IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 11 September 2023, accepted 7 October 2023, date of publication 13 October 2023,
date of current version 19 October 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3324536

RESEARCH ARTICLE

Using Continuous Integration Techniques in Open Source Projects–An Exploratory Study

MICHAL R. WRÓBEL, JAROSLAW SZYMUKOWICZ, AND PAWEL WEICHBROTH, (Member, IEEE)
Faculty of Electronics, Telecommunications and Informatics, Gdańsk University of Technology, 80-233 Gdańsk, Poland
Corresponding author: Michal R. Wróbel (michal.wrobel@pg.edu.pl)

ABSTRACT For a growing number of software projects, continuous integration (CI) techniques are becoming an essential part of the process. However, the maturity of their adoption in open source projects varies. In this paper, we present an empirical study on GitHub repositories to explore the use of continuous integration techniques in open source projects. Following the Goal-Question-Metric (GQM) approach, 3 research questions and 7 metrics were defined for such a goal. We mined 10 repositories of open source projects with 101,149 pull requests, 399,671 commits from 20,432 developers. This was followed by exploratory data analysis for each metric. In summary, our results indicate that (RQ1) most failed CI builds required a small change in the pull request to fix the code; (RQ2) CI builds of smaller pull requests are more likely to succeed than larger ones; (RQ3) there was no correlation found between developer experience in committing to the project and the success rate of CI builds. Most of the projects studied have not yet developed a mature approach to using continuous integration techniques. In these cases, developers do not thoroughly test code before submitting pull requests. Furthermore, the results of the study confirmed that developers tend to submit pull requests with small amounts of new or modified code.

INDEX TERMS Continuous integration, mining software repositories, open source projects.

I. INTRODUCTION

Undeniably, nowadays software systems support activities in almost any aspects of the human life. Despite the ubiquity of many applications and the high rate of new applications, the software development process is a complex one. It requires the involvement of many stakeholders [1], including customers, business analysts, developers, managers, testers and administrators, among others [2]. Moreover, the cooperation of many parties, a considerable challenge is meeting software development standards that have increased over time [3]. These standards include application security, reliability and development time, just to name a few.

Continuous practices [4], such as continuous integration (CI) [5] and continuous delivery (CD) [6], are now an indispensable part of most software projects [7]. They focus on the automatic verification of changes made to the software source code [8]. Such verification is essential in the development of software, which is inherently complex, with individual components dependent on each other. In addition, it is necessary to verify that the changes introduced do not violate already existing functionality [9]. There are teams where such verification is still done manually by so-called manual testers [10].

However, this approach is increasingly being supported, or even entirely replaced by automated tests run as part of continuous integration [11]. In addition, such things as code style verification [12], i.e. whether the code is written according to good practices and is consistent with the rest of the source files, is now also automatically verified [13]. Many projects have a customized environment where the appropriate style in which to write code is verified or even enforced. Such plug-ins are commonly used in integrated development environments (IDE), so that in real time the programmer can see suggestions on code style. Nowadays, it is a common standard to use these programs for continuous

The associate editor coordinating the review of this manuscript and approving it for publication was Giuseppe Destefanis.

integration as well [14], and a positive code style inspection is considered a necessary condition to let changes through [15].

Building a program after merging changes is also often a complex task [16], considering the multi-platform nature of the modern applications, as well as the use of multiple components such as various libraries [17]. The task of building a finished program from code is time-consuming when performed manually, but at the same time is well-suited to be automated [18], occasionally requiring updating the build script [19]. Automating the application building, and verifying it once the changes have been incorporated is called continuous delivery and is also now a common practice [20]. It usually occurs in projects in tandem with continuous integration and is popular in both commercial [21] and open source projects [22].

The purpose of this article is to explore the use of continuous integration techniques in open source projects. Much of the work to date has been devoted to the good and bad practices of using continuous integration techniques and how to optimise their configuration, which is described in more detail in Section II. The goal of the paper is to gain insight into how developers use CI in practice. To this end, we have posed three research questions, largely inspired by observations we have made during our involvement in projects using continuous integration techniques, which will allow us to verify our intuition about the use of these techniques by developers.

- RQ1. How much effort was required to fix the code after CI failed build?
- RQ2. How does the amount of changes introduced in pull request affect the success of the build?
- RQ3. Does the experience of the contributor affect the success of CI builds?

For this purpose, data was collected from 10 repositories of open source projects from the GitHub platform, including information on commits and results of continuous integration build runs. On the basis of the collected data, an analysis was conducted to answer the research questions posed.

The rest of this paper is organized as follows: Section II presents the related works and Section III our research design. The results of the analysis are presented in Section IV, followed by a discussion in Section V, and finally a conclusion in Section VI.

II. RELATED WORK

Numerous papers on CI/CD have been published in the literature up to today. Hence, we selected and discussed the most relevant ones, taking into account the scope and domain of the topic.

Santos et al. evaluated the impact of five CI sub-practices with regard to the productivity and quality of GitHub open-source projects [23]. As the method, regression models were used to analyze whether projects upholding the CI sub-practices are more productive and/or produce fewer bugs. The findings from this study shows that there is a positive correlation between the *Commit Activity* and *Build Activity* and the increase in the number of merged pull requests.

Silva and Bezerra classified CI bad practices, based on the criteria of their frequency occurrence and the severity level [24]. Besides, authors argue that despite using continuous integration within the industrial software projects the numerous of errors were recognized, yet some quality improvements were also acknowledged.

A similar goal was the aim of Felidré et al.'s study, where unhealthy continuous integration practices were investigated [16]. As a result, they found that almost 60% of projects had infrequent commits, the average code coverage was quite high, and the vast majority of projects had long unfixed builds.

The hypothesis, stating that automated reporting approach supports early identification and prevention of anti-patterns and decay in continuous integration, was investigated by Carmine et al. through the study on the 18,474 build logs from the 36 selected Java projects [25]. In total, 8520 incidents were detected (including 3823 high-severity warnings), and later evaluated by group of 13 developers, who eventually confirmed the relevance of delivered information with regard to the four anti-patterns, namely: Slow Build, Skip Failed Tests, Late Merging, Broken Release Branch. In conclusion, the CI anti-patterns early detection are significantly supported by deploying automated reporting tools.

Another important problem addreessed by Saidani et al. [26], concerned the impact of continuous integration on changing the way how software developers practice refactoring, understood in terms of frequency, size and involved developers. In conclusion, the authors argue that the continuous integration adoption is associated with a cut in the refactoring size, while refactoring frequency along with the number of developers, responsible for code refactoring decreased after the shift to CI.

Lima and Vergilio performed a systematic mapping study on test case prioritization in continuous integration environments, identifying their main characteristics and their evaluation aspects [27]. The results show that the great majority of studies relied on the history-based approach which takes advantage of failure and test execution history. The evaluation is based on the comparison which makes use of measures such as time and number (percentage) of faults detected.

Since continuous integration is claimed to be an expensive practice, over the years the researchers have introduced a numerous of approaches that aim to reduce the workload. However, in same cases the wrong decisions lead to skipping builds which are undesirable to be skipped. In this context, Jin and Servant [28] put forward the following research question: which builds are really safe to skip? Besides the qualitative and comprehensive topic analysis, the authors introduced a novel approach, termed as *PreciseBuildSkip*, which maximizes build failure observation and reduces the cost of CI through the strategy of build selection.

Ghaleb et al. empirically explored characteristics of CI builds which could have been related with the long duration of CI builds, performed on over 104k CI builds from 67 projects hosted on the GitHub [29]. The results show that, along with commonly known factors, including project size, team size, build configuration size, and test density, there are other affecting long build durations, namely: rerunning failed commands multiple times, as well as triggering builds on weekdays or at daytime. On the other hand, the authors argue that builds may run faster if their configurations are set up (i) to cache content that does not change frequently, or (ii) to finish as soon as all the required jobs complete. Thus, using an appropriate CI build configurations aid their proper maintenance, and in particular reduce long build durations.

A different approach to identifying the factors influencing the success of a CI build was taken by Barrak et al. They developed and trained a nine-dimensional random forest model, which confirmed that build and author history and code complexity were the most important predictors of build failures [30].

The empirical study of Golzadeh et al. shows the bigger picture of continuous integration, covering the period of nine years and collected over 91k GitHub repositories of active npm packages having used at least one CI service [31]. In summary, in 2022 the rise and the dominance of GitHub Actions was observed in less than 18 months time. On the other hand, the decrease of Travis usage occurred, likely due to a combination of policy changes and migrations to GitHub Actions.

Kinsman et al. investigated how developers perceive and use GitHub and how activity indicators (the number of pull requests merged and nonmerged, number of comments, the time to close pull requests, and number of commits) change after their adoption [32]. While only the minority of repositories have adopted GitHub Actions, then the use of GitHub Actions increased the number of monthly rejected pull requests and decreased the monthly number of commits on merged pull requests.

Saidani et al. introduced an automated approach to downsize of CI build time, as well as to deliver support tool to developers by predicting the CI build outcome [33]. More specifically, the presented continuous integration build failure prediction model was based on Multi-Objective Genetic Programming (MOGP). Designed with the aim of finding the best combination of CI built features and their appropriate threshold values, the model innovatively utilized two conflicting objective functions to deal with both failed and passed builds.

Laukkanen et al. conducted a systematic literature review in order to identify issues encountered during continuous delivery adoption, along with their causes and possible solutions [34]. In total, 40 problems, 28 casual relationships, along with 29 solutions with regard to adoption of continuous delivery were identified. The most frequent problems were related to testing and integration, while the most critical concerned testing and system design. Eventually, solutions were thematically synthesized into six domains, namely: system design, integration, testing, release, human and organizational, and resource.

Another empirical study, performed by Wang et al. on the 149 open-source projects [35], aimed to observe the effect of test automation maturity on product quality, test automation effort, and release cycle, assembled in the context of continuous integration. The results show that higher levels of test automation maturity have had a positive impact on the product quality, as well as decreasing its release cycles. Moreover, authors claim that there is a lack of increased test automation effort due to higher levels of test automation maturity and product quality. In other words, if one applies best CI practice to mature test automation activities, the outcomes are to be expected are twofold: the product quality improvement and the release cycle acceleration.

Furthermore, since its introduction, continuous integration has impacted software development industry by creating new avenues for software quality evaluation. More specifically, accordingly to Yu et al. [36], CI environments can be used as valuable information source, leading to the improvement of non-functional requirements testing in terms of its efficiency and effectiveness.

In summary, we found well-written case studies and systematic reviews focusing on continuous integration practices with the aim of evaluating and summarising the results. However, few studies have addressed the question of how much effort is required to implement the necessary changes after a CI build failure. Given its operational and economic importance, there has been a curious lack of interest in uncovering CI practices in terms of source code maintenance and effectiveness. This study aims to fill this gap by conducting a comprehensive survey of open source projects hosted on the GitHub service.

III. RESEARCH DESIGN

A Goal-Question-Metric (GQM) approach was chosen to answer the research questions and thus achieve the defined goal. The GQM method was introduced to measure goal-oriented quality [37]. Over time, however, this approach has been adopted to structure research in many areas of software engineering, and has been widely used in other studies with similar settings (e.g. [38], [39], [40], [41]).

By design, the GQM provides a well-defined framework for setting research objectives, analyzing collected data, and deriving results. It was used in the study to bring clarity and structure to the application of CI in open source projects. By setting a research goal, formulating relevant questions, and defining specific metrics, it ensures objectivity in the data analysis and interpretation [42].

A. METRICS

Three questions were posed to explore the use of continuous integration techniques in open-source projects and identify good practices in their use. Further, up to three metrics were defined for each question:

Q1. How much effort was required to fix the code after CI failed build?

- **Q1M1**: Extent of changes required to improve the code after CI failed build.
- **Q1M2**: Time to deliver a commit fixing CI failed build.
- **Q1M3**: Number of changes in files of each type after CI build failure.

Q2. How does the number of changes introduced in the pull request affect the success of the build?

- **Q2M1**: Dependency of build result on a number of lines changed in source files.
- **Q2M2**: Dependency of build result on a number of lines changed in test files.
- **Q2M3**: Number of commits in pull request.

Q3. Does the experience of the contributor affect the success of CI builds?

- **Q3M1**: Number of CI failed builds depending on the experience of the contributor.

These metrics mainly concern three aspects of source code maintenance, namely: commits, pull requests and continuous integration build runs. In version control systems, a commit refers to a certain point in a project's source code history tree. This is a way for developers to indicate that changes they have made should be retained in the repository. When developers decide that the changes they have implemented are mature enough and tested enough to be merged into the main branch of the project, they create a so-called pull request. In this way, all modifications made, which may consist of one or more commits, can undergo a review process by peers and be evaluated by the continuous integration tool. During the CI process, an application is built from source code and then all implemented tests are executed. Such build run can either succeed or fail. Once a pull request has been submitted, developers can continue to submit new commits, e.g. to improve the code in line with the reviewers' comments, or to fix bugs that were revealed during the build run that ended in failure.

B. REPOSITORIES

The subjects of the study were open source projects with both code repositories and the continuous integration infrastructure entirely using the GitHub service infrastructure. This service offers continuous practice management as one element of repository control. For open source projects, the GitHub service offers free use of infrastructure within certain limits. The data of such projects is publicly available via the REST API. This data includes pull requests, commits and CI build results, among others.

Ten repositories were selected for the study, shown in Table 1. The choice of repositories was guided by the intention to examine large and very active projects as well as smaller ones. In this respect, the number of pull requests was taken into account. The largest repository analysed, *flutter*, has more than 35,000 pull requests, and the smallest, *CBL-Mariner*, less than 1,000. The others range from 1,400 to 16,000 pull requests, more or less evenly distributed. Such a selection of study subjects allows the metrics to be analysed also in terms of project size.

C. DATA ACQUISITION

In order to collect the data, a Python program was developed that queried the GitHub service via a REST API for data related to continuous integration and stored the results in a database. When automatically collecting data, an important limitation was that the GitHub API only allows five thousand queries per hour, which severely limited the amount of data collected. As a result, it was necessary to collect data for about a month to gather sufficient volume of information.

The data was retrieved from GitHub in March and April 2022. Altogether data on 101,149 pull requests, 399,671 commits by 20,432 developers were downloaded. Detailed metrics on the data collected from the individual repositories are shown in the Table 1.

IV. RESULTS

This section presents the results of a study analyzing metrics related to continuous integration in selected open source projects software development projects. The study aimed to provide answers to stated questions about the factors that impact pull requests success.

A. Q1: HOW MUCH EFFORT WAS REQUIRED TO FIX THE CODE AFTER CI FAILED BUILD?

Understanding what effort is needed to fix code after a failed continuous integration (CI) build can help to manage development projects and allocate resources more effectively. This enables project managers and development teams to make more informed decisions about how to prioritise individual tasks and thus perform their work more efficiently.

1) Q1M1: EXTENT OF CHANGES REQUIRED TO IMPROVE THE CODE AFTER CI FAILED BUILD

This metric measures how serious the errors were in the prepared pull requests that caused the compilation to fail. The metric was called the ''fixing lines ratio'' and was defined as the ratio of the number of new lines of code in the first pass that fixed the run to the number of new lines introduced in the first pull request that failed. For example, if the first commit that failed the build changed 100 lines of code, and the commit that fixed the build changed 10 lines, the metric will be 0.1. On the other hand, if there were more lines in the commit that fixed the build than in the commit that broke it, the value of the metric will be greater than 1.

For all the repositories studied, the distribution of metric values is shown in the Figure 1, and detailed results are presented in Table 2. For more than 40% of the cases where a commit caused a build to fail, it required a minor amount of code modifications, i.e. the number of modified lines accounted for no more than 20% of the code line modifications in the initial commit. In only 33.94% of cases it was necessary to modify the same or greater number of lines of code than in the initial commit.

TABLE 1. Repositories selected for the study.

Repository name	# pull requests	# commits	# contributors
flutter/flutter	35,956	164,775	2,169
bitcoin/bitcoin	16,174	61,876	2,167
django/django	14,746	53,394	4,497
hashicorp/terraform	11,012	44,839	2,695
microsoft/vscode	10,377	31,493	3,114
laravel/framework	5,777	15,208	2,529
diasurgical/devilutionX	2,791	13,134	234
starship/starship	1,957	8,512	530
facebook/lexical	1,425	3,360	61
microsoft/CBL-Mariner	888	3,080	82

TABLE 2. Changes required to improve the code after CI failed build.

Fixing lines ratio range	Number	Percent
$(0.0 - 0.2>$	4206	42.89%
$(0.2 - 0.5>$	1316	13.42%
$(0.5 - 1.0>$	956	9.75%
$(1.0 - 2.0>$	1050	10.71%
$(2.0 - 5.0>$	688	7.02%
$(5.0 - \infty>$	1590	16.21%

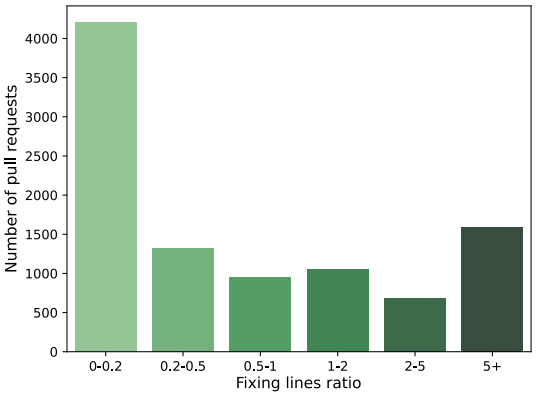


FIGURE 1. Changes required to improve the code after CI failed build.

The histograms in Figure 2, and Table 3, provide a summary of this data broken down by repository. For the repository with the highest number of cases in which a commit in pull requests caused the build to fail, i.e. the *flutter* repository, code improvements usually required only minor changes. A similar situation occurs with the *lexical* repository. This may indicate that the developers submitting pull requests are not sufficiently testing the changes they are making in their working environments. This may be a sign of immaturity in the code development process.

On the other hand, in the *bitcoin* repository, the situations where a commit causes a build to fail are relatively few, but their fixes in most cases require more work, i.e., making changes to a significant number of lines of code.

2) **Q1M2**: TIME TO DELIVER A COMMIT FIXING CI FAILED BUILD

The next metric determines how much time it takes to prepare a commit, correcting a failing build. The metric was defined as the time from submitting the commit that caused the build to fail, to submitting the commit that fixed the build.

For all the repositories studied, the distribution of metric values is shown in the Figure 3, and detailed results are presented in Table 4. In most cases (54.21%), it took no more than 3 hours to prepare a commit that fixed the build. This means that introducing the necessary code modifications was not too complex. The change to fix the build took more than a day to prepare only 19.67% of the cases. It should also be noted that only repositories of open source projects were examined. This means that at least some of the developers work on the project part-time, which can significantly affect the time it takes to prepare a fix.

The graphs in Figure 4, and Table 5, provide a summary of this data broken down by repository. Among the repositories examined, three, i.e. *flutter*, *bitcoin* and *terraform*, deviate significantly in the distribution of fix preparation time. They are among the most active projects studied, with 35,956, 16,174 and 11,012 pull requests, respectively. This may indicate that such projects have longer fix delivery times than less active projects. On the other hand, in the case of the *django* repository, which is also a very active project with 14,746 pull requests, the time to deliver a commit that fixes the build, is definitely shorter. However, this repository is notable for the very low number of situations where a build has failed after an initial commit to pull requests. This may indicate a high degree of maturity in the process of introducing new functionality into the code.

3) **Q1M3**: NUMBER OF MODIFIED FILES OF EACH TYPE AFTER CI BUILD FAILURE

The last metric for the first question, concerned the types of files that were modified after a failed build. Three groups of files were distinguished: source files, test files, and configuration files. From the point of view of the amount of work involved in preparing a fix, the last two are particularly interesting. Changing test files can imply two things: either that the tests developed so far are broken, or that the developer has changed the test just to get the build to pass. Either way, it indicates some deficiency in the application of the continuous integration process. All tests should be prepared and verified in advance to catch any errors when implementing new functionality. Changing test files indicates either that the developer who created the pull

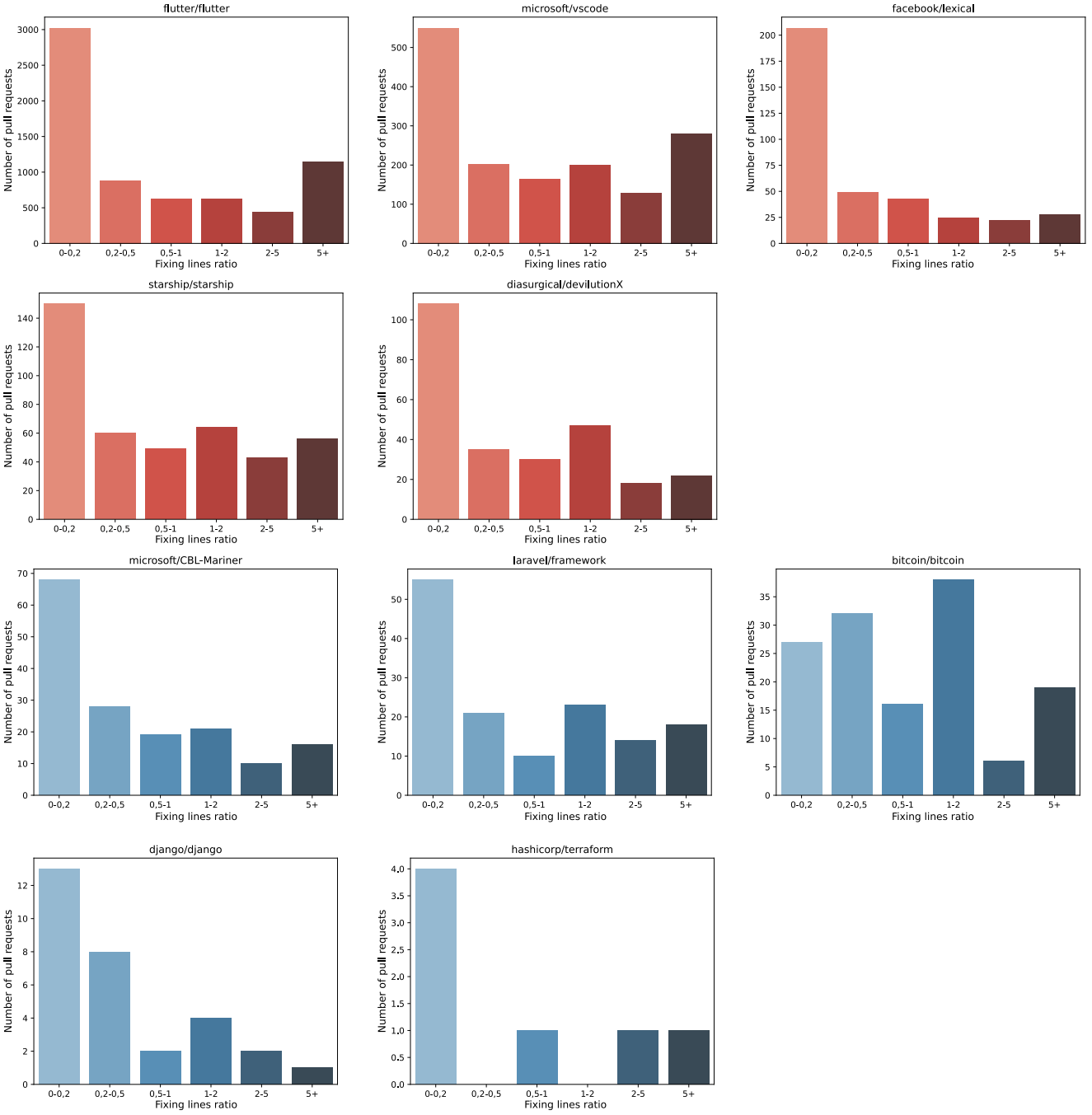


FIGURE 2. Code lines modification ratio by repositories.

request did not review the tests he or she developed, or that the tests prepared earlier did not cover all cases. On the other hand, modifying test files just to pass a build would be a completely reprehensible practice. Frequent changes to configuration files may indicate the immature state of the continuous integration process.

For all the repositories studied, the distribution of metric values is shown in the Figure 5, and detailed results are presented in Table 6. More than half of the modified files were source files, but as many as 36.74% were test files and 10.91% were configuration files. These results may indicate a certain immaturity of the continuous integration process. Analyzing the metrics for individual repositories, shown in Table 7, only two deviate from this distribution. For the *devilutionX* project, as many as 85.13% of the changes involved source files. For the *CBL-Mariner* project, on the other hand, the vast majority of changes involved configuration files.

TABLE 3. Code lines modification ratio by repositories.

Repository (number of first builds failures)	0.0–0.2	0.2–0.5	0.5–1.0	1.0–2.0	2.0–5.0	5.0+
flutter/flutter (6745)	44.83%	13.05%	9.22%	9.30%	6.57%	17.03%
microsoft/vscode (1527)	36.02%	13.29%	10.74%	13.16%	8.45%	18.34%
facebook/lexical (374)	55.35%	13.10%	11.50%	6.68%	5.88%	7.49%
starship/starship (422)	35.55%	14.22%	11.61%	15.17%	10.19%	13.27%
diasurgical/devilutionX (260)	41.54%	13.46%	11.54%	18.08%	6.92%	8.46%
microsoft/CBL-Mariner (162)	41.98%	17.28%	11.73%	12.96%	6.17%	9.88%
laravel/framework (141)	39.01%	14.89%	7.09%	16.31%	9.93%	12.77%
bitcoin/bitcoin (138)	19.57%	23.19%	11.59%	27.54%	4.35%	13.77%
django/django (30)	43.33%	26.67%	6.67%	13.33%	6.67%	3.33%
hashicorp/terraform (7)	57.14%	0.00%	14.29%	0.00%	14.29%	14.29%

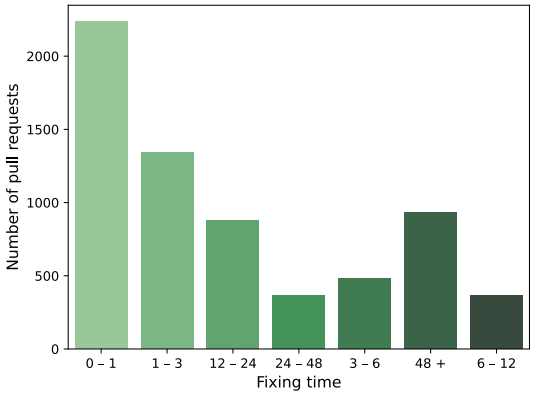


FIGURE 3. Time to deliver fix.

TABLE 4. Time to deliver fix.

Time (hours)	Volume	Share
$(0-1>$	2236	33.86%
$(1-3>$	1344	20.35%
$(3-6>$	483	7.31%
$(6-12>$	362	5.48%
$(12-24>$	879	13.31%
$(24-48>$	364	5.51%
$(48-\infty>$	935	14.16%

4) SUMMARY

Based on the metrics collected to answer RQ1, it can be concluded that implementing changes to fix a broken build generally requires little effort. However, the level of effort depends on the project's activity and the maturity of its process for introducing new functionality and changes. Projects with a high number of pull requests may require more time to prepare the necessary fixes.

In contrast, projects with a mature continuous integration process, where build failures after pull requests are relatively rare, would require more significant changes to address any issues. However, a more effective approach for such projects would be to detect bugs early during the test run, while they are still in the developer's local work environment.

Based on the data collected, it can be observed that many developers in the open source projects studied rely heavily on CI systems and often neglect to test the pulled code in their own development environments. This behavior suggests that there is room for improvement in testing practices within these projects.

B. Q2: HOW DOES THE AMOUNT OF CHANGES INTRODUCED IN PULL REQUEST AFFECT THE SUCCESS OF THE BUILD?

Intuitively, the smaller the number of changes made to the code, the smaller the chance that the build will fail. Implementing a large number of new features, and thus a large number of new lines of code, increases the likelihood that a programmer will make a mistake. On the other hand, even a small fix can lead to bugs at the interface between different modules that a programmer working on a large project may not even be aware of. Three metrics were analyzed to answer the question of how the number of changes in the project affects the success of the build run.

1) Q2M1: DEPENDENCY OF BUILD RESULT ON NUMBER OF LINES CHANGED IN SOURCE FILES

This metric summarises the number of lines modified in the source files for successful and unsuccessful build runs. Nine ranges for the number of modified lines were defined and the number of successful and unsuccessful runs was calculated for each range. The results are shown in Table 8.

The difference in percentage points between the number of successful and failed runs for each range is shown in Figure 6. It can be observed that the greatest difference in favour of successful runs actually occurs for small commits, where the number of modified lines does not exceed 30. For commits with more than 100 modified lines, more builds fail than succeed. A certain anomaly can be observed in the case of commits for the range where the number of modified lines exceeds 5,000. Closer analysis of this case showed that it was related to the fact that there were relatively few such commits and, in addition, almost all from only two repositories, i.e. *microsoft/vscode* and *flutter/flutter*. In the former, for commits in which more than 5,000 lines of code were modified, there were an unexpectedly high number of those completed successfully. Corresponding graph for data from all repositories, excluding *microsoft/vscode*, is shown in Figure 7, where a clear dependency can be seen that, as the number of modified lines in a commit increases, the chance that a build run will fail increases.

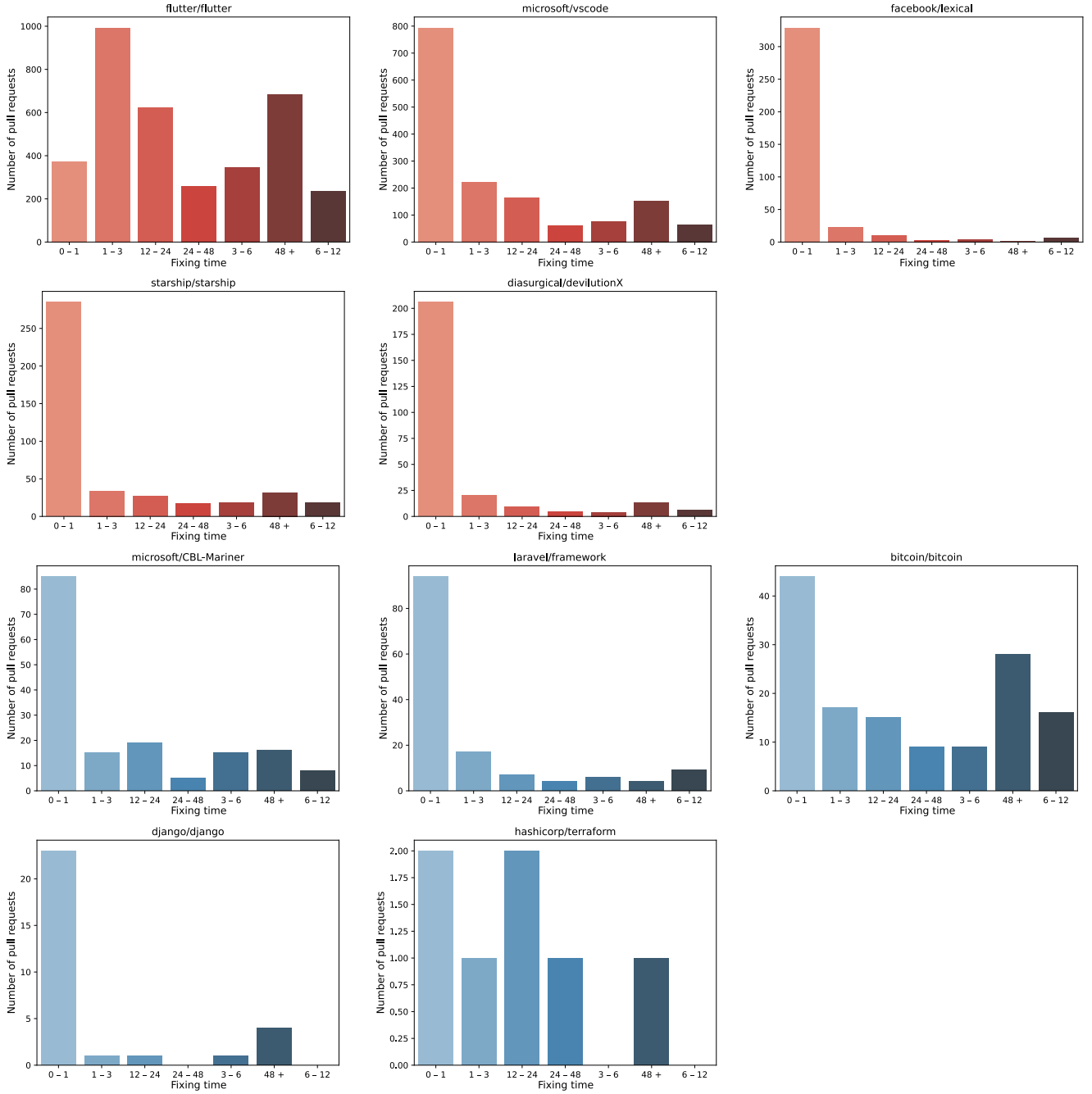


FIGURE 4. Time to deliver fix by repos.

2) **Q2M2**: DEPENDENCY OF BUILD RESULT ON NUMBER OF LINES CHANGED IN TEST FILES

The next metric is similar to the first, except that it takes into account the number of modified lines in the test files. The values for each range are shown in Table 9 and the difference in percentage points between the number of successful and failed runs for each range is shown in Figure 8

In the case of the test files, no such clear relationship can be observed as with the number of lines modified in the source files. However, it does confirm that the fewer lines modified, the greater the chance that the build will succeed.

3) **Q2M3**: NUMBER OF COMMITS IN PULL REQUEST

The last metric shows how many commits are submitted in a single pull requests. It is good practice for developers to commit small changes so that in case of an error they can easily roll back to a working version. However, on the other hand, merging numerous commits into the main branch in a single pull requests can unnecessarily obscure the git history.

TABLE 5. Time to deliver fix by repos.

Repository	0 – 1	1 – 3	3 – 6	6 – 12	12 – 24	24 – 48	48 +
flutter/flutter	10.64%	28.25%	9.90%	6.69%	17.72%	7.37%	19.43%
bitcoin/bitcoin	31.88%	12.32%	6.52%	11.59%	10.87%	6.52%	20.29%
django/django	76.67%	3.33%	3.33%	0.00%	3.33%	0.00%	13.33%
hashicorp/terraform	28.57%	14.29%	0.00%	0.00%	28.57%	14.29%	14.29%
microsoft/vscode	51.59%	14.49%	5.07%	4.16%	10.79%	3.96%	9.94%
laravel/framework	66.67%	12.06%	4.26%	6.38%	4.96%	2.84%	2.84%
diasurgical/devilutionX	78.33%	7.60%	1.52%	2.28%	3.42%	1.90%	4.94%
starship/starship	66.13%	7.89%	4.18%	4.18%	6.26%	3.94%	7.42%
facebook/lexical	87.50%	6.12%	1.06%	1.6%	2.66%	0.8%	0.27%
microsoft/CBL-Mariner	52.15%	9.20%	9.20%	4.91%	11.66%	3.07%	9.82%

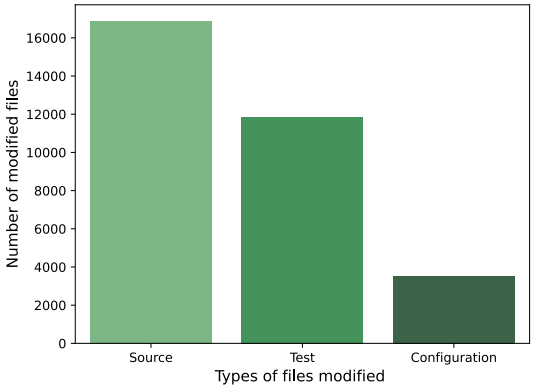


FIGURE 5. Number of modified files in each group.

TABLE 6. Number of modified files in each group.

Files type	Volume	Share
Source	16 888	52.35%
Test	11 851	36.74%
Configuration	3 520	10.91%

TABLE 7. Files type by repos.

Repository	Source	Test	Configuration
flutter/flutter	50.49%	42.26%	7.25%
bitcoin/bitcoin	59.86%	28.57%	11.56%
django/django	59.52%	40.48%	0.00%
hashicorp/terraform	50.00%	50.00%	0.00%
microsoft/vscode	56.91%	24.51%	18.58%
laravel/framework	56.38%	39.36%	4.26%
diasurgical/devilutionX	85.13%	4.09%	10.78%
starship/starship	60.42%	21.96%	17.61%
facebook/lexical	51.46%	38.20%	10.34%
microsoft/CBL-Mariner	12.96%	2.54%	84.51%

TABLE 8. Number of modified lines in source files depending on result of the build.

	Success builds		Failed builds	
No. of modified lines	Volume	Share	Volume	Share
1 – 10	12157	59.13%	8402	40.87%
11 – 30	6211	58.29%	4445	41.71%
31 – 50	2333	54.47%	1950	45.53%
51 – 100	2449	53.20%	2154	46.80%
101 – 200	1453	49.42%	1487	50.58%
201 – 500	1118	48.34%	1195	51.66%
501 – 2000	1134	48.80%	1190	51.20%
2001 – 5000	781	48.60%	826	51.40%
5001+	661	52.59%	596	47.41%

It is therefore common practice to squash commits, before submitting a pull request.

TABLE 9. Number of modified lines in test files depending on result of the build.

	Success builds		Failed builds	
No. of modified lines	Volume	Share	Volume	Share
1 – 10	5302	54.23%	4474	45.77%
11 – 30	3376	57.61%	2484	42.39%
31 – 50	1700	55.48%	1364	44.52%
51 – 100	1921	55.25%	1556	44.75%
101 – 200	1128	49.19%	1165	50.81%
201 – 500	1019	51.70%	952	48.3%
501 – 2000	1170	49.10%	1213	50.9%
2001 – 5000	504	45.16%	612	54.84%
5001+	157	50.0%	157	50.0%

TABLE 10. Number of commits in pull requests.

No. of commits	No. of pull requests	Share
1 – 2	66982	73.19%
3 – 5	13313	14.55%
6 – 10	5405	5.91%
11 – 20	3112	3.40%
21 – 50	2019	2.21%
51+	690	0.75%

A summary of the number of pull requests for each of the 6 ranges is shown in Table 10. The results confirm that the developers of the studied open source projects follow the approach of pushing a small number of commits in pull requests. Over 73% of PRs had no more than two commits, and almost 90% no more than five.

Figure 9 shows, in addition to the number of commits in the PR, the number of modified lines. There is a clear trend towards small pull requests, where the number of modified lines is low. In 43.14% of Pull Reqeusts no more than 10 lines were modified, and in a further 21.14% between 11 and 20. Only in less than 20% of cases were more than 100 lines of code modified.

Looking closely at the data in Table 11, which shows the number of commits in PR for each of the repositories studied, the *django/django* repository can be distinguished. In its case, as many as 91.11% of pull requests consisted of one or two commits, and only 3.16% consisted of more than five. On the other hand, there are the *flutter/flutter*, *microsoft/CBL-Mariner* and *microsoft/vscode* repositories, where the number of PRs with one or two commits is less than 70%. An interesting observation is that all these projects are managed by large technology corporations, the first by Google, the other two by Microsoft.

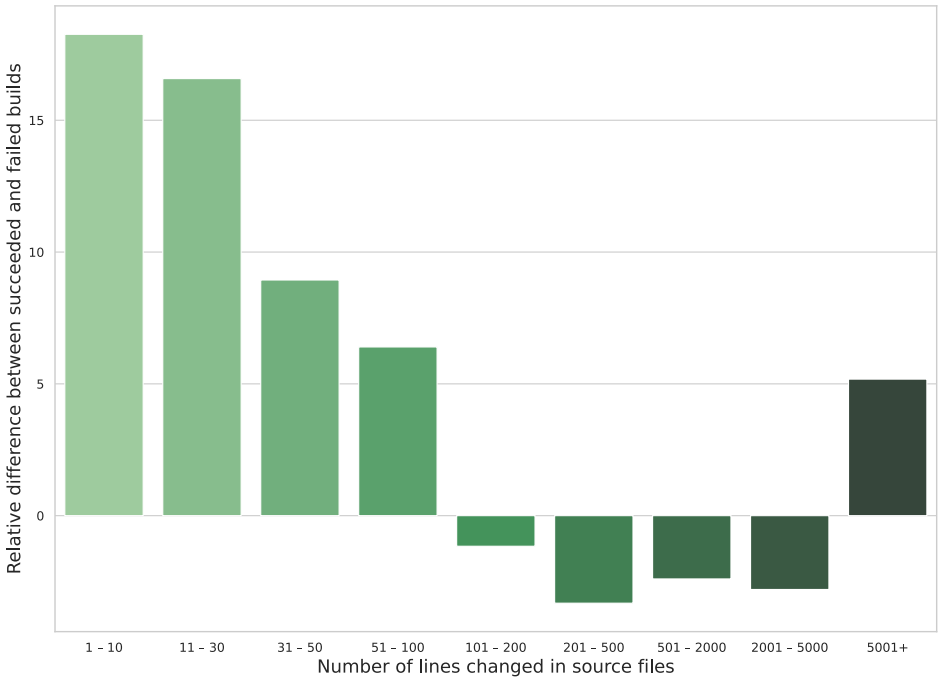


FIGURE 6. Number of modified lines in source files depending on result of the build.

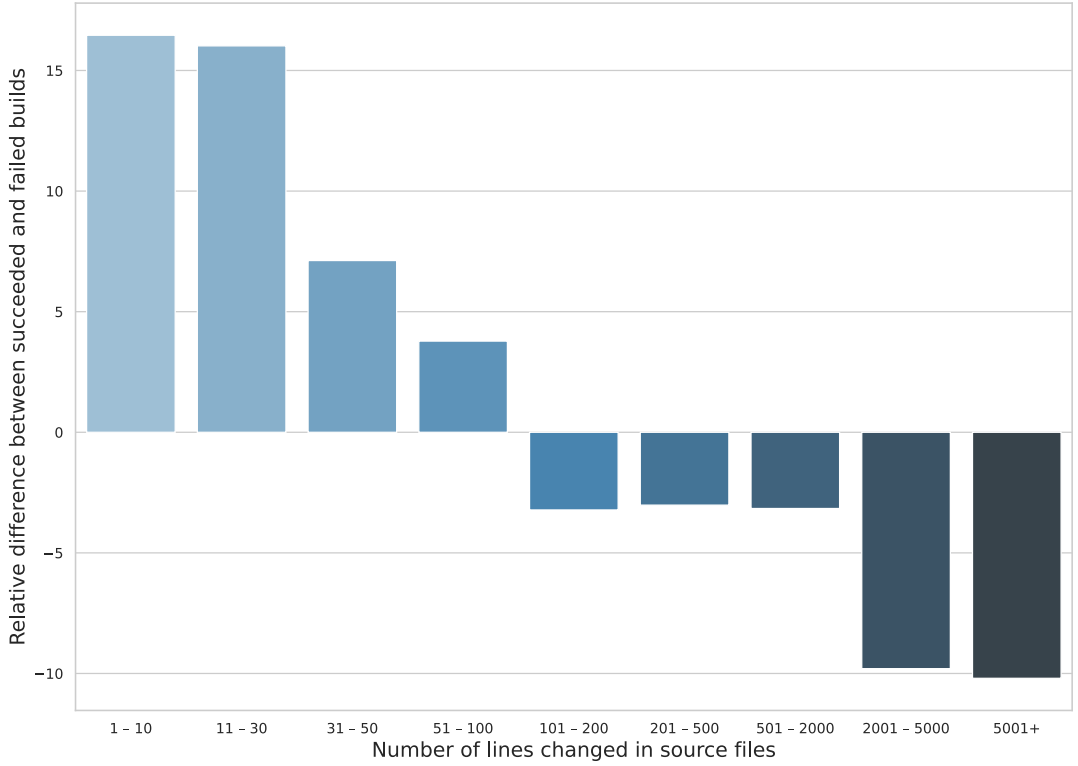


FIGURE 7. Number of modified lines in source files depending on result of the build, *microsoft/vscode* excluded.

4) SUMMARY

In response to RQ2, based on the calculated metrics, it is possible to confirm the intuitive hunch that the smaller the number of changes introduced in the code, the greater the chance that the launch of the CI build will succeed. In the repositories examined, for commits where the number of

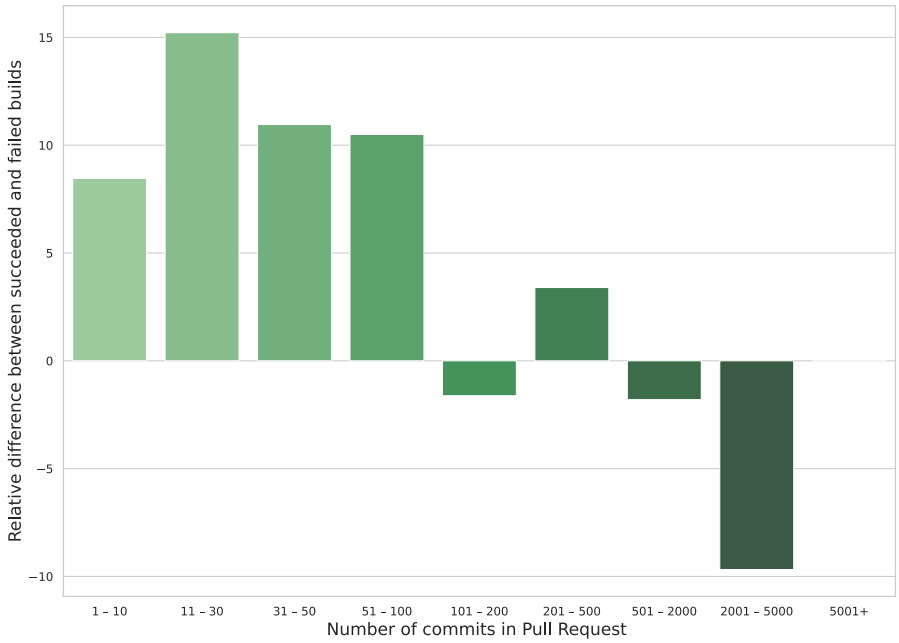


FIGURE 8. Number of modified lines in test files depending on result of the build.

TABLE 11. Number of commits in pull requests by repository.

Repository	Number of commits					
	1 – 2	3 — 5	6 — 10	11 — 20	21 — 50	50 +
bitcoin/bitcoin	79.13%	13.26%	4.30%	1.46%	0.63%	1.22%
diasurgical/devilutionX	76.85%	13.94%	4.98%	2.51%	1.51%	0.22%
django/django	91.11%	5.73%	1.41%	0.53%	0.30%	0.92%
facebook/lexical	75.19%	19.72%	3.60%	1.34%	0.14%	0.00%
flutter/flutter	63.38%	17.20%	8.51%	6.02%	4.28%	0.61%
hashicorp/terraform	80.47%	12.74%	3.83%	1.61%	0.76%	0.60%
laravel/framework	80.29%	13.59%	3.67%	1.16%	0.85%	0.43%
microsoft/CBL-Mariner	61.89%	23.56%	8.68%	3.95%	1.58%	0.34%
microsoft/vscode	66.58%	18.26%	8.03%	3.98%	2.36%	0.79%
starship/starship	72.24%	13.50%	5.98%	3.83%	2.20%	2.25%

modified lines of code exceeded 100, more runs ended in failure than in success.

The study confirmed that developers aim to submit small pull requests where possible, thus increasing the chance that the continuous integration run will be successful. In terms of both the number of commits and the number of modified lines, small pull requests were far more frequently submitted than large ones.

C. Q3: DOES THE EXPERIENCE OF THE CONTRIBUTOR AFFECT THE SUCCESS OF CI BUILDS?

Intuitively, the more experience a programmer has, the fewer mistakes he or she should make. However, our own experience and observations have shown that this is not always the case with CI builds. By investigating whether there is a correlation between the success of a build and the experience of the pull request author, we can gain valuable insight into how developers select and handle the issues they work on.

1) Q3M1: CI RUN RESULTS DEPENDING ON THE EXPERIENCE OF THE CONTRIBUTOR

The experience of the developers is crucial to the efficiency and quality of their work. For this reason, it would appear that commits from more experienced programmers will fail less often than those from novices. However, the skills of developers cannot be reliably determined from the data downloaded from GitHub. The number of commits submitted to a given repository was taken as a factor of contributor experience.

The values for the nine ranges of the number of commits submitted are shown in Table 12. The difference in percentage points between the number of successful and failed runs for each range is shown in Figure 10

The results are surprising with no apparent correlation between the experience of the contributors and the rate of successful runs. On the contrary, novice contributors, defined as those with fewer than 10 commits to the repository, appear to have the highest percentage of successful builds.

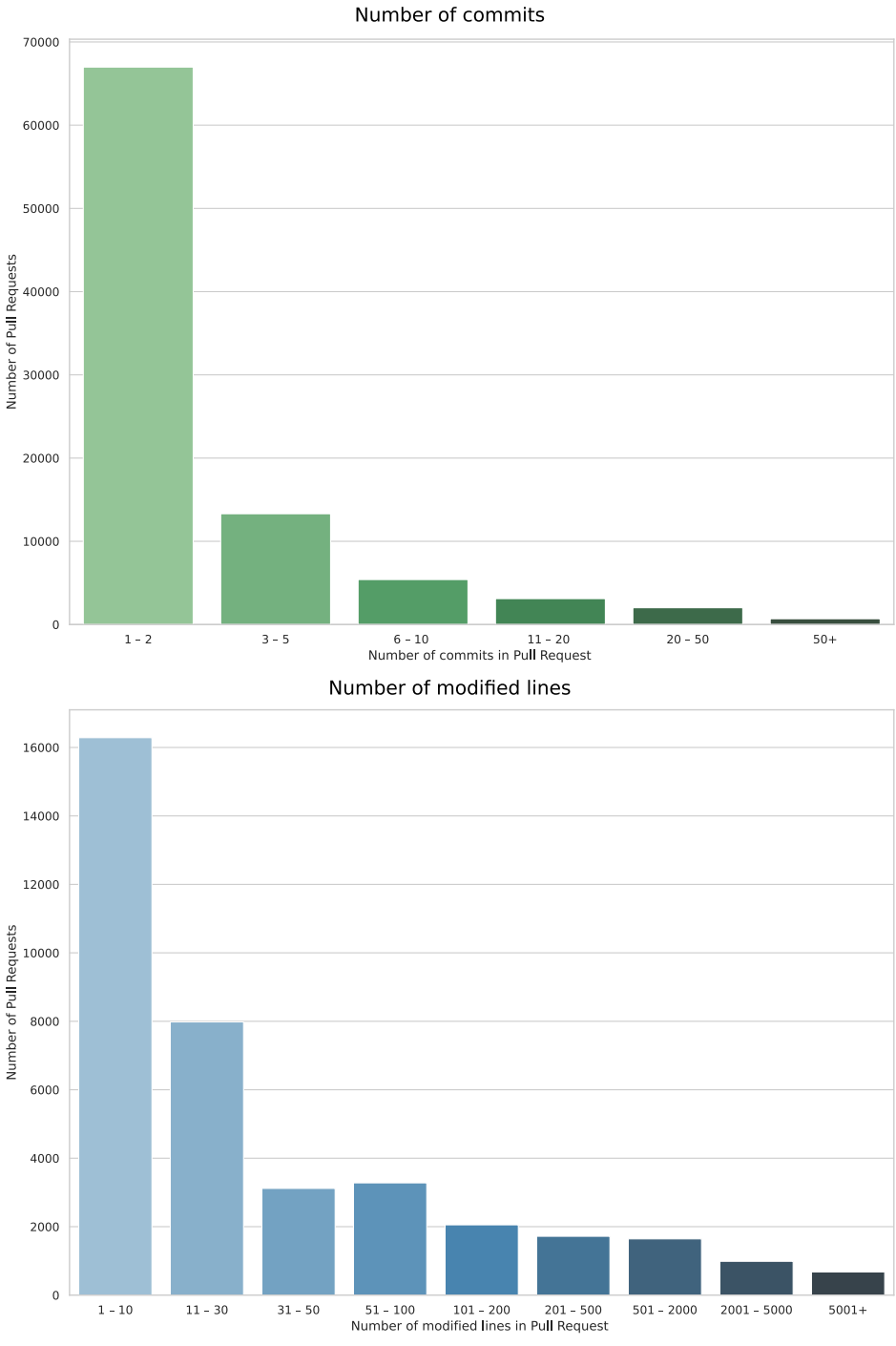


FIGURE 9. Number of commits and modified lines in pull requests.

Therefore, it was examined whether any of the repositories were disturbing the summary results. A visualisation of the results is shown for each repository in Figure 11.

In fact, only for the two repositories, i.e. *bitcoin/bitcoin* and *starship/starship*, a clear correlation can be observed that, as the experience of the contributors increases, the difference between runs resulting in success than failure increases.

This phenomenon may be related to the types of tasks contributors implement. Programmers who are new to a project tend to choose simple tasks that are easier to complete without error. Only as they gain experience they begin to implement more complex tasks. A second factor may be the increased confidence of experienced developers. In such cases, they may not test the new code thoroughly enough

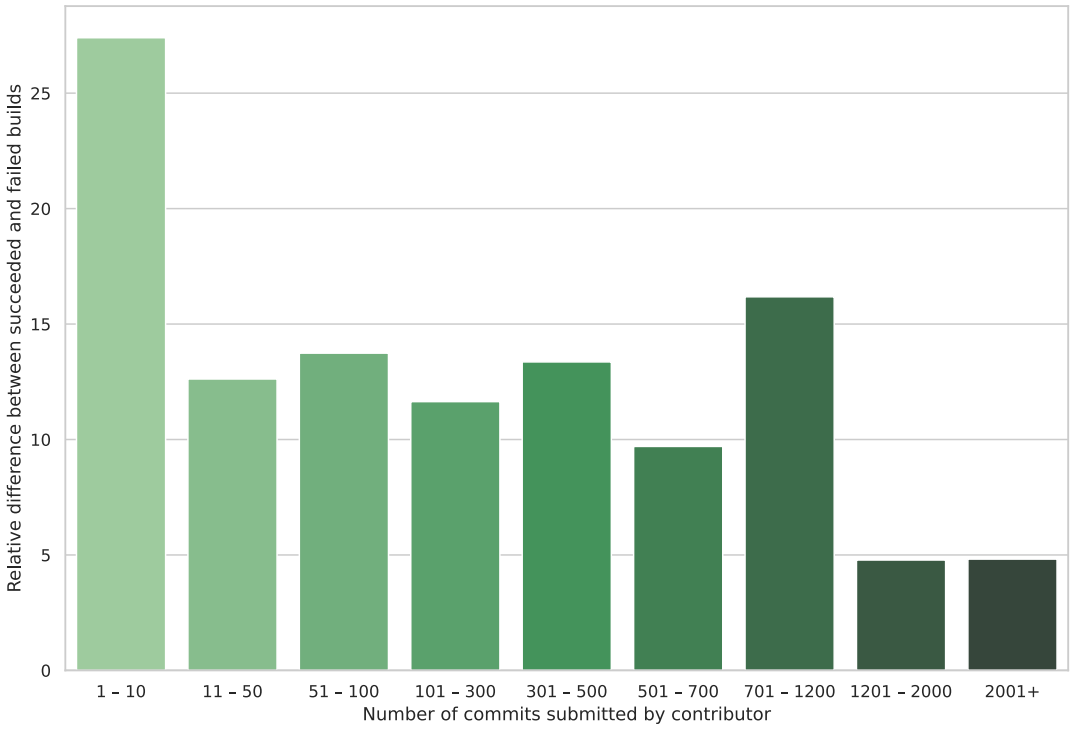


FIGURE 10. Summary CI run results depending on the experience of the contributor.

TABLE 12. Summary CI run results depending on the experience of the contributor.

Number of commits submitted by contributor	Success builds		Failed builds	
	Number	Percent	Number	Percent
1 – 10	7962	63.7	4538	36.3
11 – 50	5889	56.31	4570	43.69
51 – 100	3149	56.87	2388	43.13
101 – 300	6865	55.82	5434	44.18
301 – 500	3637	56.68	2780	43.32
501 – 700	3090	54.85	2544	45.15
701 – 1200	5735	58.09	4138	41.91
1201 – 2000	4219	52.39	3834	47.61
2001+	9156	52.41	8315	47.59

in their development environment and immediately submit a pull request.

2) SUMMARY

In response to RQ3, the experience of the contributors working on the project was found to have no effect on the build success rate. For some of the repositories studied, it appeared that builds triggered by commits from more experienced developers were more likely to fail than those submitted by novice contributors.

V. DISCUSSION

The goal of this study was to investigate how continuous integration techniques are used by developers in open source projects. Answering the three research questions posed through an exploratory study of selected code repositories of open source projects provided insights into the practical application of CI techniques. Thus, we believe that this study makes a significant contribution to the field of computer science by adding important new insights.

In most of the repositories examined, fixing the bugs that caused the build to fail required modifying a small number of lines of code and was delivered by the developers quickly, usually within a few hours. This may indicate that developers are not fully testing their code in the local development environment before submitting a pull request. However, in projects where it was much less common for a commit to break the build, bug fixes required more code changes and were more time-consuming. In our opinion, there is a much more desirable approach. The continuous integration systems should not be an excuse for developers not to sufficiently test the code they deliver. Another disturbing metric that confirms the immature approach to continuous integration is the type of files modified after a failed build. Only slightly more than half of the changes were made to source files, while more than 35% were made to test files. This again shows that developers are not paying enough attention to creating and verifying unit tests.

The results of the study confirmed that commits with a small number of changes have a greater chance of completing the build successfully. On the other hand, as the size of the commit increases, the likelihood of the build failing increases, and huge commits with several thousand modified lines are more likely to fail than succeed. According to the results collected, developers are working in line with this observation – the vast majority of pull requests consist of one or two commits and no more than a dozen lines changed. In our

FIGURE 11. Summary CI run results depending on the experience of the contributor for each repository.

view, this is a good practice that should be promoted among developers. Not only does it minimize the risk of introducing bugs into the code, but also makes it easier for collaborators to prepare code reviews.

Finally, the intuition that commits from experienced developers are more likely to succeed than those from newer developers has not been confirmed. This is most likely related to the fact that experienced developers undertake more complex tasks. In our opinion, this is the appropriate approach. By solving simple tasks, novice developers become familiar with the project and slowly gain experience. Eventually, as time passes and more tasks are completed, they can take on more challenging issues.

A. THREATS TO VALIDITY

Following the approach proposed by Wohlin et al. [43], a methodological analysis of threats to validity was conducted. The threats are outlined in the following order: internal validity, construct validity, and finally external validity.

Internal validity is a concern with threats that may have an impact on the independent variables [43]. In this context, there may be risks associated with drawing incorrect conclusions about the maturity of a project based on the analysis of the data collected alone. Future work will include analysis of other development project artifacts for a more holistic view of the problem.

Construct validity refers to the extent to which the results of a test can be generalized to the underlying theoretical construct [43]. The choice of metrics is the main risk in this area. The GQM approach, which allows exploratory research to be well structured, was chosen to minimise this risk.

External reliability refers to factors that limit the ability to generalise experimental results to industrial practice [43]. In this respect, a potential threat may be related to the number of repositories studied. However, our main goal was to thoroughly investigate the use of CI systems over the years. Therefore, for the 10 selected repositories, a large dataset consisting of 101,149 pull requests, 399,671 commits from 20,432 developers was collected. We believe that this risk has been mitigated by selecting repositories that are diverse in size and activity.

VI. CONCLUSION

This paper describes the results of an exploratory study of the use of continuous integration techniques in open source projects. The study defined 3 research questions and 8 metrics. Data extracted from 10 GitHub repositories of open source projects were then analysed for the use of CI techniques.

The results allow us to draw the following three main conclusions. Firstly, the majority of the problems that were discovered during the CI build runs required a small change in the pull request to fix the code. However, we identified projects with a more mature approach to software development, where failed build runs were rare. In these cases, fixing the code required major code changes. This shows that in most of the projects studied, developers do not sufficiently test the code before submitting a pull request, indicating an immature approach to using continuous integration techniques. Secondly, developers tend to submit pull requests with small amounts of new or modified code, leading to a greater chance of CI build success. Thirdly, there was no correlation found between developer experience in committing to the project and the success rate of CI builds. We believe this is because more experienced developers take on more complex problems to solve.

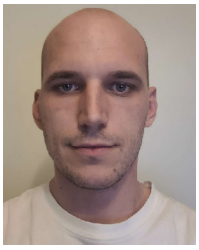
Furthermore, in future research, we would like to investigate the maturity of open source projects in terms of their use of continuous integration techniques. To this end, we plan to analyse other available software engineering artefacts from selected projects and compare the results with the data collected in this study.

REFERENCES

[1] P. Weichbroth, "A case study on implementing agile techniques and practices: Rationale, benefits, barriers and business implications for hardware development," *Appl. Sci.*, vol. 12, no. 17, p. 8457, Aug. 2022.
[2] M. A. Akbar, J. Sang, A. A. Khan, M. Shafiq, S. Hussain, H. Hu, M. Elahi, and H. Xiang, "Improving the quality of software development process by introducing a new methodology–AZ-model," *IEEE Access*, vol. 6, pp. 4811–4823, 2018.
[3] M. A. Akbar, W. Naveed, A. A. Alsanad, L. Alsuwaidan, A. Alsanad, A. Gumaei, M. Shafiq, and M. T. Riaz, "Requirements change management challenges of global software development: An empirical investigation," *IEEE Access*, vol. 8, pp. 203070–203085, 2020.
[4] R. Subramanya, S. Sierla, and V. Vyatkin, "From DevOps to MLOps: Overview and application to electricity market forecasting," *Appl. Sci.*, vol. 12, no. 19, p. 9851, Sep. 2022.
[5] M. S. Khan, A. W. Khan, F. Khan, M. A. Khan, and T. K. Whangbo, "Critical challenges to adopt DevOps culture in software organizations: A systematic review," *IEEE Access*, vol. 10, pp. 14339–14349, 2022.
[6] M. Marinho, R. Camara, and S. Sampaio, "Toward unveiling how SAFe framework supports agile in global software development," *IEEE Access*, vol. 9, pp. 109671–109692, 2021.
[7] O. Springer, J. Miler, and M. R. Wróbel, "Strategies for dealing with software product management challenges," *IEEE Access*, vol. 11, pp. 55797–55813, 2023.
[8] Z. S. Li, C. Werner, N. Ernst, and D. Damian, "Towards privacy compliance: A design science study in a small organization," *Inf. Softw. Technol.*, vol. 146, Jun. 2022, Art. no. 106868.
[9] K. Gallaba and S. McIntosh, "Use and misuse of continuous integration features: An empirical study of projects that (Mis)Use travis CI," *IEEE Trans. Softw. Eng.*, vol. 46, no. 1, pp. 33–50, Jan. 2020.
[10] R. Haas, D. Elsner, E. Juergens, A. Pretschner, and S. Apel, "How can manual testing processes be optimized? Developer survey, optimization guidelines, and case studies," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Aug. 2021, pp. 1281–1291.
[11] A. Marcolini, N. Bussola, E. Arbitrio, M. Amgad, G. Jurman, and C. Furlanello, "Histolab: A Python library for reproducible digital pathology preprocessing with automated testing," *SoftwareX*, vol. 20, Dec. 2022, Art. no. 101237.
[12] F. Zampetti, G. Bavota, G. Canfora, and M. D. Penta, "A study on the interplay between pull request review and continuous integration builds," in *Proc. IEEE 26th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Feb. 2019, pp. 38–48.
[13] A. Łuczak, K. Stróżański, and C. Orłowski, "A model of a parallel design environment for the development of decision-making IoT systems," in *Transactions on Computational Collective Intelligence XXXVII*. Cham, Switzerland: Springer, 2023, pp. 157–170.
[14] B. Loriot, F. Madeiral, and M. Monperrus, "Styler: Learning formatting conventions to repair checkstyle violations," *Empirical Softw. Eng.*, vol. 27, no. 6, p. 149, Nov. 2022.
[15] Q. Zhang, D. K. Hong, Z. Zhang, Q. A. Chen, S. Mahlke, and Z. M. Mao, "A systematic framework to identify violations of scenario-dependent driving rules in autonomous vehicle software," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 5, no. 2, pp. 1–25, Jun. 2021.
[16] W. Felidre, L. Furtado, D. A. D. Costa, B. Cartaxo, and G. Pinto, "Continuous integration theater," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, Sep. 2019, pp. 1–10.
[17] K. Weiss, M. Rottleuthner, T. C. Schmidt, and M. Wählisch, "PHiLIP on the HiL: Automated multi-platform OS testing with external reference devices," *ACM Trans. Embedded Comput. Syst.*, vol. 20, no. 5s, pp. 1–26, Oct. 2021.
[18] S. M. Embury and C. Page, "Effect of continuous integration on build health in undergraduate team projects," in *Proc. Int. Workshop Softw. Eng. Aspects Continuous Develop. New Paradigms Softw. Prod. Deployment*, Chateau de Villebrumier, France: Springer, Mar. 2018, pp. 169–183.
[19] F. A. Abdul and M. C. S. Fhang, "Implementing continuous integration towards rapid application development," in *Proc. Int. Conf. Innov. Manage. Technol. Res.*, May 2012, pp. 118–123.
[20] V. Saquicela, G. Campoverde, J. Avila, and M. E. Fajardo, "Building microservices for scalability and availability: Step by step, from beginning to end," in *New Perspectives in Software Engineering*. Cham, Switzerland: Springer, 2021, pp. 169–184.
[21] A. Alnafessah, A. U. Gias, R. Wang, L. Zhu, G. Casale, and A. Filieri, "Quality-aware DevOps research: Where do we stand?" *IEEE Access*, vol. 9, pp. 44476–44489, 2021.
[22] Y. Wu, Y. Zhang, T. Wang, and H. Wang, "Characterizing the occurrence of dockerfile smells in open-source software: An empirical study," *IEEE Access*, vol. 8, pp. 34127–34139, 2020.
[23] J. Santos, D. A. da Costa, and U. Kulesza, "Investigating the impact of continuous integration practices on the productivity and quality of open-source projects," in *Proc. 16th ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.*, Sep. 2022, pp. 137–147.
[24] R. Silva and C. Bezerra, "Investigating the impact of bad practices in continuous integration on closed-source projects," in *Anais Estendidos do XII Congresso Brasileiro de Software, Teoria e Prática*. Nashville, TN, USA: SBC, 2021, pp. 46–52.

[25] C. Vassallo, S. Proksch, H. C. Gall, and M. Di Penta, "Automated reporting of anti-patterns and decay in continuous integration," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 105–115.
[26] I. Saidani, A. Ouni, M. W. Mkaouer, and F. Palomba, "On the impact of continuous integration on refactoring practice: An exploratory study on TravisTorrent," *Inf. Softw. Technol.*, vol. 138, Oct. 2021, Art. no. 106618.
[27] J. A. P. Lima and S. R. Vergilio, "Test case prioritization in continuous integration environments: A systematic mapping study," *Inf. Softw. Technol.*, vol. 121, May 2020, Art. no. 106268.
[28] X. Jin and F. Servant, "Which builds are really safe to skip? Maximizing failure observation for build selection in continuous integration," *J. Syst. Softw.*, vol. 188, Jun. 2022, Art. no. 111292.
[29] T. A. Ghaleb, D. A. da Costa, and Y. Zou, "An empirical study of the long duration of continuous integration builds," *Empirical Softw. Eng.*, vol. 24, no. 4, pp. 2102–2139, Aug. 2019.
[30] A. Barrak, E. E. Eghan, B. Adams, and F. Khomh, "Why do builds fail?—A conceptual replication study," *J. Syst. Softw.*, vol. 177, Jul. 2021, Art. no. 110939.
[31] M. Golzadeh, A. Decan, and T. Mens, "On the rise and fall of CI services in GitHub," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Mar. 2022, pp. 662–672.
[32] T. Kinsman, M. Wessel, M. A. Gerosa, and C. Treude, "How do software developers use GitHub actions to automate their workflows?" in *Proc. IEEE/ACM 18th Int. Conf. Mining Softw. Repositories (MSR)*, May 2021, pp. 420–431.
[33] I. Saidani, A. Ouni, M. Chouchen, and M. W. Mkaouer, "Predicting continuous integration build failures using evolutionary search," *Inf. Softw. Technol.*, vol. 128, Dec. 2020, Art. no. 106392.
[34] E. Laukkanen, J. Itkonen, and C. Lassenius, "Problems, causes and solutions when adopting continuous delivery—A systematic literature review," *Inf. Softw. Technol.*, vol. 82, pp. 55–79, Feb. 2017.
[35] Y. Wang, M. V. Mäntylä, Z. Liu, and J. Markkula, "Test automation maturity improves product quality—Quantitative study of open source projects using continuous integration," *J. Syst. Softw.*, vol. 188, Jun. 2022, Art. no. 111259.
[36] L. Yu, E. Alégroth, P. Chatzipetrou, and T. Gorschek, "Utilising CI environment for efficient and effective testing of NFRs," *Inf. Softw. Technol.*, vol. 117, Jan. 2020, Art. no. 106199.
[37] V. R. B. G. Caldiera and H. D. Rombach, "The goal question metric approach," in *Encyclopedia of Software Engineering*, 1994, pp. 528–532.
[38] P. Lima, J. Melegati, E. Gomes, N. S. Pereira, E. Guerra, and P. Meirelles, "CADV: A software visualization approach for code annotations distribution," *Inf. Softw. Technol.*, vol. 154, Feb. 2023, Art. no. 107089.
[39] S. Stradowski and L. Madeyski, "Exploring the challenges in software testing of the 5G system at Nokia: A survey," *Inf. Softw. Technol.*, vol. 153, Jan. 2023, Art. no. 107067.
[40] M. Ahrens and K. Schneider, "Improving requirements specification use by transferring attention with eye tracking data," *Inf. Softw. Technol.*, vol. 131, Mar. 2021, Art. no. 106483.
[41] Y. Crespo, C. López-Nozal, R. Marticorena-Sánchez, M. Gonzalo-Tasis, and M. Piattini, "The role of awareness and gamification on technical debt management," *Inf. Softw. Technol.*, vol. 150, Oct. 2022, Art. no. 106946.
[42] R. Malhotra, *Empirical Research in Software Engineering: Concepts, Analysis, and Applications*. Boca Raton, FL, USA: CRC Press, 2016.
[43] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Berlin, Germany: Springer, 2012.

MICHAL R. WRÓBEL received the Ph.D. degree in computer science from the Gdańsk University of Technology, Gdańsk, Poland, in 2011. Since 2006, he has been with the Faculty of Electronics, Telecommunications and Informatics, Department of Software Engineering, Gdańsk University of Technology. He is currently an Assistant Professor with the Gdańsk University of Technology. He is also a member of the Emotions in HCI Research Group, where he researches software usability, affective computing, and software management methods. His research interest includes a modern approach to software development management, with a particular focus on the role of human factors in software engineering.



JAROSLAW SZYMUKOWICZ is currently pursuing the Ph.D. degree with the Gdańsk University of Technology. He is a Backend Software Developer. His research interest includes designing and creating innovative software and processes around it.

PAWEL WEICHBROTH (Member, IEEE) received the M.A. degree in statistics from the University of Gdańsk, Poland, in 2003, and the Ph.D. degree in artificial intelligence from the University of Economics in Katowice, Poland, in 2014.

Moreover, for over 20 years, he was a Business Consultant and an IT Lecturer. Since 2018, he has been an Expert for the Ministry of Digital Affairs on a project for the development of public digital services. He is currently an Assistant Professor with the Department of Software Engineering, Gdańsk University of Technology. In this regard, he has authored over 40 research papers as journal articles, conference papers, and book chapters. His main research interests include software quality, machine learning, and knowledge management. He has been a member of the Scientific Community of Business Informatics and several international conference program committees. He has been actively acting as a reviewer and as an organizer.