Continuous Code Quality: Are We (Really) Doing That?

Carmine Vassallo
University of Zurich
Zurich, Switzerland
vassallo@ifi.uzh.ch

Fabio Palomba
University of Zurich
Zurich, Switzerland
palomba@ifi.uzh.ch

Alberto Bacchelli
University of Zurich
Zurich, Switzerland
bacchelli@ifi.uzh.ch

Harald C. Gall
University of Zurich
Zurich, Switzerland
gall@ifi.uzh.ch

ABSTRACT

Continuous Integration (CI) is a software engineering practice where developers constantly integrate their changes to a project through an automated build process. The goal of CI is to provide developers with prompt feedback on several quality dimensions after each change. Indeed, previous studies provided empirical evidence on a positive association between properly following CI principles and source code quality. A core principle behind CI is *Continuous Code Quality* (also known as CCQ, which includes automated testing and automated code inspection) may appear simple and effective, yet we know little about its practical adoption. In this paper, we propose a preliminary empirical investigation aimed at understanding how rigorously practitioners follow CCQ. Our study reveals a strong dichotomy between theory and practice: developers do not perform continuous inspection but rather control for quality only at the end of a sprint and most of the times only on the release branch. Preprint [https://doi.org/10.5281/zenodo.1341036]. Data and Materials [http://doi.org/10.5281/zenodo.1341015].

CCS CONCEPTS

• **Software and its engineering** → *Maintaining software*;

KEYWORDS

Continuous Integration, Code Quality, Empirical Studies

ACM Reference Format:
Carmine Vassallo, Fabio Palomba, Alberto Bacchelli, and Harald C. Gall. 2018. Continuous Code Quality: Are We (Really) Doing That?. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3238147.3240729

1 INTRODUCTION

"Improving software quality and reducing risks" [8]. This is how Continuous Integration (CI) has been put forward by Duvall et al. [8] and is widely perceived by developers and students [22]. Concretely, CI is an agile software development process aimed at continuously integrating changes made by developers working on a shared repository; a build server that is used to build every commit, run all tests, and assess source code quality [15].

Duvall et al. [8] have proposed a set of principles that developers should methodically follow to adopt CI. For instance, CI users should build software as soon as a new change to the codebase is performed, instead of building software at certain scheduled times (e.g., nightly builds). A key principle of CI, as advocated by Duvall et al. [8], is *continuous inspection*, which includes running automated tests and performing static/dynamic analysis of the code at every build, as a way to ensure code quality. This aspect of CI is also known as *Continuous Code Quality* (CCQ) [28].

Previous work provided evidence on the potential of CI in achieving its stated goals. Vasilescu et al. [35] quantitatively explored the effect of introducing CI on the quality of the pull request process, finding that it improves the number of processed pull requests. Khohm et al. [17] studied whether and how shifting toward a shorter release workflow (i.e., monthly releases) had an effect on the software quality of Firefox, reporting significant benefits. Others found evidence of reduced time-to-market associated with CI [39] and the possibility to catch software defects earlier [14].

However, empirical knowledge is still lacking on the actual practice of CCQ: How strictly do practitioners adopt CCQ? What are the effects resulting from practitioners' approach to CCQ? To scientifically evaluate CCQ and its effects, as well as to help practitioners in their software quality efforts, one has to first understand and quantify current developers' practices. In fact, an updated empirical knowledge on CCQ is paramount both to focus future research on the most relevant aspects of CCQ and on current problems in CI adoption, as well as to effectively guide the design of tools and processes. To this aim, we conduct a large-scale analysis that involves a total of 148,734 builds and 5 years of the development change history of 119 Java projects mined by SonarCloud and TravisCI, two well-known providers of continuous code quality and continuous integration data, respectively. We study the adoption of continuous code quality by measuring metrics like the number of builds subject to quality checks and frequency of the measurements.

Our findings reveal that only 11% of the builds are subject to a code quality check and that practitioners do not apply CCQ, rather run monitoring tools just at the end of a sprint. Moreover, only 36% of branches are checked.

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3240729

2 BACKGROUND AND RELATED WORK

This section provides an overview of the principles behind continuous code quality as well as the related literature.

2.1 Continuous Code Quality

There are a few basic principles at the basis of continuous integration [8]. Besides maintaining a single source code repository, the idea behind CI is to automate the correct integration of code changes applied by developers as much as possible. This is normally obtained by having a dedicated build server responsible for taking all the new commits as input and automatically build, test, and deploy them. In addition, code quality assessment tools are used in order to control how much the performed change respects the qualitative standards of the organization. Thus, the principle of *continuous code quality* translates into having a development pipeline composed of a repository, a CI build server, and a CCQ Service. The developer commits a change to a repository (e.g., hosted on GitHub [12]), triggering a new build on the CI build server (e.g., TravisCI [31]). The server transfers the change to a different server (called *CCQ Service*) that is in charge of performing the quality analyses and reporting back the outcome to the CI build server. Based on its configuration, the CI build server decides on whether the build fails depending on the results of the CCQ service.

CI build server users can configure the build in a customized way, e.g., sending only specific builds or builds on specific branches to the CCQ service for inspection. This configuration allows users to depart from the continuous quality practice as prescribed [8] and to follow a different strategy. The decision to depart from the prescribed CCQ practice is at the basis of our work, which is focused on a deeper understanding of the actual CCQ practices.

2.2 Related Work

In the last years, researchers have proposed a growing number of studies targeting CI practices [2, 38, 39], also thanks to the increasing availability of publicly available CI data [3].

Hilton et al. [14] employed a mixed-method approach to study the use of CI in open-source projects. They first mined the change history of 34,544 systems, finding that CI is already adopted by the most popular projects and that the overall percentage of projects using CI is growing fast. In the second place, the researchers surveyed 442 developers on the perceived benefits of CI. The main perceived advantage is that CI helps projects release more often.

Hilton et al. [13] proposed a qualitative study targeting the barriers developers face when using CI. The study comprised two surveys with 574 industrial developers, with the main findings presenting the trade-offs between (i) speed and certainty, (ii) information access and security, and (iii) configuration options and usability. The authors motivated the need for new methods and tools able to find a compromise between those perspectives. The results discussed so far were also confirmed by Laukkanen et al. [21] and Kim et al. [18], who reported on industrial experiences when using CI.

Complementing the studies mentioned above, our investigation aims at understanding how rigorously developers adopt CCQ.

Other researchers investigated the use of automated static analysis tools (know as ASATs) in CI. Specifically, Zampetti et al. [40] observed that a low number of builds fail because of warnings raised by ASATs, while Vassallo et al. [37] reported that developers configured static analysis tools only at the beginning of a project. Our study further elaborates on how developers use ASATs in CI, by exploring how they use them in order to perform CCQ.

3 OVERVIEW OF THE RESEARCH METHODOLOGY

As Duvall et al. stated in previous work [8], the time between discovery and fix of code quality issues can be significantly reduced by continuously inspecting the code. Thus, the application of the *continuous inspection* principle is stated to be crucial for fulfilling the main advantage of CI, i.e., "improving software quality and reducing risks" [8].

The *goal* of the study is to quantify the gap (if any) between the *continuous inspection* principle (also known as *continuous code quality* [28]) and the actual practices applied by developers with the *purpose* of providing initial guidelines and tools for future research in the field of continuous integration. Thus, our investigation is structured around one research question: *how is CCQ applied to projects in CI?*.

The *perspective* is of researchers and practitioners interested in understanding whether code quality assessment is performed continuously in CI.

In order to answer our research question and guide future research on CCQ practice, we first need to construct a dataset containing projects developed through a CCQ pipeline. The *context* of our study consists of such a dataset, which includes 119 projects selected as reported in Section 4.3.

Then, we devise a set of CCQ metrics for assessing the actual CCQ adoption (described in Section 5.1) and measure them over the history of the projects in our dataset (Section 5.2).

4 CONTINUOUS CODE QUALITY DATA COLLECTION

To conduct our investigation, we need to study projects that not only use CI, but also: (i) adopt a CCQ pipeline, (ii) adopt a static analysis tool that stores the quality measurements performed over their history, and (iii) have CI-related events available, so that we can contextualize CCQ measurements in their evolution.

Since an already built dataset that fulfills our criteria is not available, we build our own. The definition of an ad-hoc data collection strategy is necessary because CCQ and CI events are stored on different servers and the alignment of the CCQ change history over the change history recording all the events occurred on the CI build server required the definition of heuristics to properly match the two sources. In the next sections, we describe the procedure we follow to build the dataset, which is composed of three main steps such as (i) collecting data from the CCQ server, (ii) collecting data from the CI build server, and (iii) aligning the change history coming from the two sources.

4.1 Collecting CCQ Data

SonarCloud[1] is a cloud service based on SonarQube [28] that continuously inspects code quality and detects bugs, vulnerabilities, and code smells. SonarQube is one of the most widely adopted code analysis tools in the context of CI [28]. SonarQube is a SonarSource product that is adopted by more than 85,000 organizations and that support more than 20 languages—including the most popular ones according to the TIOBE index [30]. SonarQube provides developers with its own rules and incorporates rules of other popular static and dynamic code analysis tools [28]. As an example, SonarQube runs all the most popular code analysis tools (i.e., CheckStyle, PMD, Findbugs, Cobertura) by default on Java projects. Thus, the relevance of SonarQube in the context of CI motivates the decision to focus on systems using SonarCloud as CCQ service.

Overall, 14,152 projects are actively using SonarCloud, even though some of them are private and, thus, not accessible. We query SonarCloud using the available web APIs [27] and extract the list of all the open source projects that use the free analysis service, reaching 1,772 candidate systems[2].

4.2 Collecting CI Data

Starting from the initial population of 1,772 candidate systems, we keep projects that use TravisCI as build server [31], as this ensures that the project actually adopts a CCQ practice. We select TravisCI as it provides the entire build history, as opposed to other build servers (e.g., Jenkins) where only the recent builds are typically stored [35].

Selecting projects using TravisCI as CI server and SonarCloud as CCQ service is not trivial. While TravisCI provides a direct and easy integration with SonarCloud[3], there is no explicit link between those two services, meaning that one cannot directly infer which projects use both services at the same time. Thus, we need to create such a link. Among the information available on SonarCloud, the projects report the URL referring to the source code repository; this URL provides us with an exploitable solution to identify the desired systems. In particular, TravisCI is used to build projects hosted on GitHub: therefore, we first consider all the projects available on SonarCloud that expose a GitHub URL. This step reduces the number of candidate projects to 439 (i.e., 25% of all SonarCloud systems). Subsequently, using the GitHub URL we query the TravisCI APIs [32] and check if a certain URL is present on the platform: 390 projects match the selection criteria, i.e., SonarCloud systems that are on TravisCI. As a final step, we remove projects having less than 20 CCQ checks over their history[4]. This filter is needed to avoid the analysis of projects that do not really integrate a CCQ service in their pipeline; in other words, we only consider projects that *actively* apply CCQ. At the end of this process, our dataset comprises 119 projects.

4.3 Overlaying CCQ and CI Information

Once the explicit link between SonarCloud and TravisCI is available, the final step of the data collection process is to overlay the separate change history information available in two sources. Also, in this case, there is no explicit way to link a data point available SonarCloud to one on TravisCI. We solve this as in the following. For each of the 148734 builds available on TravisCI we first collect (i) `build id`, (ii) `triggering commit` (i.e., commit message and id), (iii) `build status` (i.e., failed, errored, passed), (iv) `starting date`, and (v) `ending date`. Then, we use the `starting date` parameter of the build to identify the corresponding data point on SonarCloud.

Specifically, let $b_i \in T_i$ be a build done on the branch br in the CI history T_i of the project i available on TravisCI, and let $m_{ik} \in S_{ik}$ be a measurement of a certain metric k for project i on the branch br in the CCQ history H_{ik} available on SonarCloud, we considered m_{ik} to be the measurement corresponding to b_i if the following relation held:

$$date(m_{ik}) \geq startingDate(b_i) \wedge date(m_{ik}) \leq startingDate(b_{i+1})$$

In other words, for each of the 119 considered projects, we compute the time interval in which two subsequent builds (i.e., b_i and b_{i+1}) are performed on TravisCI and assign a quality measurement to the build b_i if it was started within that time window. For each considered project, the final result is an *overlaid change history*, which contains information about the measured metric(s) and value(s), for each measured build (i.e., a build subject to a measurement on SonarCloud).

5 CONTINUOUS CODE QUALITY IN PRACTICE

In this section, we discuss how continuous code quality is applied in the selected projects. Specifically, we first present the CCQ metrics that we conceive to automatically assess the CCQ practice. Then, we show how our projects perform against the CCQ metrics over their development's history.

5.1 Definition of CCQ Metrics

Our study aims at assessing the practical use of CCQ. Based on the constructed *overlaid change history* of the 119 subject projects, we devise four indicators for measuring the actual CCQ usage:

CQCR – Code Quality Checking Rate: Number of builds subject to a code quality check divided by the total number of builds.

EFC – Elapsed Frame between Checks: Average number of builds between two builds subject to a code quality check.

ETC – Elapsed Time between Checks: Average number of days between two builds subject to a code quality check.

CB – Percentage of Checked Branches: Number of branches containing at least one build subject to a code quality check divided by the number of total branches scheduled for build.

We design these CCQ usage indicators (based on the guidelines by Duvall et al. [9]) to understand how well CCQ is performed from different perspectives. CQCR is the basic metric that reveals the fraction of builds that are qualitatively measured during the history of a project, thus giving a view on the extent to which developers use to check builds in their projects. EFC and ETC measure the frequency of the quality checks in the considered projects, in terms

[1] https://about.sonarcloud.io
[2] The complete list is available in our online appendix [36].
[3] https://docs.travis-ci.com/user/sonarcloud/
[4] The threshold of 20 is fixed in a similar way as done in previous work [6, 16, 24].

Table 1: CCQ usage indicators applied to our projects.

Project Set			CCQ Usage Indicators			
Feature	Level	# projects	CQCR	EFC	ETC	CB
Age	Low	30	0.14	7.72	14.49	0.62
	Medium	58	0.17	10.86	18.33	0.25
	High	30	0.06	39.08	16.49	0.33
Contribution	Low	27	0.14	8.99	16.67	0.39
	Medium	61	0.12	9.74	16.69	0.41
	High	30	0.05	36.76	17.33	0.22
Popularity	Low	30	0.14	9.42	15.63	0.49
	Medium	58	0.12	10.61	15.47	0.33
	High	30	0.06	37.34	20.20	0.27
Overall			0.11	18.30	16.91	0.36

of the average number of builds and days, respectively, that are waited before performing a new quality check. CB indicates if there are branches that are not checked at all: in this case, we want to measure whether there are branches that are more prone to be subject of qualitative checks.

5.2 On the Current Application of CCQ

Table 1 reports the results of our study aimed at investigating how CCQ is applied in practice. The table reports the overall values (row "Overall") of each considered metric, i.e., *Code Quality Checking Rate (CQCR)*, *Elapsed Frame between Checks (EFC)*, *Elapsed Time between Checks (ETC)*, and *Percentage of Checked Branches (CB)*. Moreover, with the aim of deeper understanding whether the characteristics of the projects influence our observations, we also report the overall metric values when splitting the systems by age, contribution, and popularity.

We exploit the GitHub APIs [12] to identify (i) the number of performed commits, (ii) the number of contributors, and (iii) the numbers of stars of a certain repository, respectively. For each considered perspective (i.e., age, contribution, and popularity), we split projects into three different subsets, i.e., *low*, *medium*, and *high*. Specifically, we calculate the first (Q_1) and the third (Q_3) quartile of the distribution representing the number of commits, contributors, and stars of the subject systems. Then, we classify them into the following categories: (i) *low* if they have a number of commits/contributors/stars n lower than Q_1; (ii) *medium* if $Q_1 \leq n < Q_3$, and (iii) *high* if n is higher than Q_3. As shown in Table 1 (column "# projects"), we inadvertently achieved a good balance among the different subsets in terms of the number of contained projects.

Looking at the results, we can first observe that, overall, only 11% of the builds are qualitatively checked (CQCR value). This is a quite surprising result, because it clearly indicates that projects are **not** continuously inspected. In the lights of this finding, we can claim that the *continuous inspection* principle is generally not respected in practice.

When considering projects split by categories, i.e., *low*, *medium*, and *high* for age, contribution, and popularity, we can perceive a trend in the results. Young and medium-age projects exhibit higher values for CQCR with respect to the more mature projects, yet still have a pretty low percentage of monitored builds (14% and 17%, respectively). This finding seems to suggest that the application of CCQ becomes even harder when increasing the number of commits, and consequently the number of builds of a software project. We find that only 6% of the builds pass for a quality check in long-lived systems, while the percentage is 5% in case of an high number of contributors. This result triangulates the findings by Hilton et al. [14], revealing that developers are still not very familiar with all the CI principles and tend to not apply them properly. At the same time, it seems that community-related factors play a role in the application of CCQ. Indeed, our findings suggest that communities with a large number of contributors are less prone to apply CCQ: this is in line with previous work that showed how large communities generally have more coordination/communication issues, possibly resulting in technical pitfalls [5, 11, 29].

The most popular projects are generally more likely to use CI [14], however—according to our results—they do not apply CCQ properly. This is visible in Table 1, where we observe that only 6% of the builds of popular projects are qualitatively monitored. Conversely, low and medium-popular systems exhibit a higher number of measured builds.

> ***Finding 1.*** *The projects using CI do not continuously inspect the source code. Moreover, the percentage of qualitatively monitored builds is lower for systems with large numbers of commits and contributors.*

Elapsed Frame between Checks (EFC) measures the average number of builds between two builds subject to a code quality check on the same branch. The overall result for EFC strengthens our initial findings on the lack of CCQ. On the average, developers perform a code quality check every 18 builds. This number still increases where taking into account the size of the projects. Indeed, systems with a high number of commits and contributors have an EFC score of 39 and 37, respectively. It is important to highlight that such projects have a higher number of builds with respect to small projects, and therefore might benefit more of a continuous check of code quality.

Looking at and *Elapsed Time between Checks* (ETC), we can confirm what we observe for the elapsed time between quality checks: developers do not perform a continuous code quality assessment, but rather they monitor the quality at time intervals of 17 days. This number is very close to the usual duration of a SCRUM Sprint [1], which is often used in the CI context [20]: thus, our findings suggest that likely the current practice merely consists of checking code quality at the end of a sprint. This observation holds when splitting projects based on their characteristics, as we confirm that quality checks are performed at fixed intervals.

> ***Finding 2.*** *Developers perform a code quality inspection after several builds (on average every 18 builds) and, most likely, at the end of a sprint.*

As the last indicator, we compute the percentage of *Checked Branches* (CB). Table 1 shows a similar trend as for the other CCQ usage indicators. Also in this case, the higher the number of commits and contributors, the lower the percentage of branches that are subject to a quality check. This result confirms the possible role of community-related factors, as large communities tend to be more reluctant to apply CCQ.

Overall, only 36% of branches are checked, meaning that most of them are developed without a formal quality control.

Finding 3. *A low percentage of branches follow CCQ.*

6 DISCUSSION AND FUTURE WORK

Our results highlight a number of points to be further discussed, and in particular:

- **CCQ Is not Applied in Practice.** A clear result of our study demonstrates a poor usage of continuous code quality, and that indeed only a very low number of builds (11%) are qualitatively monitored. This finding opens up a number of observations. In the first place, the low use of CCQ may be due to a general biased perception that developers have with respect to source code quality [4, 25]: code quality is not the top-priority for developers [10], who prefer not to improve the existing code for different reasons, including time pressure or laziness [34]. Most of the time developers and product managers do not consider a quality decrement enough to fail the build process, or they do not know how to properly set up quality gates [26]. Besides this, our study somehow confirms the findings reported by Hilton et al. [13], highlighting once again that developers face several barriers when adopting CI principles.
- **The Relevance of a Development Community.** A key finding in our study reports that the size of a project plays a role in the adoption of continuous code quality. While projects having few developers perform a (slightly) higher percentage of code quality checks, systems with a larger community face more difficulties. This can be explained by the presence of community-related factors that might preclude an effective management of the development activities. Indeed, wrong communication and coordination within software communities have been not only largely associated to the emergence of socio-technical issues [7, 11, 23], but also related to continuous integration aspects. In particular, Kwan et al. [19] reported a strong negative impact of socio-technical congruence, i.e., a measure indicating the alignment between technical dependencies work relations among software developers, on build success. Our findings confirm the importance of studying such factors and how they influence technical aspects of software systems more deeply.
- **On the Size of Change History.** According to our results, projects having a longer change history are less likely to apply CCQ. This may suggest that a possible co-factor influencing the lack of continuous code quality control falls in the difficulty of developers to switch toward such new continuous monitoring in case the project is already mature.

Our initial findings pave the way to further study that we plan to conduct in future work:

(1) **On the Value of Continuous Code Quality.** Despite previous work in the area of agile processes [17], there is still a lack of study empirically assessing the benefits deriving from the actual practice of code quality assessment in CI. We build a dataset of projects using both CI Server and CCQ Service (as explained in Section 2). Thus, compared to previous work [35, 40] we are able to analyze the decisions of developers (i.e., whether perform code quality or not) and the obtained measurements without rerunning the analysis on projects' snapshots that might cause several threats, such as the unavailability of the configuration file or the impossibility to build a snapshot [33]. As future work, we plan to measure the effectiveness of the actual CCQ practice in maintaining software quality.

(2) **Key Scenarios in Continuous Code Quality.** Given the fact that code quality is not continuously assessed in CI, we are interested in determining the circumstances (e.g., development tasks) where the use of CCQ should be particularly encouraged, as they can lead to significantly decrease the quality of source code. It might be that CCQ is particularly effective in certain scenarios compared to others.

(3) **Code Quality Recommendation in CI.** Slow builds are serious barriers faced by developers using CI [13]. Automated testing and code quality assurance tasks and are possible causes in slowing down builds. Code quality tasks are usually postponed and scheduled in *nightly builds*, thus preventing CCQ to be performed. We aim at finding a good trade-off between scheduling code quality tasks at every new change and slowing down the build. Our vision is to predict which quality measurements perform before triggering a new build. Given the actual build context described in terms of several features (e.g., checked-out branch, type of development task, etc.), a recommender will automatically schedule a new code quality task enabling the proper warnings.

7 THREATS TO VALIDITY

This section discusses possible threats that might have affected the validity of our observations.

We mined information from different sources and combined them using heuristics that were needed because of the lack of an explicit link between them. To infer projects using both SonarCloud and TravisCI we used their Github URL—exposed on the first platform—as a means for understanding whether they also use TravisCI as build server. This linking process can be considered safe, as the Github URL of a project is unique and, thus, there cannot be cases where the history of a project on SonarCloud was overlaid with the one of another project on TravisCI. As for the overlay of the change history information of the two platforms, we exploited the build and measurement dates to understand to which build a certain measurement referred to. Also, in this case, the linking procedure cannot produce false positives because there are not cases in which different builds might have been performed between the dates considered.

As for the generalizability of the results, we conducted this study on a large dataset composed of 119 projects. We also made some precautions to take into account only projects that actively adopt CI and CCQ. We limited our study to Java projects since some of the exploited platforms (e.g., SonarCloud) mainly contained information on this type of systems. Replications aimed at targeting projects written in different programming languages as well as industrial ones would be desirable.

8 CONCLUSION

In this paper, we analyzed the current practice of Continuous Code Quality (CCQ). Our findings showed that the theoretical principles reported by Duvall et al. [8] are not followed in practice. We found that only 11% of the builds are subject to a quality control. More importantly, the current CCQ practice merely consists of checking code quality at the end of a sprint, thus basically ignoring the CCQ principle.

Based on the dataset that we built overlaying change history information coming from SonarCloud and TravisCI, we plan to investigate the impact of the current CCQ practice on the software quality and the circumstances where developers are particularly encouraged to check code quality more frequently. Our future research agenda includes also the definition of techniques for assisting developers during continuous monitoring of code quality.

ACKNOWLEDGMENTS

Vassallo and Gall acknowledge the support of the Swiss National Science Foundation for the project "SURFMobileAppsData" (SNF Project No. 200021-166275). Bacchelli and Palomba also gratefully acknowledge the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529.

REFERENCES

[1] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. 2001. The agile manifesto.

[2] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub. In *International Conference on Mining Software Repositories*.

[3] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *Proceedings of the 14th working conference on mining software repositories*.

[4] Nigel Bevan. 1999. Quality in use: Meeting user needs for quality. *Journal of systems and software* 49, 1 (1999), 89–96.

[5] Andrea Bonaccorsi and Cristina Rossi Lamastra. 2004. Altruistic individuals, selfish firms? The structure of motivation in Open Source software. (2004).

[6] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social coding in GitHub: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*. ACM, 1277–1286.

[7] Nicolas Ducheneaut. 2005. Socialization in an open source software community: A socio-technical analysis. *Computer Supported Cooperative Work (CSCW)* 14, 4 (2005), 323–368.

[8] Paul Duvall, Stephen M. Matyas, and Andrew Glover. 2007. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley.

[9] Paul M. Duvall. 2010. Continuous integration. Patterns and Antipatterns. *DZone refcard #84* (2010). http://bit.ly/l8rfVS

[10] Neil A Ernst and John Mylopoulos. 2010. On the perception of software quality requirements during the project lifecycle. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*. Springer, 143–157.

[11] Simon Gibbs, Eduardo Casais, Oscar Nierstrasz, Xavier Pintado, and Dennis Tsichritzis. 1990. Class management for software communities. *Commun. ACM* 33, 9 (1990), 90–103.

[12] GitHub. 2018. GitHub APIs. https://developer.github.com/v3/. Online; accessed 24 July 2018.

[13] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-Offs in Continuous Integration: Assurance, Security, and Flexibility. In *Proceedings of the 25th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2017*. To Appear.

[14] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 426–437.

[15] Jez Humble and David Farley. 2010. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional.

[16] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2016. An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering* 21, 5 (2016), 2035–2071.

[17] Foutse Khomh, Bram Adams, Tejinder Dhaliwal, and Ying Zou. 2015. Understanding the impact of rapid releases on software quality - The case of firefox. *Empirical Software Engineering* 20, 2 (2015), 336–373.

[18] Seojin Kim, Sungjin Park, Jeonghyun Yun, and Younghoo Lee. 2008. Automated continuous integration of component-based software: An industrial experience. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 423–426.

[19] Irwin Kwan, Adrian Schroter, and Daniela Damian. 2011. Does socio-technical congruence have an effect on software build success? a study of coordination in a software project. *IEEE Transactions on Software Engineering* 37, 3 (2011), 307–324.

[20] Lina Lagerberg, Tor Skude, Par Emanuelsson, Kristian Sandahl, and Daniel Stahl. 2013. The impact of agile principles and practices on large-scale software development projects: A multiple-case study of two projects at ericsson. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*. IEEE, 348–356.

[21] E. Laukkanen, M. Paasivaara, and T. Arvonen. 2015. Stakeholder Perceptions of the Adoption of Continuous Integration – A Case Study. In *Agile Conference (AGILE), 2015*. 11–20.

[22] Eero Laukkanen, Maria Paasivaara, and Teemu Arvonen. 2015. Stakeholder Perceptions of the Adoption of Continuous Integration–A Case Study. In *Agile Conference (AGILE), 2015*. IEEE, 11–20.

[23] Mikael Lindvall, Dirk Muthig, Aldo Dagnino, Christina Wallin, Michael Stupperich, David Kiefer, John May, and Tuomo Kahkonen. 2004. Agile software development in large organizations. *Computer* 37, 12 (2004), 26–34.

[24] Jennifer Marlow, Laura Dabbish, and Jim Herbsleb. 2013. Impression formation in online peer production: activity traces and personal profiles in github. In *Proceedings of the 2013 conference on Computer supported cooperative work*. ACM, 117–128.

[25] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. 2014. Do they really smell bad? a study on developers' perception of bad code smells. In *Software maintenance and evolution (ICSME), 2014 IEEE international conference on*. IEEE, 101–110.

[26] Gerald Schermann, Jürgen Cito, Philipp Leitner, and Harald C. Gall. 2016. Towards quality gates in continuous delivery and deployment. In *International Conference on Program Comprehension*.

[27] SonarCloud. 2018. SonarCloud Web APIs. https://sonarcloud.io/web_api. Online; accessed 24 July 2018.

[28] SonarSource S.A. 2018. SonarQube. https://www.sonarqube.org.

[29] Damian A Tamburri, Rick Kazman, and Hamed Fahimi. 2016. The Architect's Role in Community Shepherding. *IEEE Software* 33, 6 (2016), 70–79.

[30] Tiobe. 2018. Tiobe Ranking. https://www.tiobe.com/tiobe-index/. Online; accessed 24 July 2018.

[31] TravisCI. 2018. Travis CI. https://travis-ci.org. Online; accessed 24 July 2018.

[32] TravisCI. 2018. Travis CI APIs. https://developer.travis-ci.com. Online; accessed 24 July 2018.

[33] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2017. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process* 29, 4 (2017).

[34] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2017. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering* 43, 11 (2017), 1063–1088.

[35] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar T. Devanbu, and Vladimir Filkov. 2015. Quality and productivity outcomes relating to continuous integration in GitHub. In *ESEC/SIGSOFT FSE*. ACM, 805–816.

[36] Carmine Vassallo, Fabio Palomba, Alberto Bacchelli, and Harald C. Gall. 2018. Continuous Code Quality: Are We (Really) Doing That? Online Appendix. https://doi.org/10.5281/zenodo.1341015

[37] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C. Gall. 2018. Context is king: The developer perspective on the usage of static analysis tools. In *SANER*. IEEE Computer Society, 38–49.

[38] 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 *ICSME*. IEEE Computer Society, 183–193.

[39] Carmine Vassallo, Fiorella Zampetti, Daniele Romano, Moritz Beller, Annibale Panichella, Massimiliano Di Penta, and Andy Zaidman. 2016. Continuous Delivery Practices in a Large Financial Organization. In *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 41–50.

[40] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. 2017. How open source projects use static code analysis tools in continuous integration pipelines. In *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 334–344.