2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)

On the Interplay between Non-Functional Requirements and Builds on Continuous Integration

Klérisson V. R. Paixão*, Crícia Z. Felício†, Fernanda M. Delfim* and Marcelo de A. Maia*
*Universidade Federal de Uberlândia – Uberlândia (MG), Brazil
{klerisson, fernanda, marcelo.maia}@ufu.br
†Instituto Federal do Triângulo Mineiro – Uberlândia (MG), Brazil
cricia@iftm.edu.br

***Abstract*—Continuous Integration (CI) implies that a whole developer team works together on the mainline of a software project. CI systems automate the builds of a software. Sometimes a developer checks in code, which breaks the build. A broken build might not be a problem by itself, but it has the potential to disrupt co-workers, hence it affects the performance of the team. In this study, we investigate the interplay between non-functional requirements (NFRs) and builds statuses from 1,283 software projects. We found significant differences among NFRs related-builds statuses. Thus, tools can be proposed to improve CI with focus on new ways to prevent failures into CI, specially for efficiency and usability related builds. Also, the time required to put a broken build back on track indicates a bimodal distribution along all NFRs, with higher peaks within a day and lower peaks in six weeks. Our results suggest that more planned schedule for maintainability for Ruby, and for functionality and reliability for Java would decrease delays related to broken builds.**

***Index Terms*—Software repository mining; Continuous integration; Topic models; Non-functional requirements;**

I. Introduction

> "In general the answer to how to stay efficient when a build is almost always broken is: ***stop breaking the build***." – *Anonymous*[1]

This excerpt from an online Question and Answer community lays out competing concepts of Continuous Integration (CI) in the software industry. CI means that a whole developer team works together on the mainline of a software project [1]. CI build-process automatically takes source code commits, compiles the code, and then progresses through a pipeline of testing. Sometimes one developer checks in the source code repository something that breaks the build, i.e. checks in code which does not compile or pass unit or code analysis tests. If on one hand, it may disrupt colleagues' work, on the other, it prevents breakages going unnoticed.

The build of a system is one of the first steps of moving software from development to customers. A failure in the build may not only disrupt the co-workers, but also the business [2]. Hence, as important as avoiding broken builds is the time taken to fix the build. Longer times mean more wasted developer time. Understanding for what reasons a set of source code changes broke the build is hard without developer's advice and becomes crucial to prevent problems [3]. Also, relying on the developer for such analysis it is feasible on small-scale cases.

[1]https://perma.cc/NS8Z-3GX8

A growing body of work in software engineering uses *topic analysis* to make sense of textual data in software repositories [4]. As we gain access to larger datasets, it becomes important to scale our ability to conduct such analyses [5]. In this direction, Hindle et al. established a link between topics computed from commit messages and non-functional requirements (NFRs) [6]. Their technique enables large-scale topic analysis over such artifacts, because NFRs are widely spread across software projects. Furthermore, that work shed some light on what a set of commit messages means in terms of NFRs. Therefore, *if failed CI builds are related to certain NFRs, then developers can use topics to prevent failures*.

In this study, we examine the NFRs categories computed from the list of all commits that were built in a given build job from Travis-CI (a CI platform for open-source software development [7]) in 1,283 projects from GitHub repository. By studying a large corpus of projects, we aim to empirically investigate the interplay of NFRs and Travis-CI builds statuses.

Our research is guided by two main research questions:

RQ1. Which NFRs occur more frequently in failed Travis-CI builds than successful ones?

RQ2. How long do NFR-related builds remain broken?

We found significant differences among NFRs related-builds statuses. Thus, tools can be proposed to improve CI with focus on new ways to prevent failures into CI. Further, our results suggest that more planned schedule for maintainability for Ruby, and for functionality and reliability for Java would decrease delays related to broken builds.

The paper outline is standard: literature review, material and method description, results, and conclusions.

II. Related Work

Our study inherits from a rich ecosystem of tools and applications for software repository mining, and draws on the insights of prior work in NFRs and topic modeling.

Non-functional requirements. While a functional requirement describes what a system should do, NFRs place constraints on the performance of the system, i.e. how it will do so [8]. NFRs may also describe aspects of the system related to its evolution over time, e.g., maintainability, extensibility, and documentation, to name a few. Unfortunately, there are a lot of disagreements on what NFRs really are. Mairiza et al. found 114 different NFRs classes [9], which contrasts with

978-1-5386-1544-7/17 $31.00 © 2017 IEEE
DOI 10.1109/MSR.2017.33

Fig. 1. The automatic non-functional requirement labeling computes commit messages topics from over 3 million Travis-CI build jobs.

the international standard ISO 9126 quality model [10], where NFRs are defined by six high-levels classes: maintainability, functionality, portability, efficiency, usability, and reliability. Eckhardt et al. analyzed NFRs taken from industrial requirements specifications to better understand their nature [11]. Their results suggest that NFRs are buried in functional requirements, insofar as we should not make any distinguish between them. Despite the aforementioned discussion, whether NFRs are correctly categorized, we cannot deny that NFRs concepts pervade all modern software projects, therefore, we can use such definitions to compare projects.

Topic Modeling. The history of topic models in academic research related to Software Engineering is long. For a comprehensive survey on this matter, we refer to Chen et al. [4]. Herein, we focus on topic analysis applied to commit messages. Hindle et al. mined commits in a windowed time fashion [12]. They applied *latent dirichlet allocation* (LDA) [13] technique in a 30-day period of commit messages to identify topic trends. Their technique allows the automated summarization of "what has been done" in a given time. In another work [14], those authors also used topic analysis to annotate commit messages, among other software artifacts, and map the results onto software project phases. Their idea is to propose an alternative approach to monitor software process compliance. With respect to the study of CI builds statuses, there are some common themes between our work and Santos and Hindle's work [15]. In that work, a n-gram language model was proposed to compute how "unusual" is a commit message. The results suggest a positive correlation between unusualness messages and builds failures.

In comparison, our goal is to investigate the correlation between NFRs developers were working on and CI builds statuses. We rely on the method of Hindle et al. that links a set of commits messages to NFRs [6]. However, since we are interested in system-wide builds statuses, our NFR related topics are extracted from all commits reported in CI builds.

III. Material and Method

A. TravisTorrent Dataset

TravisTorrent [7] is a synthesis of software projects from GitHub that have Travis-CI enabled. Version 8.2.2017 comprehends 3,702,595 builds from 1,283 projects. For our particular interest, the structure of the build entries involves the job id, project name, status, builds duration, started timestamp, and all commits that were built.

Regarding the status of a build, there are five values in the dataset. We consider in this study three of them: *passed*, which means a project has been built and passed its test suite; *failed*, a project failed to build or failed in its tests; and *errored*, a misconfiguration was found in the project. The last two statuses were grouped. Ultimately, they both mean that the build is *broken*. We discard the other two statuses (started and canceled), because we either do not know the process outcome and the reasons behind its cancellation.

Additionally, with the project name, we fetch (clone) the repository from GitHub. Then, with the commit list, all messages are taken.

B. NFR Labeling

The overall NFR automatic labeling process is illustrated as Fig. 1. First, for each GitHub project we clone the repository. Then, we select the commits that were built for each build job. With the commit we fetch the associated messages.

Such messages, per project, are given as input to the topic modeling phase. We use the Mallet toolkit [16] to generate 20 topics with 10 words per topic. To automatically label each build job with a topic, we use the *exp3* word-list, please refer to the work of Hindle et al. [6] for the details on the word-list generation. This word set consists of keywords separated by each NFR (maintainability, functionality, portability, efficiency, usability, and reliability).

The motivation to choose this word-list instead of others (exp1 or exp2) is because it contains more words per NRF category. Since we aim to contrast diverse projects a broad list of words might be better representative. Recall that this process is done per project. So, the topic computing of one project is not affected by the topics from others.

Finally, with each build job and its associated topic, we labeled our build job with an NFR where there was a match between the topic's word and the word-list.

IV. Results and Discussion

This section reports our results. For replication purposes, raw data used for our analyses is available for download[2].

RQ1. Which NFRs occur more frequently in failed Travis-CI builds than successful ones?

While a common best practice on continuous integration is to have all the tests passing at all times, build breakage happens. The primary endpoint of the study is to identify patterns of failure that might help developers prioritize their efforts on preventing such failures.

[2]https://doi.org/10.6084/m9.figshare.2279505.v1

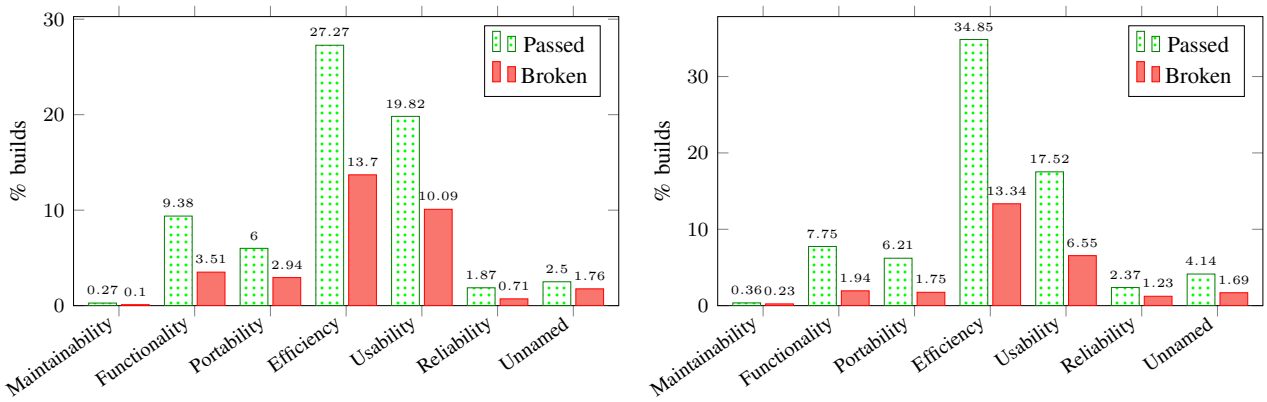
TABLE I
PAIRWISE CHI-SQUARE COMPARISON OF NFRS.

	Portability	Usability	Efficiency	Reliability	Maintainability
Usability	3.399654e-18				
Efficiency	2.919159e-09	2.834015e-06			
Reliability	5.381500e-179	5.705839e-281	1.193036e-260		
Maintainability	4.084665e-29	1.919558e-40	8.737777e-37	0.38027170	
Functionality	0.000000e+00	0.000000e+00	0.000000e+00	0.01699222	0.05420935

(a) Ruby projects.

	Portability	Usability	Efficiency	Reliability	Maintainability
Usability	4.189831e-84				
Efficiency	1.615018e-110	6.077951e-03			
Reliability	3.806572e-191	7.352944e-77	5.284936e-72		
Maintainability	4.564365e-87	1.090704e-39	6.556727e-37	7.532970e-06	
Functionality	5.678315e-12	9.714129e-188	9.305286e-240	7.173664e-288	1.335613e-117

(b) Java projects.



(a) Ruby. (b) Java.

Fig. 2. Passed vs. Broken builds. Figures on bars indicate percentages.

Fig. 2 show, for each NFR and for different programming language (Ruby and Java), the percentage of passed and broken builds. Unnamed indicates the number of builds the approach was not able to classify automatically, as explained in Section III-B. Therefore, we do not consider this category in our statistical analysis.

To test the presence of a significant difference among proportions of builds we perform a Pearson Chi-Square pair-wise test on a contingency table, where columns represent the builds per NFR and rows builds statuses ($H0$: the proportion of builds having different statuses does not change among NFRs). P values < 0.05 were considered statistically significant. Table I shows the P values of paired NFRs.

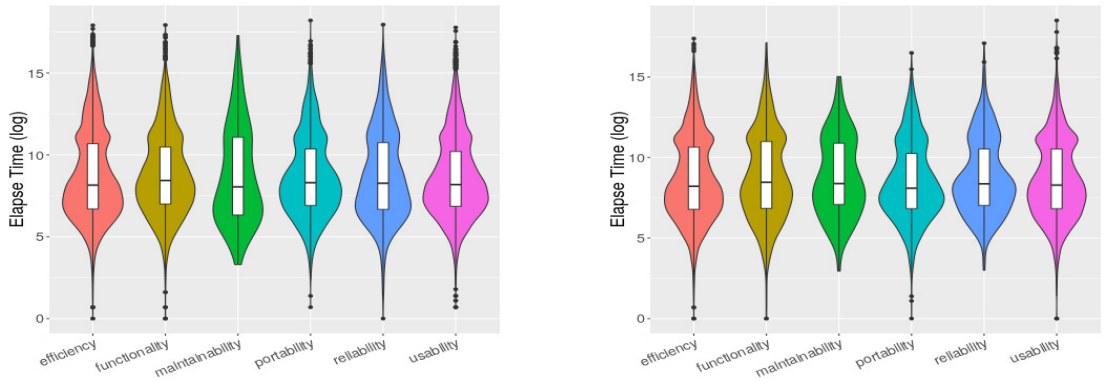
For Ruby projects, analyses revealed significant differences in 10 out of the 15 pairwise comparisons. There are no significant differences between *efficiency* and *portability* or *usability*. The same is observed with *functionality* and *reliability* or *maintainability*. With Java projects, all pair-wise comparisons were significant except between *efficiency* and *usability*, *functionality* and *portability*, and *maintainability* and *reliability*.

RQ2. How long do NFR-related builds remain broken?

Here, we investigate the impact of broken builds considering the time elapsed until the build is fixed. Although is not desirably to face build failures, they play an important role to the development process. For example, a broken build denotes a bug caught earlier [17]. However, since the developers base their work on project branches, if they remain broken for longer times they affect the project's performance.

Table II shows the average time elapsed between a broken build and a sequent passed one group by NFR. Fig. 3 shows the graphical distribution of broken builds for each setting.

Discussion. The goal of RQ1 was to examine whether providing comparison between NFR related builds statuses had an impact on continuous integration builds. The study revealed significant results. For Ruby projects, despite the absolute number of builds related to *efficiency*, it holds the same proportion of passed and broken builds as *usability* group. Together they represent around 70% of the builds. However, RQ2 results exposes that a broken build related to *efficiency* NFR takes 1.6x more time on average to be fixed than a *usability* broken build. We observe similar scenario for



(a) Ruby. (b) Java.

Fig. 3. Graphical distribution of broken builds along the time group by NFR.

TABLE II
AVERAGE DURATION OF BROKEN BUILDS IN MINUTES.

NFR	Ruby	Java
Maintainability	403	37
Functionality	144	58
Portability	121	34
Efficiency	118	40
Usability	73	36
Reliability	191	64
Unnamed	143	19
Total average	170	41

Java. Broken builds from *reliability* group has no significant proportional differences with *maintainability* group, but an issue from the former group takes 1.7x more time on average regarding the last group.

Fig. 3 shows the distribution of the time taken of broken build until a sequent successful build per NFR. Note that the distribution is bimodal along all NFRs. That is, it has two peaks, showing that most broken builds are either fixed within a day (higher peak) or takes around six weeks (lower peak), which is a common release methodology adopted in industry (rapid release cycles).

Our design decisions suggest a set of limitations, many of which we hope to address in future work. We did not measure accuracy of the NFR labeling method. Further, we study the association of builds with only one topic, but there might be cases where they can be linked with multiple topics. Although our approach can be seen as a replication of the work Hindle et al. [6], further evaluation is needed. Finally, we only consider NFRs in our study. Thus, we refrain to only discuss about the relationship of NFRs and builds statuses. Future work could use/propose other classification of builds.

V. CONCLUSION

We examined a large set of projects to expose the relationship between NFR and CI builds statuses. Certain categories of NFR related builds are more prevalent, such as efficiency and usability, regardless if Ruby or Java. So, recommendation systems to help avoiding breakages on those kind of builds would produce overall larger impact on the whole process.

Moreover, maintainability for Ruby projects, and functionality together with reliability for Java, take longer times to be fixed. So, they could be postponed to whenever developers are available to watch the builds, avoiding conflicts among themselves.

ACKNOWLEDGMENT

We thank the Brazilian agencies CAPES, CNPq, and FAPEMIG.

REFERENCES

[1] M. Fowler and M. Foemmel, “Continuous integration,” 2006, accessed 3-February-2017. [Online]. Available: http://www.thoughtworks.com/ContinuousIntegration.pdf
[2] N. Kerzazi, F. Khomh, and B. Adams, “Why do automated builds break? an empirical study,” in *Proc. ICSME*, 2014, pp. 41–50.
[3] B. Adams and S. McIntosh, “Modern release engineering in a nutshell – why researchers should care,” in *Proc. SANER*, 2016, pp. 78–90.
[4] T.-H. Chen, S. W. Thomas, and A. E. Hassan, “A survey on the use of topic models when mining software repositories,” *Empir Softw Eng*, vol. 21, no. 5, pp. 1843–1919, 2016.
[5] L. B. L. De Souza and M. D. A. Maia, “Do software categories impact coupling metrics?” in *Proc. MSR*, 2013, pp. 217–220.
[6] A. Hindle, N. A. Ernst, M. W. Godfrey, and J. Mylopoulos, “Automated topic naming,” *Empir Softw Eng*, vol. 18, no. 6, pp. 1125–1155, 2013.
[7] M. Beller, G. Gousios, and A. Zaidman, “Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration,” in *Proc. MSR*, 2017.
[8] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos, *Non-functional requirements in software engineering*. Springer, 2012, vol. 5.
[9] D. Mairiza, D. Zowghi, and N. Nurmuliani, “An investigation into the notion of non-functional requirements,” in *Proc. SAC*, 2010, pp. 311–317.
[10] ISO/IEC, *ISO 9126. Software engineering – Product quality*, 2001.
[11] J. Eckhardt, A. Vogelsang, and D. M. Fernández, “Are "non-functional" requirements really non-functional?: An investigation of non-functional requirements in practice,” in *Proc. ICSE*, 2016, pp. 832–842.
[12] A. Hindle, M. W. Godfrey, and R. C. Holt, “What’s hot and what’s not: Windowed developer topic analysis,” in *Proc. ICSM*, 2009, pp. 339–348.
[13] D. M. Blei, A. Y. Ng, and M. I. Jordan, “Latent dirichlet allocation,” *J Mach Learn Res*, vol. 3, p. 993–1022, 2003.
[14] A. Hindle, M. W. Godfrey, and R. C. Holt, “Software process recovery using recovered unified process views,” in *Proc. ICSM*, 2010, pp. 1–10.
[15] E. A. Santos and A. Hindle, “Judging a commit by its cover: Correlating commit message entropy with build status on travis-ci,” in *Proc. MSR*, 2016, pp. 504–507.
[16] A. K. McCallum, “Mallet: A machine learning for language toolkit,” 2002.
[17] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, “Usage, costs, and benefits of continuous integration in open-source projects,” in *Proc. ASE*, 2016.