach, we figured out the preserved clones between releases, those new clones added, and those removed for each release. On one hand, Figure 8 depicts the evolution of the clones throughout the. On the other hand, Figure 9 summarizes the data collected for each system under the tracking analysis. Although Figures 8 and 2 may seem quite similar, Figure 8 shows the evolution of the absolute number of clones in every release, while Figure 2 shows the cloning ratio that depends not only on the number of clones but also depends on length of clones and the length of the whole system (ie, it is a relative measure). Figure 8 shows the number of clones discarded (removed intentionally or not intentionally) as well as the new clones introduced in every release. These bar plots confirm the cloning evolution patterns depicted

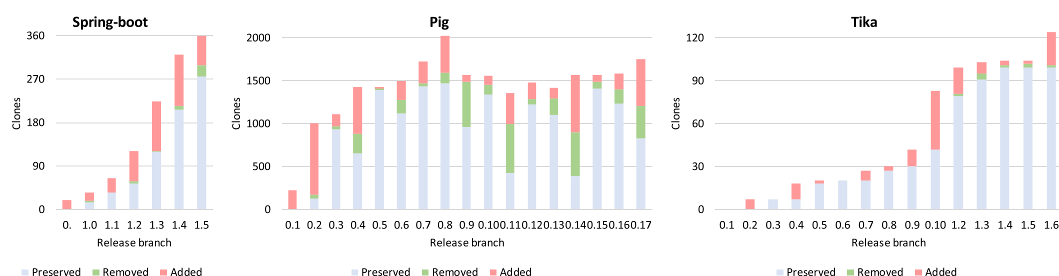

FIGURE 8 Evolution of the preserved, removed, and added clones in Spring-boot, Pig, and Tika

System	Release	# Clones Total	# Clones Added	# Clones Removed	# Clones Preserved	Commiters	Commits	System Size	Commit Size	Pattern
Sprint-boot (out of the control)	0.	20	20	0	0	0	0	0%	0	
	1.0	35	◆ 18	◆ 3	◆ 17	77	1853	▲ 30%	● 67	
	1.1	64	◆ 29	◆ 0	◆ 35	75	1129	▲ 31%	▲ 45	
	1.2	121	▲ 62	◆ 5	◆ 59	86	1557	● 36%	▲ 42	A-M
	1.3	224	● 104	◆ 1	▲ 120	145	2626	● 46%	▲ 35	A-M
	1.4	321	● 106	▲ 9	● 215	131	2312	◆ 17%	▲ 41	A-M
	1.5	359	▲ 61	● 23	● 298	89	1601	◆ 6%	◆ 17	R-S
Pig (toward control)	0.1	225	◆ 225	0	0	1	1	0%	0	
	0.2	1008	834	51	174	3	24	120%	4273	
	0.3	1113	◆ 143	◆ 38	◆ 970	6	74	▲ 15%	● 339	
	0.4	1429	● 545	▲ 229	◆ 884	8	79	● 26%	● 1125	A-L
	0.5	1429	◆ 20	◆ 20	● 1409	5	26	◆ 0%	◆ 28	
	0.6	1498	◆ 225	◆ 156	▲ 1273	7	187	▲ 13%	● 316	
	0.7	1726	▲ 260	◆ 32	● 1466	10	253	● 17%	▲ 192	A-M
	0.8	2024	▲ 428	◆ 130	● 1596	11	421	● 19%	▲ 152	A-M
	0.9	1563	◆ 72	● 533	● 1491	13	558	◆ 0%	◆ 57	R-S
	0.10	1556	◆ 106	◆ 113	● 1450	13	476	▲ 5%	◆ 24	
	0.11	1353	▲ 361	● 564	◆ 992	13	516	▲ 6%	◆ 64	R-S
	0.12	1481	◆ 194	◆ 66	▲ 1287	15	449	▲ 7%	◆ 92	
	0.13	1420	◆ 130	◆ 191	▲ 1290	10	325	◆ -9%	▲ 227	
	0.14	1567	● 664	● 517	◆ 903	7	308	▲ 9%	◆ 111	R-S
	0.15	1570	◆ 81	◆ 78	● 1489	8	276	◆ 2%	◆ 71	
	0.16	1582	◆ 181	◆ 169	● 1401	6	267	◆ 2%	◆ 52	
	0.17	1753	● 548	▲ 377	▲ 1205	5	177	▲ 7%	▲ 129	A-M
Tika (under control)	0.1	0	0	0	0	7	141	0%	0	
	0.2	7	◆ 7	◆ 0	◆ 0	7	170	● 75%	● 112	A-L
	0.3	7	◆ 0	◆ 0	◆ 7	5	78	▲ 14%	◆ 50	
	0.4	18	◆ 11	◆ 0	◆ 7	4	84	▲ 21%	◆ 58	A-M
	0.5	20	◆ 2	◆ 0	◆ 18	3	130	◆ 10%	● 193	
	0.6	20	◆ 0	◆ 0	◆ 20	5	66	◆ 2%	◆ 55	
	0.7	27	◆ 7	◆ 0	◆ 20	4	48	◆ 12%	◆ 52	
	0.8	30	◆ 3	◆ 0	◆ 27	8	232	● 31%	◆ 48	
	0.9	42	◆ 12	◆ 0	◆ 30	5	109	◆ 11%	◆ 73	
	0.10	83	● 41	◆ 0	▲ 42	8	251	● 37%	▲ 109	A-M
	1.2	99	▲ 18	▲ 2	● 81	8	419	▲ 22%	◆ 66	A-M
	1.3	103	◆ 8	● 4	● 95	7	88	◆ 2%	◆ 29	R-S
	1.4	104	◆ 3	▲ 2	● 101	7	55	◆ 3%	◆ 61	
	1.5	103	◆ 2	● 3	● 102	10	95	◆ 2%	◆ 48	R-S
	1.6	124	▲ 23	▲ 2	● 101	10	263	◆ 8%	◆ 29	

FIGURE 9 Patterns detected regarding commit size and added/removed clones. (A-L, adding clones with large commit size; R-S, removing clones with small commit size)

above. One important detail to be considered when these patterns are checked is that the volume of clones greatly differs between systems. Thus, there are thousands of clones in *Pig*, while for example *Tika* only presents around one hundred clones.

6.3 | Behavioral patterns regarding commit size

The major finding obtained through the previous quantitative analysis is that size of commits affects the cloning ratio. One possible interpretation of these results is that bigger commits with more additions and deletions introduce more copies of the existing codes. While a change is growing and is not committed, the probability that someone is changing the same parts increases.[44] This is especially probable if the commits become large (see very-large category in Figure 6) As a result, this scenario produces more merging conflicts that can be resolved by introducing further clones in an unintentional manner (the clones supporting the same functionality in the system are unknown) or deliberately (clones are already known, but the main goal is to achieve system stability and avoid refactoring).[45]

In order to confirm this, some patterns have been searched in the qualitative data collected (see Figure 9). First, relevant data were qualitatively denominated with a three-Likert flag (ie, diamond for small values, triangle for medium values, and circle for large values). These flags are statistically and independently calculated for each column and system. After that, certain patterns are searched row by row (ie, for each release)

and annotated into the right column (or empty if no evidence for one of the target patterns). According to our previous findings and assumptions, the patterns searched in this qualitative analysis are:

- **A-L**: System's releases in which a huge number of clones are introduced (in comparison with the whole system releasing history) and the commit size is large.
- **A-M**: Similar to A-M, but the commit size is medium (taking into account all the different commit sizes throughout the releases).
- **R-S**: The number of removing clones is extremely high or at least higher than the number of clones added. Additionally, the size of commits on average for such releases is small.

Figure 9 shows that A-M (9) and A-L (2) were clearly detected 11 times in the 36 releases. These results show a relationship between commit size and the number of new clones introduced in a certain release. Additionally, A-L and A-M occurrences are similarly distributed in all the systems. Hence, it seems not to be associated with none of the aforementioned cloning evolution patterns. This means, it happens for all the systems.

Regarding R-S, it was detected in six releases of those analyzed. Although this number seems to be low, it should be noticed that clones are not removed in all the releases but only in some of them. Actually, the R-S pattern mostly happens in the two expected systems according to the cloning evolution patterns, ie, "toward control" (*Pig*) and "under control" (*Tika*). Therefore, it can be said that smaller commit size is related in some cases with the reduction of clones.

6.4 | Interpretation and limitations

Results obtained in this qualitative analysis help to confirm the obtained quantitate insights. The patterns searched were detected with some exceptions, specifically in the first releases of the systems analyzed. This can be explained by means of the huge variability and randomness in first releases in comparison with later releases when the system development can be considered as stable.

Not only were some behavioral patterns detected, but they also can be associated with the cloning evolution patterns. Thus, addition of clones through medium and large commits happens in all the three evolutionary patterns, while removing clones through smaller commits happens when cloning is wanted to be under or toward control, ie, cloning that is being intentionally reduced.

The main limitations for this qualitative analysis lie in the fact that it is conducted with only three systems with 39 releases in total. Nevertheless, these were selected to consider all the different evolutionary patterns we distinguished.

7 | CONCLUSIONS

Code clones impact on code quality since defects can be propagated through several copies, thus making maintainability and the understandability of similar copies more difficult. In other cases, clones are self-admitted, and the development continues by tracking and handling those copies. These and other related effects are well known thanks to the considerable amount of research carried out as regards the root causes and effects of cloning. Nevertheless, there is not much work on how certain project context factors (although not related at the beginning) can affect the cloning ratio in a project.

This work presented an empirical validation consisting of a multi-case study of six open source projects throughout 70 different releases. The study focuses on the evolution throughout the release history of certain factors (eg, the number of developers and their participation, the average size of commits, and the total size of the system) and how these factors impact on code clones. The conclusions, which were drawn by means of the empirical results obtained, show that two of the factors analyzed do not affect cloning, while the size of commits has a certain impact.

Neither the number of committers and commits nor the system size had any effect on code clones. We observed that each system had its own evolution. These results are, in some respects, aligned with those attained in other related works.[13,33] Although these factors do not have an effect on cloning, the empirical study shows different cloning evolution patterns such as "out of the control," "toward control," or "under control."

With regard to the factors that impact on code clones, the size of commits, which is measured as the average amount of additions and deletions per commit, had a positive correlation relationship with the cloning ratio. The effect of this factor on the cloning ratio was statistically quantified at around 57%. This means that other factors could explain the cloning ratio values, together with the size of the commits. However, we believe that this percentage is sufficient to consider this effect. Our interpretation of this result is that larger commits can introduce more copies of the existing codes. This tried to be confirmed through a qualitative analysis as well. The rationale may be that, while a change is increasing and is not committed, the probability that someone is changing overlapped parts increases, thus leading to merging conflicts whose respective fixing may unintentionally (the clone supporting the same functionality is unknown) or deliberately (with the aim of achieving system stability and deferring refactoring efforts) introduce code clones.

After analyzing these factors, the main implication of this research, which was carried out in order to prevent code cloning, is that developers should commit to small changes as soon as possible rather than locally saving huge changes and committing everything at once later. This result may also help project managers to predict and prevent code cloning in open source projects.

Despite the aforementioned conclusions of the empirical validation, there are certain threats to the validity of this work, signifying that the following future research efforts will be necessary. First, most project will be involved in the same or similar study to achieve stronger conclusions. Projects using different programming languages and paradigms (not only object-oriented) will be considered, along with projects from other repositories or even business projects that are not necessarily open source projects. In the future, the results of these cases will be contrasted with the results obtained in this study by means of meta-analysis. Furthermore, cloning and factor measuring will be improved by considering other factors, such as team diversity (country, time zone, etc.) and the number of merges in the git repository. Measuring cloning in a continuous manner (for example, for every commit) is challenging and would be a powerful means to draw stronger conclusions about the evolution of the factors that may affect cloning.

ACKNOWLEDGEMENTS

This work is part of the projects SEQUOIA (TIN2015-63502-C3-1-R) and GINSENG (TIN2015-70259-C2-1-R) funded by the Spanish Ministry of Economy and Competitiveness and the FEDER Fund (European funds for regional development). This study has also been funded by projects GEMA (SBPLY/17/180501/000293) and “SOS (Software Sustainability)” funded by JCCM. The work was also supported through a grant Dr. Ricardo Pérez-Castillo enjoys from JCCM within the initiatives for talent retention and return in line with RIS3 goals.

ORCID

Ricardo Perez-Castillo http://orcid.org/0000-0002-9271-3184

REFERENCES

1. Abrahao S, Baldassarre MT, Caivano D, Dittrich Y, Lanzilotti R, Piccinno A. *Human Factors in Software Development Processes: Measuring System Quality*. Cham: Springer International Publishing; 2016.
2. Baldassarre, M. T., A. Bianchi, D. Caivano and C. A. Visaggio (2003). Full reuse maintenance process for reducing software degradation. Seventh European Conference on Software Maintenance and Reengineering, 2003. Proceedings
3. Janicijevic I, Krsmanovic M, Zivkovic N, Lazarevic S. Software quality improvement: a model based on managing factors impacting software quality. *Softw Qual J*. 2016;24(2):247-270.
4. Mondal, M., M. S. Rahman, R. K. Saha, C. K. Roy, J. Krinke and K. A. Schneider (2011). An empirical study of the impacts of clones in software maintenance. 2011 IEEE 19th International Conference on Program Comprehension.
5. Selim, G. M. K., L. Barbour, W. Shang, B. Adams, A. E. Hassan and Y. Zou (2010). Studying the impact of clones on software defects. 2010 17th Working Conference on Reverse Engineering.
6. Sheneamer, A. and J. Kalita (2016). Semantic clone detection using machine learning. 2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA).
7. Saini, V., H. Sajnani, J. Kim and C. Lopes (2016). SourcererCC and SourcererCC-I: tools to detect clones in batch mode and during software development. Proceedings of the 38th International Conference on Software Engineering Companion. Austin, Texas, ACM: 597-600.
8. Patil, R. V., S. D. Joshi, S. V. Shinde, D. A. Ajagekar and S. D. Bankar (2015). Code clone detection using decentralized architecture and code reduction. 2015 International Conference on Pervasive Computing (ICPC).
9. Ramasubbu, N. and C. F. Kemerer (2017). "Integrating technical debt management and software quality management processes: a normative framework and field tests." *IEEE Trans Softw Eng* PP(99): 1-1.
10. Evans WS, Fraser CW, Ma F. Clone detection via structural abstraction. *Softw Qual J*. 2009;17(4):309-330.
11. Aversano, L., L. Cerulo and M. D. Penta (2007). How clones are maintained: an empirical study. 11th European Conference on Software Maintenance and Reengineering (CSMR'07).
12. Ernst, N. A., S. Bellomo, I. Ozkaya, R. L. Nord and I. Gorton (2015). Measure it? Manage it? Ignore it? Software practitioners and technical debt. Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. Bergamo, Italy, ACM: 50-60.
13. Harder, J. (2013). How multiple developers affect the evolution of code clones. Software Maintenance (ICSM), 2013 29th IEEE International Conference on, IEEE.
14. Runeson P, Host M, Rainer A, Regnell B. *Case Study Research in Software Engineering: Guidelines and Examples*. John Wiley & Sons; 2012.
15. Mondal, M., C. K. Roy and K. A. Schneider (2017a). Bug propagation through code cloning: an empirical study. 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME).
16. Forbes, C., I. Keivanloo and J. Rilling (2012). Doppel-code: a clone visualization tool for prioritizing global and local clone impacts. 2012 IEEE 36th Annual Computer Software and Applications Conference.
17. Haque, S. M. F., V. Srikanth and E. S. Reddy (2016). Generic code cloning method for detection of clone code in software development. 2016 International Conference on Data Mining and Advanced Computing (SAPIENCE).
18. Mondal, M., C. K. Roy and K. A. Schneider (2017b). Identifying code clones having high possibilities of containing bugs. 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC).
19. Mondal M, Roy CK, Schneider KA. A comparative study on the intensity and harmfulness of late propagation in near-miss code clones. *Softw Qual J*. 2016;24(4):883-915.
20. Kononenko, O., C. Zhang and M. W. Godfrey (2014). Compiling clones: what happens? software maintenance and evolution (ICSME), 2014 IEEE international conference on, IEEE
21. Tiarks R, Koschke R, Falke R. An extended assessment of type-3 clones as detected by state-of-the-art tools. *Softw Qual J*. 2011;19(2):295-331.

22. Kim, S., S. Woo, H. Lee and H. Oh (2017). VUDDY: a scalable approach for vulnerable code clone discovery. 2017 IEEE Symposium on Security and Privacy (SP).
23. Priyambadha, B. and S. Rochimah (2014). Case study on semantic clone detection based on code behavior. 2014 International Conference on Data and Software Engineering (ICODSE).
24. Saha, R. K., M. Asaduzzaman, M. F. Zibran, C. K. Roy and K. A. Schneider (2010). Evaluating code clone genealogies at release level: an empirical study. 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation.
25. Tsantalis N, Mazinanian D, Krishnan GP. Assessing the refactorability of software clones. *IEEE Trans Softw Eng*. 2015;41(11):1055-1090.
26. Chen, Z., M. Mohanavilasam, Y. W. Kwon and M. Song (2017). Tool support for managing clone refactorings to facilitate code review in evolving software. 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC).
27. Duala-Ekoko E, Robillard MP. Tracking code clones in evolving software. In: *29th International Conference on Software Engineering (ICSE'07)*; 2007.
28. Wang X, Dang Y, Zhang L, Zhang D, Lan E, Mei H. Predicting consistency-maintenance requirement of code clonesat copy-and-paste time. *IEEE Trans Softw Eng*. 2014;40(8):773-794.
29. Zhang, F., S. C. Khoo and X. Su (2016). Predicting consistent clone change. 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE).
30. Barbour L, An L, Khomh F, Zou Y, Wang S. An investigation of the fault-proneness of clone evolutionary patterns. *Softw Qual J*. 2017.
31. Zibran, M. F., R. K. Saha, M. Asaduzzaman and C. K. Roy (2011). Analyzing and forecasting near-miss clones in evolving software: an empirical study. 2011 16th IEEE International Conference on Engineering of Complex Computer Systems.
32. Bladel, B. v., A. Murgia and S. Demeyer (2017). An empirical study of clone density evolution and developer cloning tendency. 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER).
33. Goon, A., Y. Wu, M. Matsushita and K. Inoue (2017). Evolution of code clone ratios throughout development history of open-source C and C++ programs. 2017 IEEE 11th International Workshop on Software Clones (IWSC).
34. Robles, G., J. Moreno-León, E. Aivaloglou and F. Hermans (2017). Software clones in scratch projects: on the presence of copy-and-paste in computational thinking learning. 2017 IEEE 11th International Workshop on Software Clones (IWSC).
35. Venkatesh V, Brown SA, Bala H. Bridging the qualitative-quantitative divide: guidelines for conducting mixed methods research in information systems. *MIS Quarterly*. 2013;37(1):21-54.
36. Creswell JW, Creswell JD. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. Sage Publications; 2017.
37. Creswell JW, Clark VLP. *Designing and Conducting Mixed Methods Research*. Sage Publications; 2017.
38. Yin RK. *Case Study Research: Design and Methods, 5th Edtion*. Sage Publications; 2014.
39. Fogel K. *Producing Open Source Software: How to Run a Successful Free Software Project*. O'Reilly Media, Inc; 2005.
40. CQSE (2017). ConQAT tool. https://www.cqse.eu/en/products/conqat/overview/.
41. Juergens E, Deissenboeck F, Hummel B, Wagner S. Do code clones matter? Proceedings of the 31st international conference on software engineering. *IEEE Comp Soc*. 2009;485-495.
42. Svajlenko, J. and C. K. Roy (2015). Evaluating clone detection tools with bigclonebench. Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on, IEEE.
43. Vasilescu, B., V. Filkov and A. Serebrenik (2015). Perceptions of diversity on git hub: a user survey. 2015 IEEE/ACM 8th International Workshop on Cooperative and Human Aspects of Software Engineering.
44. Wloka, J., B. Ryder, F. Tip and X. Ren (2009). Safe-commit analysis to facilitate team software development. 2009 IEEE 31st International Conference on Software Engineering.
45. Ahmed, I., C. Brindescu, U. A. Mannan, C. Jensen and A. Sarma (2017). An empirical examination of the relationship between code smells and merge conflicts. 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM).

How to cite this article: Perez-Castillo R, Piattini M. An empirical study on how project context impacts on code cloning. *J Softw Evol Proc*. 2018;30:e2115. https://doi.org/10.1002/smr.2115