A Tale of CI Build Failures

An Open Source and a Financial Organization Perspective

Vassallo, Carmine; Schermann, Gerald; Zampetti, Fiorella; Romano, Daniele; Leitner, Philipp; Zaidman, Andy; Di Penta, Massimiliano; Panichella, Sebastiano

DOI
10.1109/ICSME.2017.67

Publication date
2017

Document Version
Accepted author manuscript

Published in
Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017

Citation (APA)
Vassallo, C., Schermann, G., Zampetti, F., Romano, D., Leitner, P., Zaidman, A., Di Penta, M., & Panichella, S. (2017). A Tale of CI Build Failures: An Open Source and a Financial Organization Perspective. In *Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017* (pp. 183-193). IEEE. https://doi.org/10.1109/ICSME.2017.67

A Tale of CI Build Failures: an Open Source and a Financial Organization Perspective

Carmine Vassallo*, Gerald Schermann*, Fiorella Zampetti†, Daniele Romano‡,
Philipp Leitner*, Andy Zaidman§, Massimiliano Di Penta†, Sebastiano Panichella*
*University of Zurich, Switzerland, †University of Sannio, Italy
‡ING Nederland, The Netherlands, §Delft University of Technology, The Netherlands

***Abstract*—Continuous Integration (CI) and Continuous Delivery (CD) are widespread in both industrial and open-source software (OSS) projects. Recent research characterized build failures in CI and identified factors potentially correlated to them. However, most observations and findings of previous work are exclusively based on OSS projects or data from a single industrial organization. This paper provides a first attempt to compare the CI processes and occurrences of build failures in 349 Java OSS projects and 418 projects from a financial organization, ING Nederland. Through the analysis of 34,182 failing builds (26% of the total number of observed builds), we derived a taxonomy of failures that affect the observed CI processes. Using cluster analysis, we observed that in some cases OSS and ING projects share similar build failure patterns (e.g., few compilation failures as compared to frequent testing failures), while in other cases completely different patterns emerge. In short, we explain how OSS and ING CI processes exhibit commonalities, yet are substantially different in their design and in the failures they report.**

***Keywords*-Continuous Delivery, Continuous Integration, Agile development, Build failures.**

I. Introduction

Continuous Delivery (CD) is a software engineering practice in which development teams build and deliver new (incremental) versions of a software system in a very short period of time, *e.g.,* a week, a few days, and in extreme cases a few hours [1]. CD advocates claim that the practice reduces the release cycle time (*i.e.,* time required for conceiving a change, implementing it, and getting feedback) and improves overall developer and customer satisfaction [2]. An essential part of a CD process is Continuous Integration (CI), where an automated build process is enacted on dedicated server machines, leading to multiple integrations and releases per day [3]–[6]. One major purpose of CI is to help developers detect integration errors as early as possible. This can be achieved by running testing and analysis tools, reducing the cost and risk of delivering defective changes [4]. Other collateral positive effects introduced by CI in industrial environments are the improvement in developer communication [7] and the increase of their productivity [8]. Consequently, CI has become increasingly popular in software development of both, industrial and OSS projects [6], [9].

At the same time, the study of CI builds has also become a frequent research topic in academic literature. Miller [8] studied a Web Service project in Microsoft and, by observing 69 failed builds, mainly found failures related to code analysis tools (about 40%), but also to unit tests, server, and compilation errors. The latter have been investigated in depth by Seo *et al.* [10] in a study at Google. Recently, Rausch *et al.* [11] studied the Travis CI builds of selected OSS projects, and identified correlations between project and process metrics and broken builds. Other works focused on test failures [5] and on the use of static analysis tools [12]. However, no previous research provides a broad categorization of build failures and compared their occurrences between industry and OSS.

This paper studies build failures in 418 projects (mostly Java-based) from a large financial organization, ING Nederland (*referred to as ING*), as well as in 349 Java-based open source software (OSS) hosted on GitHub and using Travis CI. The purpose of the study is to compare the outcome of CI in OSS and in an industrial organization in the financial domain, and to understand commonalities and differences. As previous work by Ståhl *et al.* [13] suggested that the build process in industry varies substantially, we aim to understand the differences (also in terms of build failure distributions) between OSS and one industrial case. In total we analyzed 3,390 failed builds from ING and 30,792 failed builds from OSS projects. Based on this sample, we address the following research questions:

RQ$_1$: *What types of failures affect builds of OSS and ING projects?*

This research question aims to understand the nature of errors occurring during the build stage in ING and the analyzed OSS projects. We use an open coding procedure to define a comprehensive taxonomy of CI build errors. The resulting taxonomy is made up of 20 categories, and deals not only with the usual activities related to compilation, testing, and static analysis, but also with failures related to packaging, release preparation, deployment, or documentation. Overall, the taxonomy covers the entire CI lifecycle. We then study build failures along the taxonomy, addressing our second research question:

RQ$_2$: *How frequent are the different types of build failures in the observed OSS and ING projects?*

Given the catalog of build failures produced as output of **RQ**$_1$, we then analyze and compare the percentages of build failures of different types for both, ING and OSS projects. Furthermore, based on these percentages, we cluster

projects and discuss characteristics of each cluster, observing in particular whether different clusters contain ING projects only, OSS projects only, or a mix. Finally, we investigate the presence of build failure patterns shared by ING and OSS projects in the various clusters.

Our study shows that ING and OSS projects share important commonalities. For example, both exhibit a relatively low percentage of compilation failures, and, instead, a high percentage of testing failures. However, while unit testing failures are common in OSS projects (as also discussed in [14]), ING projects have a much higher frequency of integration test failures. ING projects also exhibit a high percentage of build failures related to release preparation, as well as packaging and deployment errors. We also found that projects cluster together based on the predominance of build failure types, and some clusters, *e.g.,* related to release preparation or unit testing, only contain or are predominated by ING or OSS projects respectively. In summary, while the behavior of CI in OSS and closed source exhibits some commonalities, closed source has some peculiarities related to how certain activities, such as testing and analysis, are performed, and how software is released and deployed.

II. Study Design and Planning

The *goal* of the study is to investigate the types of build failures that occur in the analyzed OSS and ING projects, how frequently they occur, and to understand the extent to which these failure frequencies differ in the analyzed industrial and OSS projects. This analysis has the *purpose* of understanding the commonalities and differences in the CI process of OSS and of an industrial environment (ING), at least from what one can observe from build failures and from the knowledge of the adopted CI infrastructure.

A. Study Context

The study *context* consists of build failure data from 418 projects (mostly Java-based, as declared by the organization) in ING and 349 Java-based OSS projects hosted on GitHub.

In both, ING and OSS, the CI process is triggered when a developer pushes a change to the Git repository. The code change is detected by the CI server (Jenkins [15] in ING and Travis CI [16] in OSS) and the build stage is started. The outcome of the build process is either a *build failure* or a *build success*.

In ING, if a triggered build succeeds, the generated artifacts of the new version are deployed to a remote server that simulates different environments (*i.e.,* testing, production) to perform further activities (*e.g.,* load, security testing). Indeed ING adopts a well-defined CD pipeline (illustrated in a survey conducted in ING [17]) where the build is only one node of the entire process of an application's release. Some external tools are usually plugged into the build process to augment the actions performed at some steps. For example, SonarQube [18] is used as source code quality inspector, sometimes complemented by PMD [19] and Checkstyle [20].

In our study, we analyzed 12,871 builds belonging to 418 different Maven projects in ING, and mined data from 4 different build servers. Specifically, we extracted, for each project, the Maven logs related to the failed builds that occurred between March 21st, 2014 and October 1st, 2015. The resulting total number of Maven build failures is 3,390 (≈26% of builds). Due to restrictive security policies in the financial domain, the Maven logs were the only resources we could access during the study. Hence, we did not have access to any other resource that would be valuable in order to investigate the nature of build failures in more depth, *e.g.,* data from versioning systems, testing or detailed outputs of static analysis tools (except the data printed to build logs). Therefore, our observations are limited to the information available in build logs.

As for OSS, we selected 349 projects from the *TravisTorrent* dataset [21], which contains build failure data from GitHub projects using Travis CI[1]. We restricted our analysis to all projects of the TravisTorrent dataset using Maven (in order to be consistent with ING projects) and mainly written in Java (according to the dominant repository language reported by GitHub). In addition, we only considered projects having at least one failing build. In total, the 349 projects underwent 116,741 builds, of which 30,792 (≈26%) failed. It is interesting to notice how the percentage of build failures is approximately the same in OSS and ING.

B. Data Extraction Approach

Fig. 1 depicts the research approach we followed to answer our research questions. Specifically, similarly to the work by Désarmeaux *et al.* [22] we focused our analysis on Maven projects and their related build logs. Differently from the work by Désarmeaux *et al.*, we did not analyze build scripts as they were not provided by the involved organization and thus we did not perform a more fine-grained analysis of the kinds of failures.

To extract build failure data for ING, we used the Jenkins REST APIs, which retrieved the log associated to each build and allowed us to reconstruct the history of a job in terms of build failures and build successes. To extract build failure data for OSS, we started from the information contained in the TravisTorrent database. Then, we downloaded build logs for each job IDs related to a build ID labelled as failed, by means of the Travis APIs.

Subsequently, we extracted the error messages contained in each log using a regular expression, *e.g.,* the lines that contain the word "`ERROR`", and used them to define our Build Failure Catalog.

C. Definition of the Build Failure Catalog

Our process for defining the Build Failure Catalog consists of five steps:

Keyword identification. In this step, two authors mined the most relevant keywords associated with an error section. In

[1]Data accessed on 27/10/2016

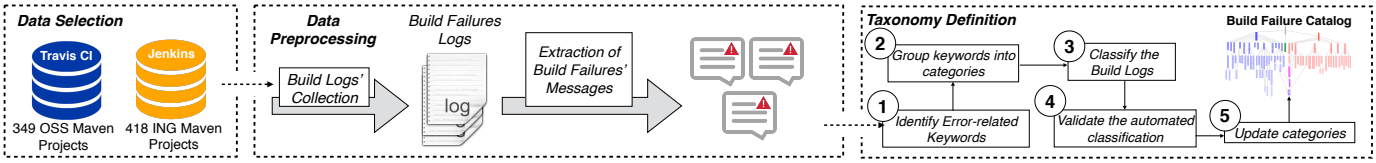

Fig. 1. Data extraction process

Maven, each build fails because of the failure of a specific step (*i.e.,* a goal). As outcome of this failure, an error section is generated in the build log. From each error section, we extracted as keywords only the phrase related to the failed goals (generated by Maven), as well as sentences generated by Maven plugins, making sure of their stability (*i.e.,* no changes in the structure of error messages) during the period of observation. For example, if an error during the execution of the "compilation" goal (*e.g.,"org.apache.maven.plugins:maven-compiler-plugin:compile"*) causes a build failure, the goal name is assumed as keyword. In other cases, when there is no evidence of failed goals in the build logs, we mined other sentences (*e.g.,"Could not resolve dependencies"* in case of missing Maven artifacts).

Keyword grouping. Two of the authors (hereby referred to as A and B) produced a first grouping of keywords into categories. For this step, we used the Maven documentation [23] and the documentation of the respective plugins in order to understand the keywords. Then, two other authors (C and D) reviewed the first categorization and made change suggestions. This process will be explained in more detail in Section III-A1. Finally, A and B checked the resulting catalog again, which led to convergence.

Build log classification. The build logs were automatically classified according to the previously defined catalog. Based on the matched keywords, each build log was assigned to zero, one, or more categories.

Result validation. After the automatic classification, a second, manual classification of a statistically significant sample has been performed to determine the error margin introduced by the automatic classification. Each sample has been validated by two authors and the reliability of the validation was computed using the Cohen k inter-rater agreement statistic [24]. After that, we discussed cases in which the raters disagreed in the classification and in which there was a mismatching with the automatic classification.

Catalog refinement. If, as a result of the manual validation, it turned out that the first automatic classification was erroneous, the manual validation helped to further refine the categories. When categories were refined, a new validation (limited to these categories only) was performed, if necessary.

This five-step procedure was first applied to ING build failures and subsequently to the OSS data set. This means that the two inspectors, starting from the taxonomy derived from the analysis of ING build failures, extended, if necessary, the taxonomy with categories specific to OSS. Regarding the validation of the automatic classification, for ING data we randomly extracted for each identified category a sample size of 764 build failures considering a confidence level of 95% and a confidence interval of $\pm 10\%$. For OSS projects we did not perform a new stratified sampling. Instead, we conducted a complementary validation. We extracted a sample of 377 build failures (which is significant with 95% confidence level and $\pm 5\%$ of confidence interval), making sure the sample contains a number of failures, for each category, proportional to the distribution of failures across categories.

D. Analysis of Build Failure Proportions

In order to investigate the extent to which proportions of build failures of different types vary among ING and OSS, we perform two types of analyses. First, we statistically compared and discussed the frequency of build failures for different types among ING and OSS projects. Then, we used the relative frequencies of different types of failures to cluster projects exhibiting similar distributions of build failures. We used k-means clustering [25] (the *kmeans* function available in the R [26] *stats* package). The k-means function requires to upfront specify the number of clusters k in which items (in our case projects) should be grouped. To determine a suitable value of k, we plotted the *silhouette* statistics [27] for different values of k. Given a document d_i belonging to a cluster C and, $a(d_i)$ the maximum distance of d_i from the cluster's centroid, and $b(d_i)$ from the centroids of other clusters, the silhouette for d_i is given by

$$S(d_i) = \frac{b(d_i) - a(d_i)}{max(a(d_i), b(d_i))}$$

and the overall silhouette of a clustering is given by the mean silhouette across all documents (all projects in our case). Typically, the optimal number of clusters corresponds to the maximum value of the silhouette.

After this clustering step, we analyzed the clusters' content, looking in particular at the extent to which clustering separates ING projects from OSS projects and the occurrences of particular failures patterns.

III. Study Results

This section reports the results of the study we conducted for addressing our research questions.

A. What types of failures affect builds of OSS and ING projects?

In the following, we first report how we obtained the Build Failure Catalog, and then discuss the catalog categories in detail.

1) From Maven phases to the Build Failure Catalog: With the aim of defining a complete and possibly generalizable Build Failure Catalog, we firstly attempted to group build failures into categories closely following the Maven lifecycle phases, *i.e.,* validate, process-resources, compile, test-compile, test, package, integration-test, verify, install, and deploy. When a build failure could not be mapped onto specific Maven phases (*e.g.,* those related to non-functional testing), we created a new category.

A first result was made up of 16 categories, specifically (i) a validation category; (ii) a pre-processing category; (iii) three compilation categories related to compiling production code, compiling test code, and missing dependencies; (iv) three separate categories for unit testing, integration testing, and non-functional testing; (v) a static analysis category; (vi) a packaging category; (vii) an installation category; (viii) a deployment category; (ix) a documentation category; and (x) three crosscutting (*i.e.,* placeable in more than one phase) categories related to testing (where mapping to specific testing activities was not possible), release preparation, and other plugins.

After the first categorization, subsequent iterations produced the following changes:

- *Dependencies* were initially mapped to *Compilation*. Then, we realized that it is not possible to generally map dependencies to the compilation phase (*i.e.,* dependencies related issues impact on several phases). Therefore, we created a crosscutting category for *Dependencies*.
- Some tools, such as *Grunt*, *Cargo*, and *MojoHaus*, were initially grouped as *Other Plugins*. Then, we decided that, due to the nature of the tasks they perform (*e.g.,* allowing the execution of external commands), it was more appropriate to group them together with *Ant* as *External Tasks*.
- The goal "prepare" of the *maven-release-plugin* was initially put into *Validation*. However, we realized that it is not bound exclusively to the "validate" phase, but it covers all phases prior "deploy". Therefore, we placed this goal into the *Release Preparation* category.
- The "installation" phase was considered as a sort of "local" deploy, therefore we created a category named *Deployment* with two sub-categories related to *Local Deployment* and *Remote Deployment*.
- We decided to keep *Clean* separate from *Validation*, as Maven separates the cleaning of all resources generated by the previous build from the default lifecycle. Instead, as Maven provides a dedicated clean lifecycle, we created a specific *Clean* category.
- For some goals (*e.g.,* "migrate" of *org.flywaydb:flyway-maven-plugin*) we decided to create a category *Support*, as they have a supporting nature for the build process. Then we grouped those goals supporting a specific phase of the build into a subcategory of the related category.
- The *Static Analysis* category was renamed into *Code Analysis* and split in two sub-categories, namely *Static Code Analysis* and *Dynamic Code Analysis*.

TABLE I
BUILD FAILURE CATEGORIES.

Category	Subcategory	Description
CLEAN		Cleaning build artifacts
VALIDATION		Check on the project's resources
PRE-PROCESSING (RESOURCES)		Generation and processing of the project's resources
COMPILATION	PRODUCTION	Compilation of production code
	TEST	Compilation of test code
	SUPPORT	Code manipulation & processing
TESTING	UNIT TESTING	Running unit tests
	INTEGRATION TESTING	Running integration tests
	NON-FUNCTIONAL TESTING	Running load Tests
	CROSSCUTTING*	Crosscutting test failures
PACKAGING		Packaging project artifacts
CODE ANALYSIS	STATIC	Code analysis (without executing it)
	DYNAMIC	Code analysis (by executing it)
DEPLOYMENT	LOCAL	Project's artifacts installation
	REMOTE	Project's artifacts deployment to a remote repository
EXTERNAL TASKS		Capability for calling other environments
DOCUMENTATION		Documentation generation and packaging
RELEASE PREPARATION*		Preparation for a release in SCM
SUPPORT*		Database migration and other activities
DEPENDENCIES*		Project's dependencies resolution and management

- We created a category *Testing* hosting test execution involving *Unit testing*, *Integration testing*, and *Non-functional testing*. Due to missing goals we were not able to classify some failed tests into one of these three subcategories, thus we assigned them to the crosscutting subcategory *Crosscutting*.

Once we had adapted our initial categorization as described, we automatically classified the logs, sampled the logs (764 ING and 377 OSS) to be manually validated, and proceeded with the validation. The validation of ING build failures was completed with an inter-rater agreement of $k = 0.8$ (strong agreement) and of OSS build failures with $k = 0.62$ (again, strong agreement). It is particularly important to discuss the 51 cases (6.7%) for which there was a consistent disagreement with the automatic categorization of ING builds. This was related to failures raised by the *org.apache.maven.plugins:maven-surefire-plugin*, initially assigned to the *Unit Testing* category. However, we found that the plugin name was usually followed by a label "(default-test)" or "(integration-testing)", the former suggesting it was indeed unit testing, while the latter pertaining to integration testing. Therefore, we decided to rematch the latter. Since this was just a matter of refining the regular expression, a new validation of the *Unit Testing* and *Integration Testing* was not necessary, but this information was used in the classification of the OSS build failures (the procedure in Fig. 1 was first applied to ING than to OSS).

2) The Build Failure categories: Table I reports the final version of the catalog we devised. It is made up of 20 categories (including sub-categories) and based on 171 keywords[2]. Crosscutting categories are tagged with an asterisk, *e.g., Dependencies**. In the following, we will briefly describe each category, in terms of included goals/keywords and corresponding standard Maven or new phase, with qualitative insights about the logged errors.

Clean. This category includes builds failed while executing the Maven lifecycle goal "org.apache.maven.plugins:maven-

[2]http://www.ifi.uzh.ch/seal/people/vassallo/build_failures_catalog.pdf

clean-plugin:clean"; it tries to remove all files generated during the previous build.

Validation. Our *validation* category is mapped to the Maven *validation* phase that aims to validate a project by verifying that all necessary information to build it is both available and correct. Indeed, the goals included in this category check Maven classpaths (the main purpose of "org.apache.maven.plugins:maven-enforcer-plugin:enforce") or environment constraints, such as Maven and JDK versions.

Pre-Processing. This category contains all failed goals related to the generation of additional resources typically included in the final package (corresponding to Maven phases *process-resources* and *generate-resources*). For example, the goal "org.apache.maven.plugins:maven-plugin-plugin:helpmojo" generates a "HelpMojo" class, corresponding to an executable Maven goal used in the subsequent steps of the build process. Other goals, such as "com.simpligility.maven.plugins:android-maven-plugin:generate-sources", generate source code (in this case R.java) based on the resource configuration parameters.

Compilation. Build failures in this category are mainly caused by errors during the compilation of the production and test code (respectively *compile* and *test-compile* in the Maven lifecycle). This leads to two sub-categories: (i) failures related to the compilation of production code, and (ii) failures related to the compilation of test code.

Production. The compilation of production code can fail due to typical programming errors that are detected by the compiler while running goals such as "org.apache.maven.plugins:maven-compiler-plugin:compile", but also because of language constructs unsupported by the build environment.

Test. Test code compilation failures are similar to production code ones, although we noticed many failures due to wrong exception handling in test code (*e.g.,* "org.apache.maven.plugins:maven-compiler-plugin:testCompile" failed because "unreported exception must be caught or declared to be thrown").

Support. In addition to the previous compilation sub-categories, we added another one related to failures occurring during activities complementary to standard compilation. Examples of these activities are represented by the goal "org.bsc.maven:maven-processor-plugin:process" that processes annotations for jdk6, or the goal "net.alchim31.maven:yuicompressor-maven-plugin:compress" that performs a compression of static files.

Testing. This category includes the execution of unit, integration, and non-functional system tests. Moreover, we identified multiple failed builds for which we were not able to classify them into one of these three sub-categories, hence the crosscutting category *Crosscutting*.

Unit Testing. The Maven phase *test* directly corresponds to this category. Builds fail while executing goals such as "org.apache.maven.plugins:maven-surefire-plugin:test". Those failures are related to the presence of failing test cases. Also other issues raised, *e.g.,* the goal execution fails with an error message that specifies the presence of unit tests which invoke `System.exit()`, or a crashing virtual machine.

Integration Testing. Integration test results are typically verified by means of the *Failsafe* Maven plug-in, which has a goal "integration-test" producing the error message "There are test failures" in case that tests fail. This category corresponds to the Maven phases *pre-integration-test* and *integration-test*. Moreover, we found that integration testing is often performed using the goal "test" of the *Surefire* plugin, even if the latter is mainly intended to be used to execute unit tests.

Non-Functional System Testing. While there is no corresponding Maven phase, the single failing goal in this category is "io.gatling:gatling-maven-plugin:execute". This goal launches *Gatling*, a load testing tool, that keeps track of load testing results across builds. As Gatling accepts *Scala* code, some build failures have occurred because of incompatibilities between Gatling and specific Scala versions, *e.g.,* Gatling 2.1 and Scala versions prior 2.11.

Crosscutting Tests. There are testing failures that we could not assign to a specific category because builds ended without reporting the failed goal. The reported message ("There are test failures") highlights the presence of test failures and is very similar to the one that occurred for unit and integration testing. Moreover, such test failures could occur within various testing related phases of the build process.

Code Analysis. Similar to *Non-Functional System Testing*, we could not identify a Maven phase related to the failed goals of the *Code Analysis* category. Failed goals in this category can be subdivided into *Static* and *Dynamic*.

Static. Static code analysis [28] within the build process is conducted by running goals such as "org.codehaus.mojo:sonar-maven-plugin:sonar", which launches the analysis of source code metrics via SonarQube, and "org.codehaus.mojo:findbugs-maven-plugin:findbugs", which is used to inspect Java bytecode for occurrences of bug patterns via FindBugs.

Dynamic. Dynamic code analysis is performed by executing goals such as "org.codehaus.mojo:cobertura-maven-plugin:instrument", which instruments the classes for the measurement of test coverage (with Cobertura).

Many goals included in the *Code Analysis* category failed because of (failed) quality checks, or in case of SonarQube, because of connection timeouts, e.g., "server can not be reached".

Packaging. This category concerns all the builds failed while bundling the compiled code into a distributable format, such as a JAR, WAR, or EAR (Maven *prepare-package* and *package* phases). This category includes goals such as "org.apache.maven.plugins:maven-war-plugin:war". There are several errors underlying these failed goals, such as the presence of a wrong path pointing to a descriptor file or non-existing files (*e.g.,* "The specified web.xml file does not exist").

Deployment. We observed two types of deployment: local and remote.

Local. This sub-category is mainly related to the *in-*

stall phase of the standard Maven lifecycle, in which the build adds artifacts to the local repository (*e.g.,* by "org.apache.maven.plugins:maven-install-plugin:install"). To this end, the build process, using the information stored in the POM file, tries to determine the location for the artifact within the local repository. Failures concern the impossibility to find and parse the needed configuration data.

Remote. The sub-category *Remote* corresponds to the Maven phase *deploy* and includes goals, e.g., "org.apache.maven.plugins:maven-deploy-plugin:deploy", which try to install artifacts in the remote repository. Failures are often due to wrong server URLs and authentication credentials. Other cases include the unsuitability of the specified repository for deployment of the artifact or a not-allowed redeployment of the same artifact.

External Tasks. This category includes failures caused by the usage of external tools scheduled to execute within the build process. Some failures are related to the execution of Ant tasks (e.g., "org.apache.maven.plugins:maven-antrun-plugin:run") or SQL statements (e.g., "org.codehaus.mojo:sql-maven-plugin:execute"). We also added goals to this category that have the task of manipulating application containers and allowing the execution of external Java programs (e.g., "org.codehaus.mojo:exec-maven-plugin:exec") from a POM file. Many errors reported by these goals are related to timeout problems (e.g., "Execution start-container:start failed: Server did not start after 120000 milliseconds"), but there are also failures related to erroneous environment specifications, e.g., wrong port numbers.

Documentation. Documentation is mapped onto the phase *site* of the Maven Site lifecyle. It concerns build failures occurring during the generation of documentation, *e.g.,* using the Javadoc tool, through goals such as "org.apache.maven.plugins:maven-javadoc-plugin:jar". Reasons for those failures include the specification of wrong target directories, incompatibilities between the JDK and the Javadoc generation tool, or syntax errors in the Javadoc comments. Moreover, failures are caused by the inability to create an archive file from the previously generated Javadocs ("Error while creating archive").

Crosscutting Categories. We will now discuss failure categories that can not be associated with one specific phase, but can rather be mapped onto several phases.

Release Preparation. This category concerns failures occurring during the preparation for the deployment of a packaged release. We included in this category the goal "org.apache.maven.plugins:maven-release-plugin:prepare", that is used to (i) check the information regarding the current location of the project's Source Configuration Management (SCM) and whether (ii) there are no uncommitted changes in the current workspace. There are several reasons for failures in these tasks, including failed executions of SCM-related commands (*e.g.,* Git commands), or the presence of uncommitted changes. In many cases, failed builds simply report the name of the failed goal without specifying additional information.

Support. Builds fail while executing tasks that are not scheduled to execute within an usual build process. This category includes goals such as "org.flywaydb:flyway-maven-plugin:migrate", which is used for database schema migration, or "com.google.code.sortpom:maven-sortpom-plugin:sort" that helps the user sorting the underlying POM file. As they are not used in a regular build process, their categorization into a specific phase (e.g., *Support* of *Compilation*) is difficult.

Dependencies. This category contains goals such as "list" and "copy" of the "org.apache.maven.plugins:maven-dependency-plugin" plugin. We also put there the keywords "Could not resolve dependencies" and "Failed to resolve classpath resource", which are used to catch dependency-related failures when there is no further information about the failed goal. Typical errors occurring in this category are invalid resource configurations in the POM file, or failed downloads due to unavailable artifacts. As is the case for all crosscutting categories, this category of failures can occur in each phase of the build process. It is possible that, in order to execute the test phase or a static analysis check, the build process needs to use (and possibly download) the proper plugins via dependency resolution.

Summary of RQ_1: Our build failure catalog comprises 20 categories. 3 are related to Compilation, and 6 to specific Testing and Code Analysis activities. Release preparation and Deployment represent 2 separate categories, and the remaining are related to other build activities (*e.g.,* Packaging).

B. How frequent are the different types of build failures in the observed OSS and ING projects?

Percentage of failing builds. For both the OSS and ING analyzed projects, the overall percentage of failing builds during the period of observation is 26%. In contrast, Kerzazi *et al.* [29] and Miller [8] observed lower percentages of 17.9% and 13% respectively; Seo *et al.*'s study at Google revealed a rate of 35% failing builds.

Fig. 2 provides, for both ING (blue bar) and OSS projects (yellow bar), a break-down of the build failures across the (sub-)categories identified in $\mathbf{RQ}_1$. Note that we performed this classification on $3,390-779=2,611$ ING and $30,792-9,774=21,018$ OSS build failure logs for which we were able to mine sentences to support the identification of the causes of failures (*e.g.,* failed goal or any of the identified regular expressions). In 779 cases for ING and in 9,774 cases for the OSS projects, we were not able to classify the failure based on the information contained in the log. However, the percentages in Fig. 2 are related to the original set of failures (including non-classified build failures).

By observing Fig. 2, we can notice that the percentage of *compilation*-related failures is fairly limited, for both *production* code (4.2% ING and 7.1% OSS) and *test* code (2.3% ING and 1.8% OSS). This is below the 26% observed by Miller [8]. In the case of ING, one possible reason is that — as also confirmed by a survey conducted within ING [17] — private builds (*i.e.,* builds executed on the developer's machine) are

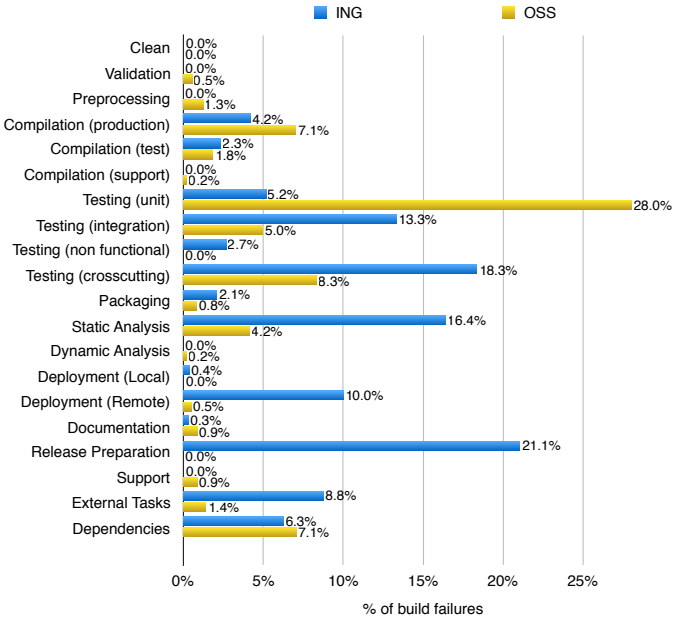

Fig. 2. Percentage of build failures for each category.

used to limit the number of build failures due to compilation errors.

Interesting considerations can be drawn for build failure categories pertaining to testing. The *unit testing* category is the one for which we can notice, on the one hand, the highest percentage of build failures for OSS (28%, perfectly in line with the percentage observed by Miller [8] at Microsoft), and, at the other hand, a large difference with respect to the relatively low percentage (5.2%) observed for ING projects. Conversely, we observe a relatively high percentage of *integration testing* failures in ING (13.3%) and only 5.0% for OSS. When we compare this insight with the one reported above about unit testing, we can a confirm the findings of Orellana *et al.* [30] indicating that in OSS projects unit testing failures dominate integration testing failures. We can also see how ING projects are affected more by integration testing than unit testing failures. Previous research [17] indicates that unit tests are often executed within private builds in ING, therefore the number of remaining failures discovered on the CI server turns out to be fairly limited. We found no evidence of *non-functional testing* failures in OSS projects, while there is a relatively low percentage of them (2.7%) in ING. This low percentage of failures was surprising, as the CD pipeline used in ING explicitly perform non-functional testing (including load testing) at a dedicated stage (*i.e.,* not during the build, as reported in a survey conducted in ING [17]). We deducted that in ING a preliminary load testing is performed at the build stage (by means of Gatling [31]), with the aim of an early and incremental discovery of bottlenecks. Finally, we found a relatively high number of *crosscutting* test failures (18.3% for ING and 8.3% for OSS) not attributable to specific testing activities.

Static analysis tools are also responsible for a relatively high percentage (16.4%) of build failures in ING; Miller [8] observed a substantially higher percentage of 40%, without discussing specific reasons. In contrast, for our OSS projects, the percentage is 4.2%, similar to the observation of Zampetti *et al.* [12] who found percentages almost always below 6%. One may wonder whether the higher percentage of static analysis related failures in ING are caused by stricter quality checks. We have no evidence of this, as we had no access to the SonarQube entries related to the build failures. Instead, OSS projects used static analysis tools on the CI server (without running them on a dedicate server as in ING) and the results are usually visible in the build logs. For this reason, a manual scrutiny of some build failure logs confirmed that, in such cases, build failures were indeed due to specific warnings raised by static analysis tools.

For ING projects we perceive a high percentage of *release preparation* problems (21.1% of the total). This can be attributed to the way ING handles the deployment of a new application: it relies on the standard Maven process instead of using a combination of different goals, as it is common in OSS projects. Also, for *deployment*, the percentage of build failures that occurred in ING (10.0%, higher than the 6% that Miller reported [8]) is much higher than for OSS (0.5%). As discussed in Section III-A2, such failures are mostly due to mis-configurations for accessing servers (*i.e.,* wrong server's IP address) hosting the application artifacts. These mis-configurations could potentially be very costly. Indeed, earlier research in the area of storage systems [32] has shown that 16.1%–47.3% of mis-configurations lead to systems becoming either fully unavailable or suffering from severe performance degradations. The *packaging* category exhibits a relatively low percentage of build failures, but still much more frequent in ING (2.1%) than for OSS (0.8%). In summary, release preparation, packaging and deployment have quite different trends in ING and OSS. In ING, the CD machinery is massively used to produce project releases and deploy them on servers, and in particular on servers where further quality assurance activities (*i.e.,* load and security testing) occur before the release goes in production. Instead, we assume that in most of the studied OSS projects this rarely happens (but it is not excluded), as the main goal of CI is to make a new release available for download on GitHub.

External tasks are responsible for respectively 8.8% (ING) and 1.4% (OSS) of the build failures. The use of external tasks (*e.g.,* the execution of Ant tasks in a Maven build) might make the build process more complex and possibly difficult to maintain, *e.g.,* when one has to maintain both Maven and Ant scripts. While there is no evidence that build maintenance is related to the increase of build failures, it is a phenomenon to keep into account, *e.g.,* by planning, whenever possible, build restructuring activities.

Dependency-related failures exhibit similar percentages for ING and OSS (6.3% and 7.1% respectively). Finally, we found a very small failure percentage ($< 1.5\%$) for other categories, such as clean, validation, pre-processing, compilation support, documentation, and support (crosscutting) both in the case of

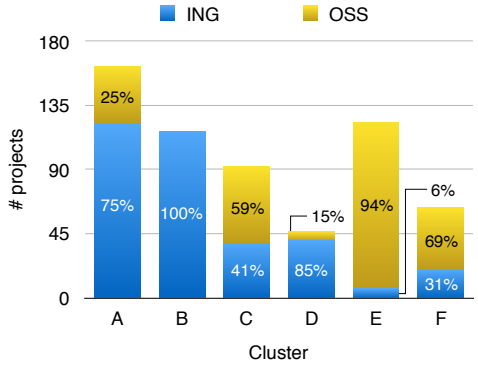

Fig. 3. Composition of projects clusters.

TABLE II
% OF BUILD FAILURES (CLUSTER MEAN POINT).

Class	A	B	C	D	E	F
CLEAN	0%	0%	0%	0.02%	0.01%	0%
VALIDATION	1.46%	0%	0.78%	0%	0.23%	0.38%
PRE-PROCESSING	0.73%	0%	8.62%	0%	0.49%	0.22%
COMPILATION	6.06%	2.79%	9.57%	1.42%	12.07%	**74%**
UNIT TESTING	4.05%	3.02%	7.16%	0.67%	**72.81%**	15.12%
INTEGRATION TESTING	7.52%	0.54%	2.03%	0.06%	0.70%	0.28%
NON FUNCTIONAL TESTING	0%	0%	9%	0%	0%	0%
CROSSCUTTING TESTING	2.75%	0.19%	2.5%	**93.54%**	0.42%	1.53%
PACKAGING	1.16%	2.93%	5.88%	1.10%	1.71%	0.87%
CODE ANALYSIS	**32.12%**	0.24%	4.59%	0.64%	1.64%	1.03%
DEPLOYMENT	**22.2%**	5.01%	1.93%	0.79%	1.96%	0.83%
EXTERNAL TASKS	**14.29%**	3.77%	0.88%	0.33%	1.02%	0.14%
DOCUMENTATION	1.75%	0%	1.23%	0%	0.98%	1.21%
RELEASE PREPARATION	3.66%	**80.11%**	3.15%	1.18%	0.26%	0.23%
SUPPORT	0.57%	0.17%	3.80%	0%	0.63%	0.08%
DEPENDENCIES	1.67%	1.23%	**38.85%**	0.22%	5.07%	4.09%

ING and OSS projects.

Clustering based on build failure percentages. As explained in Section II, to better investigate the differences in the distribution of build failures in ING and OSS projects, we clustered the whole set of projects (both OSS and ING) using the *K-means* algorithm [33]. While the first analysis of Fig. 2 provides a broad overview of build failure distribution in the entire context (ING or OSS), the clusters help to understand the extent to which there are projects (within ING, within OSS, or in both environments) exhibiting certain distributions of build failures and, in general, to investigate the build failure distribution on a single project level.

We modelled the projects as vectors where each dimension is a category of our taxonomy (Table I). Note that only for Testing we considered the sub-categories as dimensions, because they lead to a significantly different distribution of build failures. Then we computed for each project the percentage of its build failures belonging to each category, and assigned values to the related vector dimension. Finally, we applied the *K-means* algorithm several times in order to find the optimal value of Silhouette statistics.

Fig. 3 depicts the clusters' composition for $k = 6$ (where k is the number of clusters), which corresponds to the optimal value of Silhouette. The clusters can be interpreted by looking at Table II, showing for each cluster the percentage of build failures of its mean point (as computed by the k-means algorithm).

Three main clusters can be observed: A, B and E (containing 27%, 19% and 20% of the projects respectively). All projects in Cluster B come from ING, while Cluster E includes almost exclusively OSS projects. Cluster A contains again a high percentage of ING projects (75%).

Cluster A includes projects that fail mostly because of Code Analysis (32%), Deployment (22%) and External Tasks (14%), while B exhibits mainly build failures belonging to Release Preparation (80%). The last result is not surprising since, as shown in Fig. 2, only ING builds are affected by this type of failures.

Cluster E contains projects that typically fail because of Unit Testing (on average the 73% of the build failures belongs to this category) and this justifies that within the cluster we mainly found OSS projects.

Cluster C is balanced (59% OSS projects vs. 41% ING projects) and contains projects mainly exhibiting dependency failures, while the projects in Cluster D are mainly ING and exhibits mostly Crosscutting Testing build failures for which we cannot specify the type of testing involved.

Cluster F is relatively small (64 projects) and catches only projects that usually fail for compilation errors (74%).

In summary, the clustering analysis highlights groups of projects exhibiting similar characteristics in terms of build failure distributions. Specifically, some clusters are mainly constituted by ING (only) or OSS projects, as certain failures dominate in one case or the other, while there are also some clusters that are almost equally distributed. Also, we noticed how some clusters group together projects mainly exhibiting one specific kind of build failure.

Summary of RQ$_2$: The OSS and ING projects exhibit a different distribution of build failure types. Overall, in OSS projects build failures happen mainly due to unit testing failures, while ING projects fail, above all, because of release preparation failures. Finally, the clustering analysis of the projects (based on the distribution of build failure types) points out how projects from the two different contexts in some cases spread out into different clusters.

IV. DISCUSSION

In this section we discuss the main findings and their implications for future research. Specifically, given the differences and commonalities that we observed in RQ$_2$, some CI practices deserve additional investigation.

Compilation errors are typically fixed in private builds. Compilation failures are considered to be particularly relevant during the build process, such that the study of Seo *et al.* [10] conducted at Google focused entirely on that. Our study reports a small (6.5% in ING and 9% in OSS), yet non-negligible percentage of compilation-related build failures. On the one hand, our study confirms that compilation success is a key prerequisite for code promotion (as stated in [17]), and therefore mostly achieved in private builds. On the other hand, even such low percentage highlight how CI can still be beneficial to spot compilation errors due to the adoption of anti-patterns – *e.g.,* when a change is pushed without compiling it – or due to the usage of different development

environments, *e.g.,* incompatibility between the JDK versions installed on the CI server and the local machine.

OSS projects run every test on CI servers, ING mostly integration testing. Integration testing failures are more frequent in ING than in OSS. Instead, and consistently with a previous study by Orellana Cordero *et al.* [14], OSS projects exhibit more unit testing related failures. This indicates a different distribution of testing activities across the development process in the open source and industrial context. In OSS projects, developers often rely directly on the CI server to perform testing (as the very high number of unit testing failures may suggest). In ING, a conservative strategy is preferred, revealing unit testing failures before pushing changes on the server (by running unit tests on local machine), and referring to the build server mostly to verify the correct integration of the changes made by different developers. This is also confirmed by results of a previous survey with ING DevOps [17].

Early discovery of non-functional failures in ING. For large and business-critical projects like the one used by ING for their online banking, appropriate non-functional system testing is crucial. As explained in Section II, this is mostly done offline, on a separate node of the CD pipeline. Nevertheless, as our study shows, developers in ING rely on the build process to spot, whenever possible, non-functional issues, and specifically load test failures. While this happens in a relatively small percentage of the observed build failures (2.1%), it suggests that the early discovery of some problems during the CD process (*i.e.,* as soon as a change is pushed) could save time to solve some performance bottlenecks that would otherwise be discovered at a later stage only. We have no evidence of this activity in the OSS builds.

Release preparation and deployment failures are very common in industry, less so in OSS. We noticed how release preparation errors are very frequent in ING (21.2% of our build failures are associated to this category). At the same time, we did not find such evidence in OSS projects. In ING, the CI process is built using fewer steps (*i.e.,* Maven goals). Developers prefer to use (when possible) predefined bundled steps (as in "prepare" goal, that "covers" several default Maven lifecycle stages), instead of adopting a combination of different goals (each covering a single stage) at least for the most critical stages of the build process (*i.e.,* all stages before the deploy of a new release). Deployment errors are also conspicuous (10%), and mostly due to mis-configurations. For OSS projects, in most of the cases, there is no real deployment of a new release, it is just a matter of making the new release available on GitHub, and using CI for assessing and improving its quality.

Static Analysis (SA) tools: on CI server in OSS, remotely in ING. Our results indicate an intensive percentage of failures related to static analysis tools in ING (16.4%) compared to OSS (4.2%). Looking at the failed goals in the build process of projects in both contexts, we noticed how OSS developers prefer to run SA tools on the build server while in ING SA tools are run on a different server (via SonarQube). This choice did not necessarily lead to less build failures (we noticed more static analysis related failures in ING), but it is an indication about the willingness of ING to have i) well collected data, easy to query and monitor, and ii) a separate analysis of code smells without overloading the CI server.

V. Threats to Validity

Threats to *construct validity* concern the relationship between theory and observation. The most important threat of this type in our study is due to our limited observability of the ING build failures (*i.e.,* we had to rely on build failure logs only). For example, we could not perform a fine-grained classification of some testing or static analysis failures whose nature was only visible from separate reports.

Another threat is due to possible mis-classification of build failures, given that the classification is only based on failed build goals and, in some cases, on keyword matching. As explained in Section II, we mitigated this threat by performing a manual validation on a statistically significant sample of ING build failures, mainly aimed at verifying that such a matching was correctly performed, and reported results of such a validation. We checked whether the agreement of such a validation was due to chance by using the Cohen k inter-rater agreement. As for OSS failures, we repeated the process, however on only 377 build failures in total, as it was not a complete new validation of the taxonomy, but rather a check of its validity when applied to OSS projects.

Threats to *internal validity* concern internal factors of our study that could influence our results. The most important threat of this class is related to the subjectiveness likely introduced when devising the build failure catalog. We limited such a subjectiveness in different ways, *i.e.,* by performing multiple iterations conducted by different authors independently, and by using the Maven standard lifecycle as a roadmap for creating such a catalog. For ING projects, we could not observe and control the projects' programming languages (although an ING team leader reported us that are mostly written in Java). Another threat is related to the choice of the number of clusters in $\mathbf{RQ}_2$. We mitigated this threat by using the silhouette statistic [27] to support our choice. Furthermore, while the number of OSS and closed-source projects is comparable, the number of build failures for closed-source is one order of magnitude greater than for OSS, and this sample imbalance could have affected our results.

Threats to *external validity* concern the generalization of our findings. On the one hand, results related to our closed-source sample are necessarily specific to ING. It is possible that these results partially generalize to other organizations in the financial domain, but this has not been specifically validated in our work. In Section IV, we discussed in how many cases our findings are consistent with those of previous studies, and also in which cases we obtained different results. OSS results, on the other hand, are representatives of OSS Java-based projects using Maven and relying on the Travis-CI infrastructure. In some cases (*e.g.,* for testing), we found confirmations of results from previous studies. Our study does not necessarily generalize to applications built using other languages, build

scripts, and different CI infrastructures. Finally, it is possible that the different domain of closed-source projects and open source projects could have impacted our results.

VI. Related Work

Ståhl *et al.* report that CI is becoming increasingly popular in software development [6]. With this continued uptake in industry, researchers have also focused more on build systems, and more specifically on build failures.

Build Failures. Miller's seminal work [8] on build failures in Microsoft projects describes how 66 build failures were categorized into compilation, unit testing, static analysis, and server failures. Our observation is larger (18 months vs 100 days, 13k builds vs 515, and 3,390 build failures vs 66), and describes a more detailed categorization of build failures, but covers a different domain. Rausch *et al.* [11] studied build failures in 14 popular open source Java projects, finding that most of the failures ($> 80\%$) are related to failed test cases, and that there is a non-negligible portion of errors due to Git interaction errors. Seo *et al.* [10] conducted a study focusing on the compiler errors that occur in the build process. They devised a taxonomy of compilation errors that lead to build failures in *Java* and *C++* environments. Kerzazi *et al.* [29] analyzed $3,214$ builds in a large software company over a period of 6 months to investigate the impact of build failures. They observed a high percentage of build failures (17.9%) that brings a potential cost of about $2,035$ man-hours considering that each failure needs one hour of work to succeed.

Beller *et al.* [5] focused on testing with an in-depth analysis of 1,359 projects using both *Java* and *Ruby* programming languages. Testing is the main activity responsible for failing builds (59% of build failures during test phase for Java projects). While our results confirm their finding, we also highlight the importance of failures due to other tasks. Orellana Cordero *et al.* [14] studied test-related build failures in OSS projects. They identified that unit test failures dominate integration test failures. Our results for OSS projects are similar, yet our findings for ING projects are the opposite. This may be due to more intensive usage of private builds to deal with unit tests in an industrial environment, which is not the case for the OSS projects studied by Orellana Cordero *et al.* [14]. Zampetti *et al.* [12] looked at build failures (mostly related to adherence to coding guidelines) produced by static analysis tools in Java-based OSS projects. Our work found a high percentage of build failures due to static analysis (for ING projects), although in most cases due to infrastructure (Jenkins and SonarQube) misconfiguration.

Build Activities. McIntosh *et al.* [34] studied the relationship between changes to production code, test code, and the build system. They noticed that a strong relation between changes made at all three levels. McIntosh *et al.* [35] studied version histories of 10 systems to measure the overhead that build system maintenance imposes on developers. Finally, Desarmeaux *et al.* [22] investigated how build maintenance effort is distributed across the build lifecycle phases of systems built through Maven. They observed that the compile phase requires most maintenance activity.

Continuous Integration (CI) and Continuous Delivery (CD). Hilton *et al.* [9] investigated why developers use or do not use CI, concluding that this concept has become increasingly popular in OSS projects and [36] present a qualitative study of the barriers and needs developers face when using CI. Ståhl *et al.* [13] noticed that there is no homogeneous CI practice in industry. They identified that there are many variation points in the usage of the CI term. Not only CI practices vary between different industries, we also identified noticeable differences between OSS and industry projects. Vassallo [17] investigated CD practices in ING focusing attention on how they impact the development process and the management of technical debt. Conversely, Savor *et al.* [37] reported on an empirical study conducted in two high-profile Internet companies. They noticed that the adoption of CD does not limit the scalability in terms of productivity of one organization even if the system grows in size and complexity. Finally, Schermann *et al.* [38] derived a model based on the trade-off between release confidence and the velocity of releases. Schermann *et al.* [39] investigated the principles and practices that govern CD adoption in industry and concluded, amongst others, that architectural issues are one of the main barriers for CD adoption.

VII. Conclusion

This paper investigates the nature and distribution of Continuous Integration (CI) build failures occurring in 418 Java-based projects from ING (an organization in the financial domain), and in 349 Java-based OSS projects hosted on GitHub and using Travis CI as CI infrastructure. The results of our study highlight how OSS and ING projects exhibit substantially different distributions of build failure types, confirming but also contradicting some of the findings of previous research which are based on OSS projects' data or only data from a single industrial organization. Our findings are important for both researchers and practitioners since they shed some more light on the differences and commonalities of CI processes adopted in the analyzed OSS projects and the observed industrial organization highlighting interesting build failure patterns.

Work-in-progress aims at replicating the study in other industrial environments and further open source projects, and at performing a deeper analysis on the build failures observed, *e.g.,* studying the difficulty in fixing different kinds of problems. Additionally, we plan to use our results to aid developers to properly maintain build process pipelines to make it more efficient, *e.g.,* by deciding what to do in private builds on the developer's local machine and what to delegate to CI servers, or how to mitigate problems by conceiving approaches able to automate their resolution.

Acknowledgment

The authors would like to thank developers from ING that provided precious inputs during this study.

REFERENCES

[1] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st ed., 2010.

[2] L. Chen, "Continuous delivery: Huge benefits, but challenges too," *IEEE Software*, vol. 32, no. 2, pp. 50–54, 2015.

[3] M. Fowler and M. Foemmel, *Continuous Integration*. 2016.

[4] P. Duvall, S. M. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.

[5] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of travis ci with github," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pp. 356–367, ACM, 2017.

[6] D. Ståhl and J. Bosch, "Modeling continuous integration practice differences in industry software development," *J. Syst. Softw.*, vol. 87, pp. 48–59, Jan. 2014.

[7] J. Downs, B. Plimmer, and J. G. Hosking, "Ambient awareness of build status in collocated software teams," in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 507–517, 2012.

[8] A. Miller, "A hundred days of continuous integration," in *Proceedings of the Agile 2008*, AGILE '08, pp. 289–293, 2008.

[9] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "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)*, pp. 426–437, 2016.

[10] H. Seo, C. Sadowski, S. G. Elbaum, E. Aftandilian, and R. W. Bowdidge, "Programmers' build errors: a case study (at Google)," in *Proc. Int'l Conf on Software Engineering (ICSE)*, pp. 724–734, 2014.

[11] T. Rausch, W. Hummer, P. Leitner, and S. Schulte, "An empirical analysis of build failures in the continuous integration workflows of java-based open-source software," in *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pp. 345–355, 2017.

[12] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. D. Penta, "How open source projects use static code analysis tools in continuous integration pipelines," in *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pp. 334–344, 2017.

[13] D. Ståhl and J. Bosch, "Automated software integration flows in industry: A multiple-case study," in *Companion Proc. Int'l Conf. on Software Engineering (ICSE Companion)*, pp. 54–63, 2014.

[14] G. Orellana, G. Laghari, A. Murgia, and S. Demeyer, "On the differences between unit and integration testing in the travistorrent dataset," in *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pp. 451–454, 2017.

[15] "Jenkins. https://jenkins.io (last access 05.04.2017)."

[16] "Travis-CI. https://travis-ci.org (last access 05.04.2017)."

[17] C. Vassallo, F. Zampetti, D. Romano, M. Beller, A. Panichella, M. Di Penta, and A. Zaidman, "Continuous delivery practices in a large financial organization," in *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 41–50, 2016.

[18] "SonarQube. http://www.sonarqube.org (last access 05.04.2017)."

[19] "PMD. https://pmd.github.io/ (last access 05.04.2017)."

[20] "Checkstyle. http://checkstyle.sourceforge.net (last access 05.04.2017)."

[21] M. Beller, G. Gousios, and A. Zaidman, "TravisTorrent: Synthesizing Travis CI and GitHub for full-stack research on continuous integration," in *Proceedings of the working conference on mining software repositories (MSR)*, pp. 447–450, 2017.

[22] C. Désarmeaux, A. Pecatikov, and S. McIntosh, "The dispersion of build maintenance activity across maven lifecycle phases," in *Proc. Int'l Conference on Mining Software Repositories (MSR)*, pp. 492–495, 2016.

[23] "Maven. http://maven.apache.org/plugins/index.html (last access 05.04.2017)."

[24] J. Cohen, "A coefficient of agreement for nominal scales," *Educ Psychol Meas.*, vol. 20, pp. 37–46, 1960.

[25] J. A. Hartigan and M. A. Wong, "A k-means clustering algorithm," *Applied Statistics*, vol. 28, p. 100108, 1979.

[26] R Core Team, *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2012. ISBN 3-900051-07-0.

[27] J. Kogan, *Introduction to Clustering Large and High-Dimensional Data*. New York, NY, USA: Cambridge University Press, 2007.

[28] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 470–481, 2016.

[29] N. Kerzazi, F. Khomh, and B. Adams, "Why do automated builds break? an empirical study," in *30th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 41–50, IEEE, 2014.

[30] A. M. Manuel Gerardo Orellana Cordero, Gulsher Laghari and S. Demeyer, "On the differences between unit and integration testing in the travistorrent dataset," in *Proceedings of the 14th working conference on mining software repositories*, 2017.

[31] "Gatling. http://gatling.io/#/ (last access 05.04.2017)."

[32] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pp. 159–172, 2011.

[33] J. Macqueen, "Some methods for classification and analysis of multivariate observations," in *In 5-th Berkeley Symposium on Mathematical Statistics and Probability*, pp. 281–297, 1967.

[34] S. Mcintosh, B. Adams, M. Nagappan, and A. E. Hassan, "Mining co-change information to understand when build changes are necessary," in *Proc. Int'l Conf on Software Maintenance and Evolution (ICSME)*, pp. 241–250, IEEE, 2014.

[35] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," in *Proceedings of the Int'l Conference on Software Engineering (ICSE)*, pp. 141–150, 2011.

[36] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "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*, p. To Appear, 2017.

[37] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm, "Continuous deployment at facebook and OANDA," in *Companion proceedings of the 38th International Conference on Software Engineering (ICSE Companion)*, pp. 21–30, 2016.

[38] G. Schermann, J. Cito, P. Leitner, and H. C. Gall, "Towards quality gates in continuous delivery and deployment," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pp. 1–4, May 2016.

[39] G. Schermann, J. Cito, P. Leitner, U. Zdun, and H. Gall, "An empirical study on principles and practices of continuous delivery and deployment." PeerJ Preprints 4:e1889v1 https://doi.org/10.7287/peerj.preprints.1889v1, 2016.