On the impact of Continuous Integration on refactoring practice: An exploratory study on TravisTorrent

Preprint *in* Information and Software Technology · May 2021

DOI: 10.1016/j.infsof.2021.106618

4 authors:

Islem Saidani
École de Technologie Supérieure

Ali Ouni
Osaka University

Mohamed Wiem Mkaouer
Rochester Institute of Technology

Fabio Palomba
Delft University of Technology

On the Impact of Continuous Integration on Refactoring Practice: An Exploratory Study on TravisTorrent

Islem Saidani[a], Ali Ouni[a], Mohamed Wiem Mkaouer[b], Fabio Palomba[c]

[a]*ETS Montreal, University of Quebec, Montreal, QC, Canada*
[b]*Rochester Institute of Technology, Rochester, NY, USA*
[c]*SeSa Lab - University of Salerno, Italy*

Abstract

Context: The ultimate goal of Continuous Integration (CI) is to support developers in integrating changes into production constantly and quickly through automated build process. While CI provides developers with prompt feedback on several quality dimensions after each change, such frequent and quick changes may in turn compromise software quality without Refactoring. Indeed, recent work emphasized the potential of CI in changing the way developers perceive and apply refactoring. However, we still lack empirical evidence to confirm or refute this assumption.

Objective: We aim to explore and understand the evolution of refactoring practices, in terms of frequency, size and involved developers, after the switch to CI in order to emphasize the role of this process in changing the way Refactoring is applied.

Method: We collect a corpus of 99,545 commits and 89,926 refactoring operations extracted from 39 open-source GitHub projects that adopt Travis CI and analyse the changes using Multiple Regression Analysis (MRA).

Results: Our study delivers several important findings. We found that the adoption of CI is associated with a drop in the refactoring size as recommended, while refactoring frequency as well as the number (and its related rate) of de-

URL: `islem.saidani.1@ens.etsmlt.ca` (Islem Saidani), `ali.ouni@etsmtl.ca` (Ali Ouni), `mwmvse@rit.edu` (Mohamed Wiem Mkaouer), `fpalomba@unisa.it` (Fabio Palomba)

velopers that perform refactoring are estimated to decrease after the shift to CI, indicating that refactoring is less likely to be applied in CI context.

Conclusion: Our study uncovers insights about CI theory and practice and adds evidence to existing knowledge about CI practices related especially to quality assurance. Software developers need more customized refactoring tool support in the context of CI to better maintain and evolve their software systems.

Keywords: Continuous Integration, Refactoring, Exploratory Study, Mining Software Repositories, Multiple Regression Analysis

---

1. Introduction

A major challenge in modern software engineering is ensuring the quality of increasingly large and complex software systems. To this end, software development companies have massively adopted Continuous Integration (CI) in order to deliver software with fewer defects and shorter release cycles. CI aims at supporting developers in integrating changes, into a shared repository, more frequently (and even daily) and the key to making this possible, according to Martin Fowler [13], is automating the build and test processes. For its valuable benefits, such as significant improvements in productivity [43], CI has been promoted as the leading edge of software engineering practices [19].

To take full advantage of CI, a set of guiding principles have been introduced to support developers adopting CI in practice. For instance, as advocated by Duvall et al. [9], CI users should continuously inspect code quality, which includes performing Static Code Analysis (SCA), in order to maintain the code of good health. Another key principle is Continuous Refactoring (CR) [7] which consists of *"searching for refactoring opportunities at every completed change and to perform refactoring immediately, without postponing it"* [46]. Indeed, as an *Agile* method, the incremental nature of CI requires the code to be continuously refactored in order to maintain high quality [34] and keep the quality gates, steps required to ensure the reliability of code changes [31], always green

[46]. Otherwise, it may be hard for development teams to understand, maintain and extend their code [37]. Moreover, the absence of CR may result in the need for large refactorings [34] that, like any other complex change, may hinder the CI build progress and requires more debugging effort [56]. Hence, it is encouraged to partition the large change into many smaller ones of few hours each [57].

From the academic side, the adoption of refactoring techniques for CI has received some attention and automatic tools were proposed [51, 2], while others used the outcome of SCA tools to detect refactoring opportunities [49]. However, in practice, there is a lack of empirical knowledge of how refactoring is applied in CI context. The only preliminary study was conducted by Vassallo et al. [46] through a survey with CI developers. Their findings point out the potential of CI to change the way developers adopt refactoring as it is commonly known that the late is often not applied [27, 32, 26] and performed only by specific developers [39]. However, no empirical evidence was provided to confirm this assumption.

In this paper, we want to investigate the possible impact of CI on the way refactoring is applied in practice. First, we study whether CI adoption has increased the likelihood of applying refactoring more frequently. Second, we study whether the size of refactoring changes would decrease after the switch to CI. Third, we study the relationship between adopting CI and the involvement of developers in refactoring activities.

We present an extension of Vassallo et al. [46] work and conduct the first exploratory study involving a benchmark of 99,545 commits and 89,926 refactoring operations during four year development of 39 Open-Source Software (OSS) projects centered around the adoption of Travis CI, a widely used CI service [43]. Using Multiple Regression Analysis (MRA), we show that the adoption of CI is associated with a drop in the refactoring size, which aligns with the "small refactoring" guideline [34], while refactoring frequency as well as the number (and its related rate) of developers that perform refactoring are estimated to decrease after the shift to CI, indicating that refactoring is less likely to occur and, in contrast with the earlier findings [46], refactoring is not spread in CI context. Our MRA also indicates that these trends will continue over time but with different variations between projects with different sizes, ages and releasing frequency. Based on these findings, we conjecture that software developers may need more customized refactoring tool support in the context of CI to better maintain and evolve their software systems.

Replication Package. The dataset used in our study is publicly available for future replication and extension purposes [30].

Structure of the paper. The remainder of this paper is organized as follows. Section 2 places this work with respect to the existing literature. We present our research methodology in section 3, while present and analyze the obtained results in Section 4. In Section 5, we discuss the obtained results and their implications. Then, we review threats to validity in Section 6, and finally we address the conclusions to draw in Section 7.

2. Related Work

In the following subsections, we present the work most related to our study. We divide the prior work into three main areas: work related to CI impacts on software quality and development practices, works on the challenges, barriers and bad practices in CI and finally studies related to code refactoring.

2.1. Studies about the impacts of CI adoption

As we summarise in Table 1, research works have focused on studying the outcomes of the adoption of CI on teams' productivity, development practices and code quality thanks to the increasing availability of publicly hosted Travis CI data [5]. In particular, Vasilescu et al. [43] have found, using multiple regression modeling, that CI improves the number of processed Pull Requests(PRs), *i.e.*, a submitted candidate code change to be merged into the mainline repository, and reduces the quantity of rejected ones, indicating a significant improvement in the team's productivity, and this without affecting code quality measured as the number of closed bugs per month. Hilton et al. [19] claimed also that CI improves team's productivity. Indeed, they found that after adopting CI (i) the studied CI projects release twice more than those that do not use CI and (ii) the PR is accepted faster. Yu et al. [54] studied the acceptance and latency of PR in CI context. Using regression models in a sample of 10 GitHub projects that use Travis CI, the authors found that the availability of the CI pipeline is a dominant factor in hastening the PR evaluation process. Yu et al. [53] studied the nature of CI detected defects and social factors are associated with them and how they relate to eventual bugs. To this end, they performed both quantitative and qualitative analysis: regression modeling and a qualitative study of 50 PRs. The main results of their work are (1) CI failures are not highly correlated with eventual bugs, (2) A mature CI process is associated with better fault detection and (3) The use of CI in a PR doesn't necessarily mean having a request of good quality. Zhao et al. [57] used regression discontinuity design [20] to quantitatively evaluate the effect of adopting CI on development practices, such as code writing and submission, issue and PR closing. The main result of their study is that CI practice aligns with the "commit often" guideline [13] while merged commits seem to be getting smaller as recommended by Fowler.

While studies mentioned above suggest that the adoption of CI increases the release frequency of a software project, other works did not observe such an increase in their quantitative analyses. For instance, Bernardo et al. [6] have observed, by training regression models, that CI does not always reduce the time for delivering merged PRs. Their models also reveal that PRs, that are merged more recently in a release cycle, experience a slower delivery time. Rahman et al. [29] have observed for the studied OSS projects some CI benefits *e.g.*improvements in bug and issue resolution. However, for the proprietary projects, they could not make similar observations.

Table 1: A summary of the literature on the impact of CI adoption on software quality/development.

Study	Year	Studied impact(s)	Methodology	Results
Vasilescu et al. [43]	2015	Quality and Productivity	Regression Analysis of 246 Travis CI projects	CI improves the productivity without an observable diminishment in code quality.
Hilton et al. [19]	2016	Productivity	-Mining 1,529,291 builds from Travis CI	• Projects that use CI release more than twice as often as those that do not use CI. • The PR is accepted sooner.
Yu et al. [54]	2016	Productivity	Regression Analysis of top 10 Travis CI projects	the presence of CI is a dominant factor for both PR acceptance and latency
Yu et al. [53]	2016	Quality	Regression Analysis of 246 Travis Ci projects	• CI failures are not highly correlated with eventual bugs, • A mature CI process is associated with better fault detection • The use of CI in a PR does not necessarily mean having a request of good quality.
Zhao et al. [57]	2017	development practices	Regression Analysis of 575 Travis CI projects	CI adoption is associated with: • An increase in the number of merged commits • the "commit small" guideline is followed to some extent • An increasing trend in the number of closed PR • An increase in the PR latency • A drop trend in the number of issues closed issues.
Bernardo et al. [6]	2018	Productivity	Regression Analysis of 162,653 PR of 87 Travis CI projects	CI does not always reduce the time for delivering PRs
Rahman et al. [29]	2018	Productivity and quality	Mining 150 OSS and 123 proprietary projects	- Closed bugs, closed issues, and frequency of commits, significantly increased after adoption of CI for OSS projects, but not for proprietary projects.

2.2. Studies on the challenges, barriers and bad practices of CI

Despite its valuable benefits, previous studies have pointed out challenges and barriers characterizing CI adoption. For instance, Hilton et al. [18] have found that developers face trade-offs between speed and assurance, between better access and information security, and between more CI configuration options and better flexibility in use.

Build failure is considered a major challenge that developers face [18] as it requires immediate actions to resolve it. In addition, the build resolution may take hours or even days to complete, which severely affects both, the speed of software development and the productivity of developers [1] and may lead to CI abandonment [50].

Another CI barriers are due to social processes within the team, the frequent turnover of developers after introducing CI and the wide variations in their coding experiences is one of challenges for CI process's success. For instance, Lu et al. [24] results show that the casual contributors introduced greater quantity and severity of code quality issues than the main contributors.

Research efforts also reported some bad practices that developers usually incur, limiting the effectiveness of CI. For instance, Vassallo et al. [45] revealed a strong dichotomy between theory and practice in CI context as they found that developers do not perform continuous code inspection but rather control for quality only at the end of a sprint and most of the times only on the release branch.

Felidré et al. [11] have investigated a set of CI bad practices including infrequent commits, poor test coverage and broken builds for long periods. By inspecting 1,270 OSS projects that use Travis CI, they observed that (i) 60% of the studied projects face infrequent commits, (ii) the average code coverage was 78% among 51 projects in which they were able to find code coverage and (iii) 85% of the studied projects have at least one broken build that takes more than four days to be fixed.

Zampetti et al. [55] compiled a catalog of 79 CI bad smells belonging to 7 categories related to CI pipeline management and process. As the main result, they found some CI bad smells related to quality assurance. For example, a branch is not tested before merging it, quality test thresholds are fixed on what reached in previous builds and quality gates are set without being relevant for developers and/or customers.

The closest work to ours is by Vassallo et al. [46]. They provide a preliminary overview of the way refactoring is applied in CI. The authors conducted a survey study that involved 31 developers to understand (*i*) how developers perform refactoring and (*ii*) what are the pros and cons of adopting Continuous Refactoring (CR). Their findings showed that developers tend to perform refactoring at every new build and they need CR. Still they face several barriers while refactoring especially with the lack of time. In this paper, we presented an extension of Vassallo et al. [46] work, we showed to which extent refactoring is performed in practice.

2.3. Studies about code refactoring

A series of interesting works in the field of refactoring have been published to made strides into understanding the practice of refactoring. For instance, Negara et al. [27] provided a detailed breakdown on the manual and automated usage of refactoring, using continuous code change analysis of Eclipse IDE users. Their main findings are (i) more than half of the refactorings are performed manually (52%) (ii) except for renaming refactorings, the automated refactorings are underused. Tsantalis et al. [39] investigated refactoring activity as part of the software engineering process. They have identified that the refactoring application is often performed by specific developers. They also found a strong alignment between refactoring activity and release dates and revealed that the development teams apply a considerable amount of refactorings during releasing periods.

Many studies have investigated the relationship between refactoring and software quality. Kim et al. [21] conducted a survey performed with professional software engineers working at Microsoft and a quantitative analysis of version history data, to understand refactoring benefits and challenges. The main findings of this study are: (i) the most important motivation that pushes developers to perform refactoring is to enhance the readability of source code and (ii) the quantitative analysis revealed a significant reduction in the number of defects indicating a visible benefit of refactoring. Szóke et al. [37] analyzed the source code of five software systems to investigate the relationship between refactoring and code quality. They found that atomic refactoring operations performed in isolation make a small change. However, when refactoring is performed in sequence, we can perceive a significant increase in quality. Moser et al. [25] conducted a case study in an industrial software project aimed at investigating the impact of refactoring on reusability. The achieved results sustain the hypothesis that refactoring enhances quality and reusability of classes.

3. Study Design and methodology

The *goal* of this study is to investigate the possible impact of CI adoption on refactoring activities by analyzing how developers change the way they refactor their software systems in practice. In this section, we define our research questions and present the design of our study.

3.1. Research questions

The study aims at addressing the following research questions:

RQ1. Does CI impact the refactoring frequency? In this first RQ, we are particularly interested in investigating how frequently developers refactor their software systems after the adoption of CI. Our motivation is based on the fact that the aim of CI is to get changes into production as quickly as possible, without compromising software quality. We speculate that without continuous refactoring, such frequent and quick changes during the CI process may negatively affect some quality attributes such as readability, understandability, flexibility, etc. [45]. Indeed, refactoring is known to have a paramount importance to deliver a high-quality software product, by removing defects and reducing technical debt [15] which are introduced by quick and often unsystematic development [34].

RQ2. Does the adoption of CI affect the refactoring change size? In this research question, we want to assess the size of the changes related to refactoring through the software system before and after the adoption of CI. Indeed, refactoring is recommended to be small in size [34] as this would (i) help developers track the progress, (ii) reduce the risk of introducing complexity or defects during refactoring and (iii) avoid breaking the build [56]. Hence, we expect that after adopting CI, developers would integrate refactoring related changes with smaller chunks.

RQ3. How are developers involved in code refactoring before and after the adoption of CI? The motivation of this research question stems from previous research works [39] confirming that refactoring is performed by specific developers that usually have a key role in the management of the project. In this study, we want to analyze whether CI raises the code authorship, *i.e.*, motivation to program the code with high quality by performing the refactoring [48].

3.2. Methodology

Figure 1 provides an overview of our research methodology to address our defined research questions. Our methodology comprises three main steps: (*i*) context selection, (*ii*) refactoring data extraction, and (*iii*) analysis method. In the following, we present the details of each of these three steps.

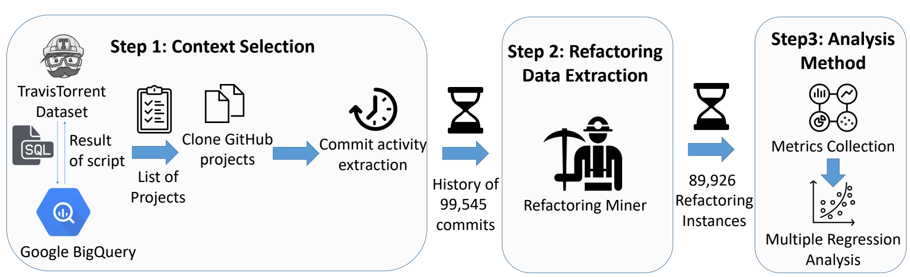


Figure 1: An overview of our research methodology to study the refactoring practices in CI.

3.2.1. ***Step 1:*** *Context Selection*

We gather our dataset from 39 OSS projects hosted on GitHub which have switched to Travis CI, a widely used CI system, at some point during their life-cycle. To answer our research questions, we mined these projects based on the latest TravisTorrent dump dated on 2017/02/08[1] and using the Big Query Google Tool [2] to query pieces of information such as the programming language and the repository URL. The choice of the subject systems was driven by the following criteria:

- Projects with sufficiently long historical code change records, *i.e.*, at least two years before and after the adoption of CI to get deep insights into the possible impacts and feed our regression models with sufficient data.
- Projects that have a consistent change activity during the studied period *i.e.*, having at least one merged commit in the mainline branch each month for the studied period. We chose a monthly partition following previous studies on the impact of CI [53, 16, 29, 43, 57] because (*i*) it leads to more meaningful results than providing only one value per year and (*ii*) to fit our regression models and control for time variable. Hence, we avoid biasing our results with zero values due to projects not being active during some months (thus no refactoring activities will take place).
- We also restricted our analysis to Java projects as we rely on the *RefactoringMiner* tool [40], an automated tool for detecting refactoring activities applied in software projects during their development life-cycle (Section **-B).

Thereafter, we cloned all project repositories and extracted all their commits change history to be used in next steps. We recorded a total of 99,545 commits on the mainline branch for the studied projects. Table 2 reports the analyzed

[1] `https://travistorrent.testroots.org/`
[2] `https://bigquery.cloud.google.com`

projects, the number of commits, refactoring related commits and contributors. Moreover, we report other historical statistics about the projects such as the age in months and the number of releases. All the data collected and used in our exploratory study is publicly available for replication purposes in our comprehensive replication package [30].

3.2.2. ***Step 2:*** *Refactoring Data Extraction*

We use in our study the tool *RefactoringMiner*[3], a commit-based refactoring detection tool that is based on the UMLDiff algorithm [52] for computing the differences between object-oriented models [41]. Table 3 presents the list of refactoring operations that can be detected by RefactoringMiner with their respective number of refactoring instances identified in the 39 projects involved in our study. We selected the refactoringMiner tool as it provides high precision of 98% and recall of 87% [40], implements the detection of over 32 refactoring operations, and has been widely used in recent empirical studies [38, 32, 44, 3].

[3] `https://github.com/tsantalis/RefactoringMiner`

Table 2: Systems involved in the study

Project	Description	Historical Statistics			Considered in the study		
		Age	Total Commits	Total Contributors	# of releases	# of commits	# of ref. commits
airlift/airlift	Framework for building REST services	37	2,371	73	227	905	114
apache/pdfbox	Mirror of Apache PDFBox	75	8,120	24	49	4,340	801
apache/storm	Mirror of Apache Storm	42	7,321	477	39	4,722	412
aws/aws-sdk-java	The official AWS SDK	36	1,835	188	787	291	110
chocoteam/choco3	A Java library for Constraint Programming	35	3,819	30	28	2,328	565
dropwizard/dropwizard	A library for RESTful web services	26	3,905	444	143	2,240	290
druid-io/druid	A real-time analytics database.	25	7,330	381	428	5,308	881
DSpace/DSpace	A digital asset management system	134	9,209	213	95	2,370	236
FasterXML/jackson-databind	Data-binding package	25	4,237	184	115	2,225	545
FenixEdu/fenixedu-academic	Student Information System	123	36,934	162	345	5,517	644
geoserver/geoserver	Open source software server	27	7,568	295	123	3,562	575
GeoWebCache/geowebcache	Caching server	101	2,134	75	122	461	80
google/error-prone	Static analysis tool	35	3,597	222	31	1,497	307
google/guava	Google core libraries	61	4,995	319	87	2,383	393
grails/grails-core	Grails Web Application Framework	106	16,152	329	189	5,513	405
igniterealtime/Openfire	A XMPP server	115	7,931	183	158	1,377	201
jOOQ/jOOQ	Light database-mapping software library	23	7,135	84	73	4,137	697
jpos/jPOS	Open source library/framework	179	4,378	74	49	713	62
junit-team/junit	A testing framework	155	2,002	207	23	843	137
lenskit/lenskit	Recommender toolkit	40	5,884	50	52	3,575	588
maxcom/lorsource	Website engine	64	6,759	89	1	3,500	415
mybatis/mybatis-3	SQL mapper framework	32	2,399	142	29	1,220	146
nutzam/nutz	Web Framework	47	5,379	94	57	1,897	256
oblac/jodd	An open-source Java utility library	53	5,055	63	54	2,446	597
orbeon/orbeon-forms	Open source web forms solution	90	22,092	36	50	4,844	304
owncloud/android	Android App	28	6,141	91	92	3,607	511
perfectsense/brightspot-cms	Enterprise user experience platform	32	5,678	49	23	4,557	298
proofpoint/platform	Security Awareness & Education Platform	49	3,132	69	216	1,203	187
sparklemotion/nokogiri	Web parser	36	4,013	195	147	1,585	81
spring-data-commons	shared infrastructure across the Spring Data	47	1,891	91	155	714	147
tananaev/traccar	GPS Tracking System	58	5,214	113	37	3,162	388
TGAC/miso-lims	An open-source LIMS for NGS sequencing centres	58	3,209	25	219	2,908	450
tinkerpop/blueprints	A Property Graph Model Interface	28	1,532	64	19	1,414	322
tinkerpop/rexster	A Graph Server	26	1,476	26	17	1,400	259
twall/jna	Java Native Access	171	3,112	170	52	1,272	125
Unidata/thredds	A middleware	86	9,780	63	60	3,739	1,122
weld/core	Integrations for Servlet containers and Java SE	71	7,534	108	160	2,351	501
xtreemfs/xtreemfs	Distributed Fault-Tolerant File System	66	4,742	52	20	2,175	255
zxing/zxing	Barcode scanning library	74	3,434	143	27	1,244	118
	Median	**49**	**4,995**	**94**	**60**	**2,328**	**307**
	Average	**64.5**	**6,395.6**	**146.1**	**117.9**	**2,552.4**	**372.4**
	Total	**-**	**249,429**	**5,697**	**4,598**	**99,545**	**14,525**

Table 3: Analysed Refactoring operations statistics with their different levels.

Refactoring Operation	**Level**	**Instances**	**Projects**
Move Class	Class	13,312	38
Rename Method	Method	10,749	39
Rename Variable	Block	9,527	39
Rename Attribute	Field	7,341	39
Rename Parameter	Block	6,706	39
Extract Method	Method	6,154	39
Pull Up Attribute	Field	5,780	38
Move Method	Method	5,527	39
Move Attribute	Field	3,691	39
Pull Up Method	Method	3,414	39
Extract Variable	Block	2,964	39
Rename Class	Class	2,855	39
Inline Method	Method	2,009	39
Push Down Method	Method	1,077	36
Extract Class	Class	997	39
Move And Rename Class	Class	915	37
Move Source Folder	Package	655	31
Inline Variable	Block	653	39
Push Down Attribute	Field	602	30
Extract Super-class	Class	553	37
Parameterize Variable	Method	479	38
Replace Variable With Attribute	Block	461	36
Extract Interface	Class	324	32
Change Package	Package	305	28
Extract Subclass	Class	126	32
Move And Rename Attribute	Field	32	13
Replace Attribute	Field	24	8
	Total	89,926	39

3.2.3. ***Step 3:*** *Analysis Method*

Used Metrics:

To address **RQ1**, we define two measures including the number of refactoring commits per month (NRC) and the refactoring rate (RRC) as follows:

- **NRC:** *Number of Refactoring Commits.* It counts the number of merged refactoring commits, *i.e.*, code commits that have at least one refactoring operation applied, in each month.

- **RRC:** *Rate of Refactoring Commits* which computes the ratio of refactoring commits (NRC) among the total number of merged commits (NC) per month. This measure gives insights about the extent to which developers tend to refactor their code during the development of their projects.

To answer **RQ2**, we capture the change size of a refactoring commit. For this aim, we define the following measures. Note that each mean value bellow is computed over all refactoring commits in the considered month.

- **RB:** *Refactoring Breadth.* The average number of files where at least one refactoring operation was applied per commit.

- **RBR:** *Refactoring Breadth Rate.* The average rate of refactoring breadth per commit. The rate refers to the number of files related to refactoring divided by the total number of modified files.

To answer **RQ3**, we assess the extent to which developers are involved in refactoring activities before and after the adoption of CI by defining the following metrics:

- **NRefDev:** *Number of Refactoring Developers.* Counts the number of developers who applied at least one refactoring per month.

- **RRD:** *Rate of Refactoring Developers.* The ratio of the number of committers who applied refactoring in their commit changes divided by the total number of committers.

Multiple Regression Analysis

To evaluate the effects of the adoption of CI (RQs 1-3), we use Multiple Regression Analysis (MRA) [10] as a method for analyzing the relationship between a set of explanatory variables (predictors, *e.g.*the time in months) and a response (outcome, *e.g.*the rate of refactoring commits), while controlling for known covariates (*e.g.*, project age) that might influence the response. Solving the regression gives us the coefficients for each predictor. If the coefficient is significant, it can help us reason about the treatment (*e.g.*, the adoption of CI in our case) and its effects, if any, while controlling for confounding variables. In our study, we perform our MRA to estimate the trends in our set of metrics (Section 3.2.3) marked as Y before the adoption of CI, and the changes in the trend after the adoption CI as follows:

$$y_i = \alpha + \beta * \textit{time\_before\_ci}_i + \gamma * \textit{time\_after\_ci}_i + \delta * \textit{ci\_is\_adopted}_i + \epsilon$$

where *time_before_ci* indicates the time in months at time i from the start of the observation and coded 0 after CI (*i.e.*, from -24 to -1); *time_after_ci* counts the number of months at time i after the CI adoption and coded 0 before the adoption (*i.e.*, from 1 to 24); *ci_is_adopted* indicates whether CI is adopted at time i (ci_is_adopted = 1) or not (ci_is_adopted = 0). Using this model, we can capture any divergence (regression) in the slopes (decrease/increase) before and after the adoption of CI. Moreover, we consider the following confounding variables (ϵ):

- **Total number of commits (TotalComm).** Following Zhao et al. [57], we consider the total number of commits in a project's history as an indicator for project activity/size.
- **Total number of developers (TotalDev).** We also consider the total number of developers as a proxy for the project's community size.
- **Project age at the time of CI adoption (AgeAtCI)** in months. Mature projects may be less affected by the adoption of CI than other projects [57].
- **Number of releases (NReleases)** We manually checked the timeline of each project to collect its number of releases. We want to inspect the

releasing frequency on refactoring practice as it is known that projects with frequent releases may have the chance to fix bugs faster [17] and hence apply more refactorings.

We implement the MRA using the function `lm` from `lmerTest`[4] package in R. Log transform predictors [8] are used to stabilize the variance and improve model fit. To avoid multicollinearity phenomenon in which one predictor variable can be linearly predicted from the others [8], we consider the Variance Inflation Factor (package `car`[5] in R). To improve robustness, the top 3% of the data was filtered out as outliers in order to avoid inflating the model's fit [43]. For each model, we report (*i*) the coefficients that describe the mathematical relationship between each independent variable and the dependent variable and higher values suggests higher effect, (*ii*) $\rho - values$ that provide the significance level of the coefficients, (*iii*) the sum of squares which computes the variance explained by each variable, and (*iv*) the standard error which indicates how wrong the regression model using the units of the response variable; smaller values are better to provide evidence of the fitted model.

4. Study Results

In this section, we present and discuss the results of our study to answer our research questions RQ1-3. All the data collected and used in our study is publicly available for replication and extension purposes in our comprehensive replication package [30].

For the sake of clarity, the key metrics used in our study are shown in Table 4. The results of our MRA are presented and discussed in the next section.

4.1. RQ1: Trends in refactoring frequency after the adoption of CI

We start by quantifying the trends in the number of refactoring commits (NRC) and Rate of Refactoring Commits (RRC) using the Multiple Regression

[4]`https://cran.r-project.org/web/packages/lmerTest/index.html`
[5]`https://cran.r-project.org/web/packages/car/car.pdf`

Table 4: Summary of the study measures.

Metric	**Description**
NRC	Number of Refactoring Commits
RRC	Rate of Refactoring Commits
RB	Refactoring Breadth
RBR	Refactoring Breadth Rate
NRefDev	Number of Refactoring Developers
RRD	Rate of Refactoring Developers
TotalComm	Total number of commits
TotalDev	Total number of developers
AgeAtCI	Project age at the time of CI adoption
NReleases	Number of releases

Analysis (MRA) as described in Section 3.2.3. Table 5 summarizes the regression analysis results for refactoring frequency measures. For each variable, we report its coefficients (*Coeff*) and corresponding sum of squares (*Sum Sq*), a measure of variance for each variable and the standard error of the regression (*Error*) which represents the average distance between the observed values and the regression line. The statistical significance is indicated by stars symbols. We consider coefficients to be important if they are statistically significant ($\rho < 0.05$).

From the obtained results in Table 5, the NRC model confirms a statistically significant negative baseline trend in the response with *ci_is_adopted* which means that the number refactoring commits would decrease after introducing CI. The coefficient for time is negative, suggesting a decreasing baseline trend in terms of refactoring commits after the adoption of CI. However, the model does not detect any effect for the time before the adoption of CI since the coefficient *time_before_ci* is not statistically significant. Overall, the trend remains descending (the sum of the coefficients for *time_after_ci* and *ci_is_adopted* is negative): less refactoring commits after the adoption of CI.

Table 5: MRA results for refactoring frequency in terms of Number of Refactoring Commits (NRC) and Rate of Refactoring Commits (RRC).

Metric	**NRC Model**				**RRC Model**			
	Coeff	*Error*	ρ	*Sum Sq.*	*Coeff*	*Error*	ρ	*Sum Sq.*
Intercept	-11.76	6.39	.		0.34	0.11	**	
ci_is_adopted	-2.20	0.51	***	548.9	-0.01	$7.9*10^{-3}$	*	0.041
time_before_ci	0.01	0.02		9.25	$-7.9*10^{-4}$	$4.0*10^{-4}$	.	0.027
time_after_ci	-0.12	0.02	***	631.8	$-7.9*10^{-4}$	$4.0*10^{-4}$	.	0.027
log(TotalComm)	4.07	0.72	***	947.4	-0.01	0.01		0.004
log(TotalDev)	-0.72	0.61		42	-0.01	0.01		0.019
log(AgeAtCI)	-3.12	0.83	***	423.1	-0.02	0.01		0.013
log(NReleases)	0.09	0.43		1.51	0.01	$7.7*10^{-3}$		0.016

*** $\rho < 0.001$, **$\rho < 0.01$, *$\rho < 0.05$, '.' 0.1 ' ' 1

Next, we assess the confounding variables namely the project size in terms of total number of commits, developers, project age, and number of releases. As reported in Table 5, the NRC model confirms a statistically significant, positive, baseline trend in the response with project size ($TotalComm$) which explains an important amount of variability in the response (*Sum Sq.* = 947.4). This finding suggests that refactoring is performed more frequently within bigger projects as they are more active and have larger codebase. For example, in `Unidata/thredds` project for which we recorded a 9,780 of commits, developers merged 20 refactoring commits per month on median, while in `airlift/airlift` project with 2,371 commits, developers tend to merge about 2 refactoring commits per month on median for the studied period. Also, the model reveals a particular trend for older projects ($AgeAtCI$) to apply less code refactorings. This finding is quite surprising since it is commonly admitted that as projects age, the maintenance focus is generally shifted to bug-fixing [57] or quality assurance to master the increasing software complexity [23] which is usually performed through the assistance of refactoring [14]. Moreover, we observe that the team size ($TotalDev$: the total number of commit authors over the entire history) has no statistically significant effect which means that projects with larger committers base do not necessarily apply more the refactoring (cf. 5). For example, `apache/storm` project which has the larger base of contributors in our dataset with 477 contributors, developers tend to merge 6 refactoring commits per month while in `TGAC/miso-lims` with 25 contributors, we recorded a median number of refactoring commits of 9 in the studied period. Furthermore, we found no evidence for the releasing frequency (*NReleases*) to affect refactoring frequency estimators which means that, for the studied projects, a higher releasing frequency does not necessarily imply that developers apply more refactoring.

Looking at the rate of refactoring commits, *i.e.*, the RRC model, we see that the only significant predictor is the CI adoption variable suggesting that the rate of refactoring commits would decrease after introducing CI with a slight decrease trend of 0.01. The model reveals no evidence for time variables to be effective as the coefficients are not significant. With regard to the confounding variables, we observe that all the studied project characteristics have no significant effect.

> *Our MRA study results suggest that the adoption of CI can result into a decrease in terms of refactoring frequency. However, the regression analysis reveals that projects with larger size are less sensitive to this trend. Moreover, the MRA models suggest the more aged is the project, less performed is the refactoring.*

4.2. RQ2: Trends in refactoring change size after the adoption of CI

In this research question, we are particularly interested in exploring the possible effects of CI on refactoring breadth. Hence, we analyze by using MRA models the relationships between CI related variables and metrics related to refactoring churn and breadth while controlling for confounding variables. The MRA models for refactoring breadth are summarized in Table 6.

First, Table 6 reveals a significant drop in the number of changed files related to refactoring after the adoption of CI since *ci_is_adopted* variable is statistically significant but with no significant effect for the time which indicates that this

Table 6: MRA analysis results for the refactoring breadth (RB) and the refactoring breadth rate (RBR).

Metric	**RB Model**				**RBR Model**			
	Coeff	Error	$\rho - value$	Sum Sq.	*Coeff*	*Error*	$\rho - value$	*Sum Sq.*
Intercept	1	1.9			0.3	0.2		
ci_is_adopted	-0.6	0.2	**	51.6	-0.04	0.02	*	0.22
time_before_ci	$5{*}10^{-3}$	0.01		1.1	$9{*}10^{-4}$	10^{-3}		0.03
time_after_ci	-0.02	0.01	*	33.8	$-2{*}10^{-3}$	10^{-3}	*	0.24
log(TotalComm)	0.4	0.2	.	22.75	0.04	0.02		0.13
log(TotalDev)	-0.1	0.1		2.1	$-{*}10^{-3}$	0.02		$2{*}10^{-3}$
log(AgeAtCI)	-0.6	0.2	*	36.8	-0.02	0.02		0.02
log(NReleases)	0.2	0.1		14.7	-0.03	0.02	*	
R_c^2		0.14				0.19		
R_m^2		0.04				0.04		

*** $\rho < 0.001$, **$\rho < 0.01$, *$\rho < 0.05$, '.' 0.1 ' ' 1

trend may change over time. This result reveals that *refactoring tends to be less diffused after the adoption of CI*. Looking at the confounding variables, we see through the RB model that in aged projects, refactoring changes tend to affect fewer files since the relative coefficient (-0,6) is negative while in the RBR model, this effect was not significant. Additionally, we found no evidence for the project size to affect the refactoring breadth. Moreover, we see that the rate of refactoring breadth slightly decreases after CI with a higher frequency of releasing as suggested in the RBR model indicating that $NReleases$ predictor has a significant effect on the response variables.

Our MRA results reveal that the refactoring tend to affect less files after the adoption of CI. Additionally, our model suggests a slight drop in the the relative rate after the adoption of CI but with different variations between projects especially with higher releasing frequency.

4.3. RQ3: How are developers involved in refactoring activities?

In this research question, we analyze using MRA whether the adoption of CI impacts the way developers are involved in refactoring activities. The statistical model for the number of refactoring developers and its relative rate are summarized in Table 7.

Table 7: The MRA analysis results for the Number of Refactoring Developers (NRefDev) and the Rate of Refactoring Developers (RRD).

Metric	**NRefDev Model**				**RRD Model**			
	Coeff	Error	$\rho - value$	Sum Sq.	*Coeff*	*Error*	$\rho - value$	*Sum Sq.*
Intercept	-4.4	1.8	*		0.82	0.28	**	
ci_is_adopted	-0.39	0.1	***	17.2	-0.12	0.02	***	1.83
time_before_ci	0.01	$5*10^{-3}$	***	17.4	$-2*10^{-3}$	10^{-3}	*	0.33
time_after_ci	-0.02	$5*10^{-3}$	***	19.6	$-5*10^{-3}$	10^{-3}	***	1.12
log(TotalComm)	0.7	0.2	***	17.5	0.04	0.03		0.14
log(TotalDev)	0.2	0.1		2.05	-0.1	0.02	***	0.88
log(AgeAtCI)	-0.39	0.2		3.3	-0.08	0.03	*	0.29
log(NReleases)	0.14	0.12		1.66	$7*10^{-3}$	0.01		0.01
R^2_c			0.45				0.31	
R^2_m			0.16				0.13	

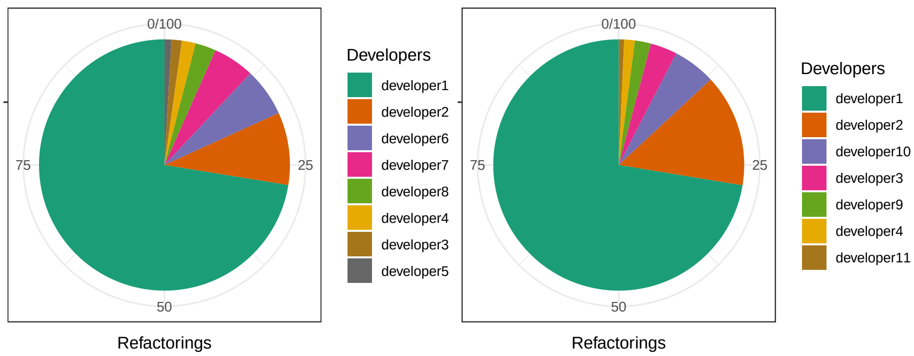
*** $\rho < 0.001$, **$\rho < 0.01$, *$\rho < 0.05$, '.' 0.1 ' ' 1

Looking at the number of refactoring developers (NRefDev) model, we observe that the *time_after_ci* and *ci_is_adopted* predictors exhibit negative coefficients scores of -0.02 and -0.39, respectively. We first note such a slight increasing trend in the number of refactoring developers prior to the adoption of CI, although the trend slows down following the adoption of CI. In addition, our model results indicate that the variable counting for the project size ($TotalComm$) behaves consistently with NRC model 5: bigger projects tend to have larger base of refactoring developers. Moreover, we found no evidence for the contributor base ($TotalDev$) to have any effects which indicates that a large number of contributors does not imply having more developers to apply refactoring. Neither the age nor the releasing frequency seems to have any significant effect.

With regard to RRD model, we observe a significant negative for time variable before the adoption of CI which remains decreasing after the switch to CI considering its related predictors. When we look at the confounding variables, we observe also a significant negative trend for the variables accounting for the age. Another important result to highlight is the significant negative effect of the contributor base ($TotalDev$) on the rate of refactoring developers which indicates that the more involved developers are in the project, the less is the rate of those who apply refactoring. Overall, this model suggests that the rate of refactoring developers tends to decrease as the time passes and this trend is slightly accelerated after the adoption of CI.

To get more insights, we provide an example extracted from our dataset namely MYBATIS-3 project[6] which is an SQL mapper framework for Java. During its development, the `Mybatis` project version control system involves, for the studied period, 8 developers before the adoption of CI and 40 developers after its adoption. Figures 2a and 2b show the percentage of the refactoring operations performed by `Mybatis` developers before and after the adoption of CI, respectively. While all the developers have applied at least one refactoring before CI (8/8 developers), refactoring activities were performed by a limited number of developers after the adoption of CI (7/40 developers). Additionally, the top-one refactoring developer, namely "developer1" (a core team member), performed 72% of the refactoring commits before and after the adoption of CI. He is also the top-one committer with 67% and 52% of the commits before and after the adoption of CI, respectively. These observations are consistent with previous results [39] claiming that most of the applied refactorings are generally performed by specific developers (usually core team members). Moreover, we can confirm a previous assumption about developers attraction in the context of CI [43].

[6]https://github.com/mybatis/mybatis-3



(a) Before CI (b) After CI

Figure 2: Distribution of refactoring contributors in the project `mybatis/mybatis-3`.

> *MRA results reveal a decreasing trend for the rate of refactoring developers especially with the adoption of CI with a considerable negative effect for aged projects and those with larger number of contributors. This may be due to the fact that the refactoring is usually performed by particular developers and as the contributors base gets larger after the adoption of CI, the refactoring rate will decrease.*

5. Discussion

In this section, we further discuss the main findings of our study along with outlining their practical implications for future research on the refactoring of modern systems.

Refactoring is less applied in CI. Our results reveal that the refactoring frequency tend to decrease after the shift to Travis CI. This finding was surprising as CI principles may suggest developers to refactor their code more frequently to improve software quality. This may be due to the fact that CI developers may not consider quality degradation to affect the success of the build process as stated by Vassallo et al. [47]. Based on this finding, we believe that future research effort should be devoted to build techniques able to increase the developer's awareness of refactoring in the context of CI, for instance through improved visualization approaches that may graphically show to developers how a certain refactoring action, conducted at build-time, would be beneficial for the quality of source code.

Towards Just-In-Time refactoring recommendation. Our results for RQ2 reveal that developers tend to make smaller refactoring changes to software projects, as they have a lower refactoring breadth, which is consisting with "refactor smaller" [34] and "commit smaller" [13] guidelines. We believe that this finding would encourage tool builders to conceive refactoring recommendation systems that can be adopted in a CI environment and able to recommend micro-refactoring operations or, even better, small local refactoring operations that targets specific files touched by developers during a code change (*i.e.*, commit). These just-in-time refactoring tools would (*i*) avoid changing the program design radically, and (*ii*) allow developers reviewing the recommendations, and their relative impacts and hence easily decide whether to apply or ignore them. Such tools could be in the form of refactoring recommendation systems or bots that can be integrated into existing CI systems. While some preliminary research has been conducted toward this direction [22, 28, 42, 2, 51], we believe that additional effort is needed to build refactoring tools that more properly reflect the developer's needs in the context of CI development.

Support for newcomers to better practice refactoring. To survive and thrive, a software project must attract, support and retain new developers and help them be productive. However, our findings show that newcomers may be reluctant to practice refactoring activities in the project: these are perfectly in line with the results reported by previous studies on the barriers that newcomers face when joining a new project [36, 35]. However, our study shows that an additional barrier consists of newcomers not being able to refactor source code to improve its quality. Based on this result, we envision a novel category of tools that may support newcomers when performing code quality tasks: more specifically, tool builders should provide development teams with more practical tools and/or techniques for supporting newcomers during the integration in the development team as well as instruments that community shepherds may use to identify the developers having adequate skills to properly guide the newcomers in their refactoring phases.

6. Threats to validity

A number of possible threats might affect the validity of our empirical study.

Internal threats to validity of our study could be related to the accuracy of the used data collection tools. We opted for RefactoringMiner, an open-source tool that has a high F1-score of 81% according to recent experiments conducted in [38]. To alleviate any potential threats with RefactoringMiner, we are planning to replicate our study with other existing tools such as *RefDiff*, a state-of-the-art refactoring detection tool that has shown a high accuracy [33]. More interestingly, to enable other researchers to verify and extend our study, we provide our replication package along with detailed results available for the research community [30].

Construct threats to validity are mainly related to the fact that some projects may leave CI systems after its adoption [50]. To address this issue, we manually checked whether Travis-CI was disabled/abandoned by investigating all the commits in which the CI configuration file was modified and found that none of our studied projects has abandoned CI during the studied period. Another potential threat could be related to selecting projects that used another CI system before adopting Travis-CI. Hence, we mitigated this issue by inspecting the existence of any other CI configuration file (*e.g.*,".appveyor.yml" for AppVeyor CI system) before the adoption of Travis-CI. In this investigation, we considered AppVeyor,[7] Circle-CI,[8] and Drone.[9] Additionally, we checked that our studied projects never used a self-hosted CI system (*i.e.*, using their CI ser-

[7] `https://www.appveyor.com/`
[8] `https://circleci.com/`
[9] `https://drone.io/`

vice locally) like Jenkins,[10] Team-city,[11] or Easy-CIS,[12] by inspecting whether the commit messages contained the name of the above mentioned CI systems.

Conclusion threats to validity refer to issues that affect our ability to draw the correct conclusions and the way we estimated refactoring practice. The fact that developers did not apply more frequently/intensively refactoring, this does not mean that they did not search for refactoring opportunities. In other terms, developers could have some recommendations of refactoring (or checked manually refactoring opportunities) but find them not relevant, so they may end up not applying them. It is worth remarking that we have studied refactoring practice by looking at the actions actually performed by developers over the history of the considered software projects. Yet, we cannot exclude that developers still employed refactoring recommendation tools (*e.g.,* JDeodorant [12] or Aries [4]) to get suggestions on how to improve source code quality. However, we were interested in understanding how the actual application of refactoring changes from before to after the adoption of CI. As such, the investigation of whether refactoring recommendation tools have been used is out of the scope of our paper.

External threats to validity concern the generalizability of our results. First, we conducted this study based on a large dataset of 99,545 commits from 39 GitHub projects consistently active during our 48-month observation period. This filtering was required to fit our models and control for *time* variable as well as to avoid biasing our conclusions due to an inflation of zero values in our data. We also made restrictions, since we depend on RefactoringMiner, to Java projects. To our knowledge, current available refactoring detection tools are dedicated to Java language [38]. Moreover, we only-considered Travis CI, the most popular CI service on GitHub [19]. As a large fraction of projects on GitHub are small and not highly active [57], we plan, as a future work, to

[10]`https://jenkins.io/`
[11]`https://www.jetbrains.com/teamcity/`
[12]`http://easycis.aspone.cz/`

extend our study to other open-source and industrial projects.

7. Conclusion

We presented in this paper the *first empirical study* that investigates the possible impacts of continuous integration (CI), a quality-driven process, on changing the way developers practice refactoring. To analyze potential CI impacts, we (1) employed different heuristics estimating refactoring commits frequency, size and involved developers, (2) used Multiple Regression Analysis (MRA) to estimate CI impacts on refactoring practice while controlling for different confounding variables and (3) analyzed the change in refactoring tactics two years before and after the adoption of CI.

Based on data extracted from a sample of 39 GitHub projects deploying CI, our results revealed that the refactoring change size tends to decrease as recommended. However, the frequency and refactoring authors tend to drop during the two years following the CI adoption. These findings lend support to previous research efforts claiming the presence of barriers, related especially to lack of time and knowledge, preventing developers from adopting refactoring techniques/tools in CI context.

Our future work will include extending our study to other open-source and industrial projects from different programming languages and application domains. We also plan to conceive refactoring tools that can support CI developers in their quality enhancement efforts.

References

[1] Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Juergen Rilling. 2019. Which Commits Can Be CI Skipped? *IEEE Transactions on Software Engineering* (2019).

[2] Vahid Alizadeh, Mohamed Amine Ouali, Marouane Kessentini, and Meriem Chater. 2019. RefBot: intelligent software refactoring bot. In *34th*

IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 823–834.

[3] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. 2019. On the impact of refactoring on the relationship between quality attributes and design metrics. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–11.

[4] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, Rocco Oliveto, and Fabio Palomba. 2012. Supporting extract class refactoring in eclipse: The aries project. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 1419–1422.

[5] M. Beller, G. Gousios, and A. Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 447–450. `https://doi.org/10.1109/MSR.2017.24`

[6] João Helis Bernardo, Daniel Alencar da Costa, and Uirá Kulesza. 2018. Studying the impact of adopting continuous integration on the delivery time of pull requests. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 131–141.

[7] Lianping Chen and Muhammad Ali Babar. 2014. Towards an evidence-based understanding of emergence of architecture through continuous refactoring in agile software development. In *2014 IEEE/IFIP Conference on Software Architecture*. IEEE, 195–204.

[8] Patricia Cohen, Stephen G West, and Leona S Aiken. 2014. *Applied multiple regression/correlation analysis for the behavioral sciences*. Psychology Press.

[9] Paul Duvall, Stephen M. Matyas, and Andrew Glover. 2007. *Continuous*

Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series). Addison-Wesley Professional.

[10] Allen L Edwards. 1985. *Multiple regression and the analysis of variance and covariance.* WH Freeman/Times Books/Henry Holt & Co.

[11] Wagner Felidré, Leonardo Furtado, Daniel A da Costa, Bruno Cartaxo, and Gustavo Pinto. 2019. Continuous Integration Theater. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–10.

[12] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. 2011. JDeodorant: identification and application of extract class refactorings. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 1037–1039.

[13] Martin Fowler. 2006. Continuous Integration. `https://www.martinfowler.com/articles/continuousIntegration.html`,. Accessed: 2020-02-20.

[14] Martin Fowler. 2018. *Refactoring: improving the design of existing code.* Addison-Wesley Professional.

[15] Martin Fowler, Kent Beck, John Brant, William Opdyke, and don Roberts. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[16] Yash Gupta, Yusaira Khan, Keheliya Gallaba, and Shane McIntosh. 2017. The impact of the adoption of continuous integration on developer attraction and retention. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 491–494.

[17] Sarra Habchi, Romain Rouvoy, and Naouel Moha. 2019. On the Survival of Android Code Smells in the Wild.

[18] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in continuous integration: assurance, security, and flexibility. In *11th Joint Meeting on Foundations of Software Engineering*. ACM, 197–207.

[19] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-source Projects. In *31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 426–437. https://doi.org/10.1145/2970276.2970358

[20] Guido W Imbens and Thomas Lemieux. 2008. Regression discontinuity designs: A guide to practice. *Journal of econometrics* 142, 2 (2008), 615–635.

[21] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A field study of refactoring challenges and benefits. In *ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 50.

[22] Stefano Lambiase, Andrea Cupito, Fabiano Pecorelli, Andrea De Lucia, and Fabio Palomba. 2020. Just-In-Time Test Smell Detection and Refactoring: The DARTS Project. In *28th International Conference on Program Comprehension*. 441–445.

[23] Manny M Lehman. 1996. Laws of software evolution revisited. In *European Workshop on Software Process Technology*. Springer, 108–124.

[24] Yao Lu, Xinjun Mao, Zude Li, Yang Zhang, Tao Wang, and Gang Yin. 2018. Internal quality assurance for external contributions in GitHub: An empirical investigation. *Journal of Software: Evolution and Process* 30, 4 (2018), e1918.

[25] Raimund Moser, Alberto Sillitti, Pekka Abrahamsson, and Giancarlo Succi.

2006. Does refactoring improve reusability?. In *International Conference on Software Reuse*. Springer, 287–297.

[26] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. 2012. How we refactor, and how we know it. *IEEE Transactions on Software Engineering (TSE)* 38, 1 (2012), 5–18.

[27] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E Johnson, and Danny Dig. 2012. *Using Continuous Code Change Analysis to Understand the Practice of Refactoring*. Technical Report.

[28] Jevgenija Pantiuchina, Gabriele Bavota, Michele Tufano, and Denys Poshyvanyk. 2018. Towards just-in-time refactoring recommenders. In *IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. 312–3123.

[29] Akond Rahman, Amritanshu Agrawal, Rahul Krishna, and Alexander Sobran. 2018. Characterizing the influence of continuous integration: Empirical results from 250+ open source and proprietary projects. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics*. 8–14.

[30] Islem Saidani. 2020. Replication Package. Available at : `https://github.com/ci-ref/replication-package`.

[31] Gerald Schermann, Jürgen Cito, Philipp Leitner, and Harald C Gall. 2016. Towards quality gates in continuous delivery and deployment. In *24th international conference on program comprehension (ICPC)*. IEEE, 1–4.

[32] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors. In *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. 858–870.

[33] Danilo Silva and Marco Tulio Valente. 2017. RefDiff: detecting refactorings in version histories. In *14th International Conference on Mining Software Repositories*. 269–279.

[34] Ioannis G Stamelos and Panagiotis Sfetsos. 2007. *Agile software development quality assurance.* Igi Global.

[35] Igor Steinmacher, Tayana Conte, Marco Aurélio Gerosa, and David Redmiles. 2015. Social barriers faced by newcomers placing their first contribution in open source software projects. In *18th ACM conference on Computer supported cooperative work & social computing*. 1379–1392.

[36] Igor Steinmacher, Igor Wiese, Ana Paula Chaves, and Marco Aurélio Gerosa. 2013. Why do newcomers abandon open source software projects?. In *6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. 25–32.

[37] Gábor Szóke, Gábor Antal, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. 2014. Bulk fixing coding issues and its effects on software quality: Is it worth refactoring?. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 95–104.

[38] Liang Tan and Christoph Bockuisch. [n.d.]. A Survey of Refactoring Detection Tools. ([n. d.]).

[39] Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. 2013. A multidimensional empirical study on refactoring activity. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*. IBM Corp., 132–146.

[40] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 483–494. https://doi.org/10.1145/3180155.3180206

[41] Nikolaos Tsantalis, Matin Mansouri, Laleh M Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *40th International Conference on Software Engineering*. ACM, 483–494.

[42] Naoya Ujihara, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2017. c-JRefRec: Change-based identification of Move Method refactoring opportunities. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 482–486.

[43] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and Productivity Outcomes Relating to Continuous Integration in GitHub. In *10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 805–816. https://doi.org/10.1145/2786805.2786850

[44] Carmine Vassallo, Giovanni Grano, Fabio Palomba, Harald C Gall, and Alberto Bacchelli. 2019. A large-scale empirical exploration on refactoring activities in open source software projects. *Science of Computer Programming* 180 (2019), 1–15.

[45] Carmine Vassallo, Fabio Palomba, Alberto Bacchelli, and Harald C. Gall. 2018. Continuous Code Quality: Are We (Really) Doing That?. In *33rd ACM/IEEE International Conference on Automated Software Engineering*. 790–795.

[46] Carmine Vassallo, Fabio Palomba, and Harald C. Gall. 2018. Continuous Refactoring in CI: A Preliminary Study on the Perceived Advantages and Barriers. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. 564–568. https://doi.org/10.1109/ICSME.2018.00068

[47] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano

Panichella. 2017. A tale of CI build failures: An open source and a financial organization perspective. In *IEEE international conference on software maintenance and evolution (ICSME)*. 183–193.

[48] Yi Wang. 2009. What motivate software engineers to refactor source code? evidences from professional developers. In *2009 ieee international conference on software maintenance*. IEEE, 413–416.

[49] Fadi Wedyan, Dalal Alrmuny, and James M Bieman. 2009. The effectiveness of automated static analysis tools for fault detection and refactoring prediction. In *International Conference on Software Testing Verification and Validation*. 141–150.

[50] David Widder, Bogdan Vasilescu, Michael Hilton, and Christian Kästner. 2018. I'm leaving you, Travis: a continuous integration breakup story. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 165–169.

[51] Marvin Wyrich and Justus Bogner. 2019. Towards an autonomous bot for automatic source code refactoring. In *IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*. 24–28.

[52] Zhenchang Xing and Eleni Stroulia. 2005. UMLDiff: an algorithm for object-oriented design differencing. In *IEEE/ACM international Conference on Automated software engineering*. ACM, 54–65.

[53] Yue Yu, Bogdan Vasilescu, Huaimin Wang, Vladimir Filkov, and Premkumar Devanbu. 2016. Initial and eventual software quality relating to continuous integration in GitHub. *arXiv preprint arXiv:1606.00521* (2016).

[54] Yue Yu, Gang Yin, Tao Wang, Cheng Yang, and Huaimin Wang. 2016. Determinants of pull-based development in the context of continuous integration. *Science China Information Sciences* 59, 8 (2016), 080104.

[55] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald Gall, and Massimiliano Di Penta. 2020. An empirical charac-

terization of bad practices in continuous integration. *Empirical Software Engineering* 25, 2 (2020), 1095–1135.

[56] Chen Zhang, Bihuan Chen, Linlin Chen, Xin Peng, and Wenyun Zhao. 2019. A large-scale empirical study of compiler errors in continuous integration. In *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 176–187.

[57] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. 2017. The Impact of Continuous Integration on Other Software Development Practices: A Large-scale Empirical Study. In *32nd IEEE/ACM International Conference on Automated Software Engineering.* 60–71.