Why do Automated Builds Break? An Empirical Study

Noureddine Kerzazi
Dept. Research & Development, Payza.com
Montreal, Canada
noureddine@payza.com

Foutse Khomh
SWAT, École Polytechnique de Montréal
Montreal, Canada
foutse.khomh@polymtl.ca

Bram Adams
MCIS, École Polytechnique de Montréal
Montreal, Canada
bram.adams@polymtl.ca

***Abstract*—To detect integration errors as quickly as possible, organizations use automated build systems. Such systems ensure that (1) the developers are able to integrate their parts into an executable whole; (2) the testers are able to test the built system; (3) and the release engineers are able to leverage the generated build to produce the upcoming release. The flipside of automated builds is that any incorrect change can break the build, and hence testing and releasing, and (even worse) block other developers from continuing their work, delaying the project even further. To measure the impact of such build breakage, this empirical study analyzes 3,214 builds produced in a large software company over a period of 6 months. We found a high ratio of build breakage (17.9%), and also quantified the cost of such build breakage ranging from 904.64 to 2034.92 man-hours. Interviews with 28 software engineers from the company helped to understand the circumstances under which builds are broken and the effects of build breakages on the collaboration and coordination of teams. We quantitatively investigated the main factors impacting build breakage and found that build failures correlate with the number of simultaneous contributors on branches, the type of work items performed on a branch, and the roles played by the stakeholders of the builds (for example developers vs. integrators).**

Keywords—Empirical Software Engineering; Automated Builds; Data Mining; Software Quality.

I. INTRODUCTION

Nowadays, many software organizations are adopting Continuous Integration (*CI*) practices aiming to integrate the source code faster [1]. A typical CI system continuously monitors the version control system for new commits, after which the resulting configuration of the system is checked out, compiled, tested and analyzed for common code quality measures. As such, CI practices depend on build automation [2] for compiling, bundling, linking required dependencies, packaging into an executable form, and running automated tests. Automated builds provide early feedback on the integration process [1, 2] and are the cornerstone used by *(1)* testers to cross all quality gates and *(2)* the release team to fully automate the software release process, with the aim of speeding up the release cycle in a safe way.

The essential goal of CI is to reduce the number of broken builds ("build breakage"). For example, over the past 2 years, we have observed a high rate of broken builds in a typical large, commercial web-based system. This is a problem, since broken builds mean no (testable) product and hence delay the project while the problem is being analyzed and resolved [3]. Tests are frozen and lucky are the developers who did not pull the latest code, otherwise they would be blocked as well and unable to continue working on the project. In this organization, the release engineers would spend at least 1 hour per day to fix broken builds, which could take anywhere between minutes and multiple days. Project timelines slipped for 1 hour per day because of team members' disruptions, and the organization losing a lot of man-hours. The developer who broke the build looks bad and the team spirit drops. On top of that, distributed teams (who use CI more intensively) suffer even more from a build breakage than collocated teams because of the additional overhead of communication [4].

Despite the growing interest for continuous integration practices [5], software integration [4] and build systems [6], build breakage is a relatively unexplored area. As such, there are no general guidelines on how to handle breakage, let alone heuristics or indicators to handle them. Hassan et al. [7] built a prediction model to predict build breakage, yet this ignored factors related to multiple concurrent branches and merge errors. Furthermore, no qualitative analysis or interviews were performed, nor was the impact of a broken build quantified.

In order to understand build breakage and identify heuristics to predict them, we report on the analysis of 3,214 builds produced in the context of the parallel development of a large commercial web application over a period of 6 months. We have conducted both quantitative analysis and interviews. The quantitative analysis was used to investigate issues such as the frequency of build breakages and how much time one spends on fixing them. To understand under which circumstances build breakage happens and how it impacts the productivity of the overall team, we interviewed 28 software engineers (2 *Architects*, 2 *Team Leads* (*T-Lead*), 1 *Integrator*, 4 *Front-End Developers* (*F-Dev*), and 16 *Back-End Developers* (*B-Dev*), 3 Testers). We build on this qualitative study to characterize the main factors impacting build breakage. The contribution of the study is threefold:

- it quantitatively estimates build breakage costs;
- it reports on qualitative data analysis from interviews with practitioners about their automated build system;
- it quantitatively analyzes characteristics of broken builds according to technical and organizational dimensions.

The remainder of the paper is organized as follows: Section 2 provides related work and motivation. Section 3 describes the research methodology. Section 4 presents and discusses our findings related to *(1)* the cost of build breakage; *(2)* qualitative analysis of 28 interviews; *(3)* and the quantitative analysis of the

factors impacting build breakage. Section 5 outlines the threats to validity and Section 6 concludes the paper.

II. Related Work and Motivation

Continuous Integration (CI) is a software development practice aiming at frequent integration of team members' changes [1]. Coming from the context of agile development, the integration of small changes, several times a day, helps not only to detect issues as soon as they happen (i.e., immediate feedback), but also to minimize the duration and effort compared to a delayed big-bang integration. CI relies on an automated build system that is responsible for compiling and packaging the system using dedicated servers, triggering automated unit, regression and acceptance tests. Moreover, the resulting builds constitute a starting point for the release team to continue speeding up the release cycle with more automation, and hence less opportunities for human error.

Builds can be grouped into two categories based on whether they result from development or integration activities: *Continuous Builds* (*CB*) and *Integration Builds* (*IB*) [8]. *CB* are triggered when a developer commits her code changes to a given branch. For instance, build breakage might happen due to a developer mistakenly omitting some files when committing. *IB* happen when developers incorporate changes from other branches, which were developed in parallel by other developers or teams.

There has been little research on the cost of build breakage. Recent work discussed the organizational impact of the build process, but none has discussed the costs. McIntosh et al. [9] examined the version histories of nine open source projects in order to assess the effort required to maintain their build system. They found that the build maintenance increases the overhead of the development and test activities respectively by 27% and 44%. Phillips et al. [10] conducted an interview study to get more insights into how Release Managers (RMs) make integration decisions. The authors found 10 factors that help RMs make informed integration decisions, organized into 4 categories: *(1)* Branch Evolution, *(2)* Branch Health, *(3)* Project Awareness, and *(4)* Project Traffic Control. We have been inspired by those factors in our quantitative exploration. To the best of our knowledge, we are the first to investigate the cost of build breakages on a real industrial case.

A number of studies have discussed software integration failures. Hassan and Zhang [7] introduced a model to predict the outcome of the build process according to indicators organized along four dimensions: *(1)* Social; *(2)* Technical; *(3)* Coordination; and *(4)* Prior-results of the previous builds. Cataldo and Herbsleb [4] examined the factors leading to integration failures in the context of geographically distributed teams. They found that architectural dependency information was a major predictor of integration failures.

The closest research relating to ours is the build success prediction based on socio-technical aspects proposed by Kwan et al. [8]. The researchers examined the effect of socio-technical congruence on build success probability in an empirical case study conducted in a large industrial system. Their logistic regression models explore variables such as weighted congruence, number of files per build, number of authors contributing to the build, number of files in the build, number of work items, the build type and date of the build. Our work complements these studies by examining the following three research questions:

RQ1. What is the relative impact of build breakage on a software project?

RQ2. What are the typical circumstances under which a build breaks?

RQ3. What are the factors impacting build breakage?

The first question (*RQ1*) explores the impact of failed builds on software projects in terms of their frequency, costs, and consequences. To understand the circumstances and context under which the build is broken, we interviewed team members involved in build breakage (*RQ2*). Finally, *RQ3* builds an explanatory model based on the relevant indicators pointed out for *RQ2*. The paper iterates between qualitative and quantitative approaches to refine our analysis.

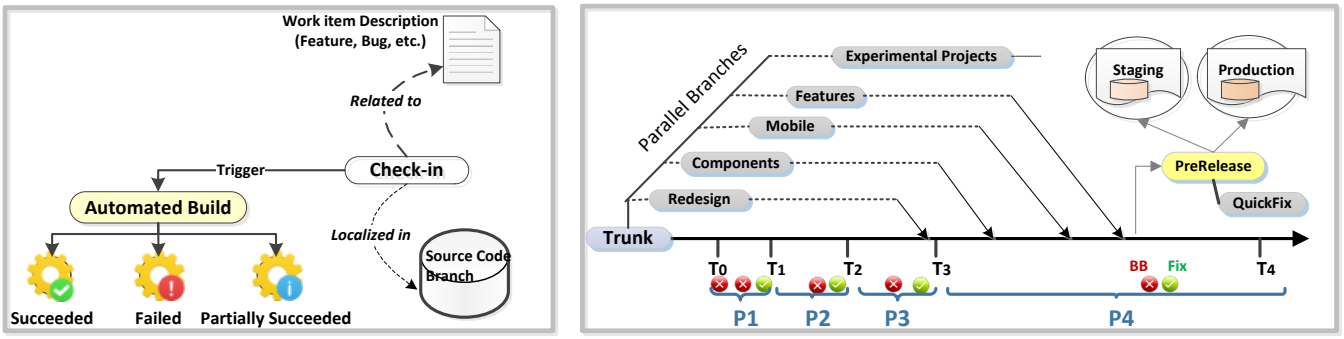
III. Methodology

A. Research Setting

The study takes place in a large software development organization with 200 employees dedicated to the development of a web based financial system. The software system that we studied is a mature product developed and maintained since 2004. The system is composed of over 1.5 million lines of .Net code organized in 8,524 source code files that feed six interdependent projects (2 Front End Applications, Back Office, Mobile, APIs, and Sandbox). The development team is distributed across two sites located in Canada and India.

The organization uses an agile development approach, where all development activities (new features and bug fixes) are carried out within separate branches. When development completes, QA engineers test the code on that branch, before giving the green light to integrate this code into the main branch (Trunk). Afterwards, to support Continuous Integration practices, the organization relies on one central build server acting as build controller and four build Agents dedicated to the processor-intensive work of the build process. Figure 1-a shows how automated builds are triggered by the version control system.

Before committing changes, developers must describe the work carried out by the check-in in a work item description such as a feature, bug, or change request. This piece of information is relevant to figure out the nature and the context of the work. Since a developer's changes are always localized in a Version Control System (VCS) branch, the scope of build breakage is limited within the branching structure provided by the VCS (i.e., only one branch is down). It is worth noting that all automated builds are associated with one VCS branch. Thus, for each source code branch, we have a history of builds. The result of an automated build will either be success or fail. A partial success means that the system did not pass the automated tests, yet in this paper we consider this case also as a failed build.



(a) Automated build process and related events

(b) The branching structure and abstract timeline

Fig. 1. Data collection, time periods, and branching structure.

Since previous studies indicated the importance of the size of changes [11] and the proximity to the release date as underlying causes of quality issues, we consider the following major time periods related to build breakage (Fig.1 (b)):

P1	[T0, T1]	Beginning of the project/iteration
P2	[T1, T2]	Progress of the work including forward merges (sync) from Trunk to that branch
P3	[T2, T3]	Pre-release period for polishing bugs and changes inside the branch (code Stabilization)
P4	[T3, T4]	After the backward merge from branch to Trunk, which is when build breakages can occur in Trunk. If not fixed, those breakages can impact other branches via later forward merges.

From experience, we found that in parallel development (see Fig.1-b) the timing of build breakage in a branch is a relevant predictor for the ease of future integration of this branch into the main streamline (Trunk) of the code, and as such transitively into future branches. For instance, a large amount of failed builds in the P_1 period might indicate a challenging code refactoring in the branch itself (Fig.1,P_1), while a large amount of build breakage at P_3 predicts a considerable workload for code stabilization in the Trunk. This assumption will be verified quantitatively in *RQ3*.

B. Research Method

Instead of analyzing multiple systems at a rather high level, we study one large web system in depth using a mixed method of qualitative and quantitative approaches [12]. We aimed to triangulate data sources covering multiple dimensions, to ensure that the results are valid and representative of an industrial environment. Figure 2 shows an overview of the methodological approach used for this study.

We start by examining the impact of build breakage on the software project by assessing the extent of the build breakage on the cost of software projects (*RQ1*). This is followed by an analysis of interviews with 28 team members to figure out the circumstances (*RQ2*) of build breakage and to gather indicators for a deeper quantitative analysis.

Then, we explore quantitatively a number of relevant characteristics of build breakage and use a statistical approach to analyze which characteristics explain best the likelihood of having a build breakage (*RQ3*). We explain the specific method for each research question within the result sections.

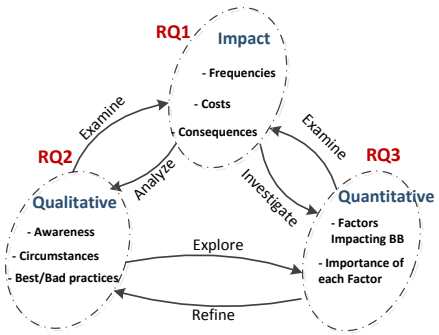


Fig. 2. Triangulation approach applied in this study.

IV. Results

A. *RQ1. What is the Relative Impact of Build Breakage on a Software Project?*

1) Motivation

Although one can argue about the impact of build breakage, intuitively, software development managers and researchers alike want a concrete estimation of the costs associated with it. In this research question we are interested in calculating the direct cost of a build breakage. Direct cost pertains to the cost of the team causing or relying directly on the now broken code. The longer the time during which development and tests are frozen (downtime), the higher are the direct costs. Although developers in theory can continue working on other branches (as long as they do not merge the broken code into their branch), at some point they as well will be blocked until the broken code is fixed and can be merged for further development. Furthermore, QA team members will wait for a consistent build to avoid any re-work. Note that we do not study the indirect cost of a broken branch on other branches.

2) Approach

Knowing, for a given branch, the time interval between each broken build and the next successful one, as well as the stakeholders involved in each branch, we are able to provide an estimated value of the cost related to broken builds. This metric gives us an estimate about the downtime of development and test activities related to that branch. Knowing the number of people working on the branch, we can then estimate the number of man-hours lost by the organization, which relates directly to a monetary cost. Assuming that the direct cost is a linear function, we use a fairly straightforward formula for its calculation:

$$Cost_{Branch} = P \times \sum_{i=0}^{n} (Downtime(Branch)_i) \quad (1)$$

$$Cost_{Global} = \sum_{b=1}^{m} (Cost_b) \quad (2)$$

Here, *n* is the number of build breakages within Branch i, *P* is the number of people (assuming constant for simplification) involved in the branch (*Devs*, *Testers* or *Integrators*), and *m* is the total number of branches that support parallel development of the software product. The global cost is the sum of the branches' costs expressed in number of man-hours. It is worth noting that, by virtue of parallel development, the downtime of a given branch affects only people working on that branch.

3) Results

At its peak, there are 19 broken builds per day (mean of 1.78 per day, with skew of 3.03). Figure 3 shows the distribution of unsuccessful builds over the period of our study. We can see how build breakage seems to have a report with the growing development activity. For instance, one can observe a high number of builds in November and early December 2013 with a relative augmentation of the build breakage rate. The results of those builds constitute the cornerstone for the release process.

Development and tests freeze on average 56.03 minutes (± 8.51) per branch (median of branches' median). Figure 4 displays the distribution of downtime for all 25 active branches[1]. The average downtime ranges between 35.7 and 85.3 minutes. Table I summarizes our findings for the 25 branches, which cover the five types of branches: Architecture, Projects, QuickFix, PreRelease, and Trunk. For example, the branch called Trunk, which represents the mainline development stream, has a median downtime equal to 46 minutes. It is worth noticing that the downtime of a build depends on the importance of the branch (see *RQ3*). For instance, as depicted in Figure 4, the downtime is very short in the branch dedicated to the releases (counted in minutes) compared to a branch dedicated to re-architecting (counted in hours or even days).

The global direct cost of the 25 branches ranges from 904.64 to 2034.92 Man-Hours over roughly 6 months. In order to estimate an average direct cost of downtime (in man-hours) by branch, we multiplied the median value of downtime by the number of members working on that branch. For example, 27 broken builds occurred in Trunk, which might slow down the work of 10 to 20 persons. Hence, the total direct cost for the Trunk branch ranges from 270 to 540 man-hours. It is worth noting that the Trunk branch is not used for development, but mainly for the integration of source code coming from other branches and stabilization. Based on the data presented in Table I, we computed the total amount of downtime across all branches.

TABLE I. SUMMARY OF COSTS FOR 25 BRANCHES BROKEN DOWN ACROSS 5 TYPES OF BRANCHES.

Branch	Median Downtime	∑ Broken Build	∑ Members	Cost M/H
Architecture	150	57	[1-3]	[142.5-487.35]
Quick Fix	20	3	[2-6]	[2-12]
PreRelease	16	23	[2-17]	[12.26-104.26]
Trunk	46	27	[10-20]	[207-414]
Branch-2	30	1	[3-4]	[1.5-2]
Branch-3	28	5	[2-3]	[4.66-7]
Branch-4	44	10	[2-3]	[14.66-22]
Branch-5	25	2	[2-3]	[1.66-2.5]
Branch-6	28	17	[2-4]	[16.33-31.73]
Branch-7	44	7	[2-4]	[10.26-20.53]
Branch-8	360	11	[2-4]	[132-264]
Branch-9	52	28	[6-10]	[145.6-242.66]
Branch-10	33	3	[2-4]	[3.3-6.6]
Branch-11	30	5	[2-5]	[5-12.5]
Branch-12	65	2	[1-1]	[2.16-2.16]
Branch-13	182	10	[2-3]	[60.66-91]
Branch-14	66	5	[2-4]	[11-22]
Branch-15	62	3	[2-3]	[6.2-9.3]
Branch-16	30	37	[4-10]	[74-185]
Branch-18	40	1	[2-6]	[1.33-4]
Branch-20	130	3	[4-7]	[26-45.5]
Branch-21	150	1	[1-2]	[2.5-5]
Branch-23	58	6	[1-2]	[5.8-11.6]
Branch-24	70	5	[2-4]	[11.66-23.33]
Branch-25	46	3	[2-3]	[4.6-6.9]
Total				**[904.64-2034.92]**

In overall, a minimum of **904.64** man-hours were lost. Development and Tests Freeze on average **56.03** minutes (± 8.51) per branch.

Developers and management should consider taking steps to reduce this amount of man-hours lost to build breakage. In the following research questions, we investigate the circumstances and the factors affecting build breakage in more details, in order to support managers in making informed decisions about how to reduce build breakages and their related costs.

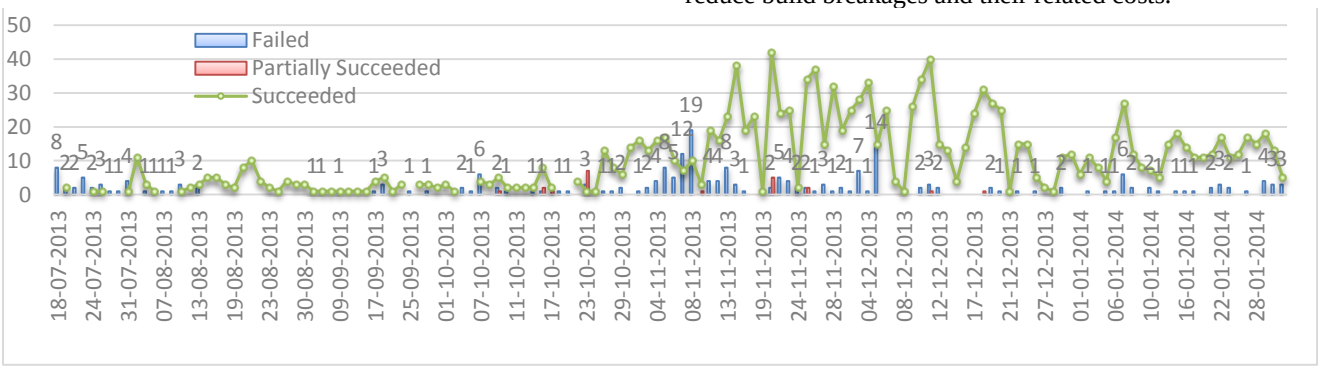


Fig. 3. Distribution of the build results by day.

[1] Obfuscated for business confidentiality.

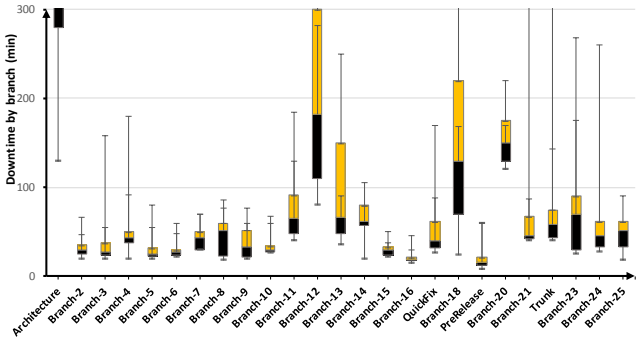


Fig. 4. Distribution of downtime by branch for the studied period.

B. *RQ2. What are the Typical Circumstances under which a Build Breaks?*

1) Motivation

Since build breakage has a major impact in terms of number of man-hours lost (see *RQ1*), it is important to understand why developers keep on breaking builds. Ideally, they should not commit their changes to version control (and hence the automated CI system) before making sure that all tests pass on their machine. Yet, build breakage keeps on happening and impacting their teammates and even other teams. To gain insight into the circumstances of build breakage and the emergent effects on the collaboration and coordination with other team members, we interviewed 28 team members (spread across different teams and roles) within the company.

2) Approach

We used semi-structured interviews for our investigation. First, we limited the group of interviewed employees to the contributors responsible for either the most broken or the most successful builds (above the 20th percentile). Inside this group, candidate interviewees were carefully selected according to three criteria: years of experience in the company, software development role played, and how frequently they trigger the builds. All persons interviewed had more than 3 years of work experience at the company (median= 4.5, average = 4.6, SD =1.34). We made sure that participants covered all roles in order to address different points of view: 2 *Architects*, 2 *Technical Leads*, 1 *Integrator*, 4 *FrontEnd-Devs*, and 16 *BackEnd-Devs*. After the first 3 interviews with the above participants (all related to source code development), we figured out that QA members should be interviewed as well because they too are concerned with the build results. Hence, we made an informed decision to add 3 testers to the list of persons to be interviewed.

The participants were solicited by personalized emails. The interviews themselves were done in person, and typically lasted 50 minutes. After introducing the goals of the interview, we notified the participant that the answers would be anonymized and that we were not assessing the quality of their work, but trying to learn more about the usage of the automatic build system, and therefore continuous integration practices. The interviews were private to avoid targeting persons and to encourage more sincere responses [13].

Table II presents our series of questions, organized by themes. The first part of the interviews focused on the awareness of the interviewees about build breakages. In the second part of the interviews, we were more interested in exploring (a) the main factors impacting build breakages according to the point of view of each interviewee's role in the software development process, and (b) the best practices that they followed to avoid build breakages. The latter two themes resulted in open-ended questions (Q5, Q6, Q7, and Q8) to allow participants to provide their insights. Emergent observations were classified using the card sort approach [14], and later served as input for a focused quantitative analysis in *RQ3*. Card sort is a technique widely used to classify textual concepts according to their intrinsic relationships. We use the technique in our case to discover, organize, and share common themes in the answers to the interview questions as well as the relationships between different themes.

TABLE II. QUESTIONS ASKED TO PARTICIPANTS.

Theme	Question	The purpose of this question
Awareness	***Q1.*** Do you pay attention to the automated build notifications or are you rather informed by a co-developer?	To explore build breakage awareness and its implicit effects on team collaboration.
	Q2. How long does it take you to be aware that your build failed?	To find out the importance of CI builds for a stakeholder.
	Q3. Does fixing broken builds consume a lot of time? How much time does a fix take on average?	To assess the impact on the project schedule and how to eliminate wasted time.
	Q4. What collaboration issues have you encountered when breaking the build?	To figure out the negative consequences on collaboration and coordination.
Impacting Factors	***Q5.*** Based on your experience, what were the most important factors responsible for build breakage?	To investigate common root causes of build breakage and how to manage and fix broken builds.
	Q6. How many workspaces do you have in your development environment and how easy is it to switch from one to another?	A workspace corresponds to a developer's files and changes related to a particular branch (for example a checked out version of a Subversion repository), hence the more workspaces, the more branches in which the developer is active at a given time. This could be a reason for developers to not care about one branch being blocked (since the developer could continue working on another branch in the meantime), or to be under more pressure and hence more likely to cause a build breakage.
Best Practices	***Q7.*** What are the steps followed before committing changes related to merges or regular development activities?	To analyze actions carried out before committing changes.
	Q8. Are you aware of any best practices that could help to decrease the amount or risk of broken builds?	To highlight best practices before committing pending changes.

3) Results

This section presents the results of our card sort analysis, then describes the situations under which software engineers broke the build.

Awareness

We were interested in the degree to which source code contributors were aware of their own build breakage as well as those at the team level. Awareness about changes occurring in parallel is a precondition for successful and timely integration and propagation of code changes [3], especially regarding components (and hence branches) of the system on which we have interdependencies. Hence, when developers are not aware of previously committed refactoring changes, they might break the build. For instance, a developer might modify code that calls a function, while its signature has been modified by a teammate within another branch.

Q1. **To our surprise, only 28% of the contributors are aware of build breakage.** Only 7 out of 28 individuals stated that they track the build results after committing their changes. The other interviewees explained that they only learn about broken builds later on (i.e., after the automatic CI build and tests) from the testers, since those require a successful build before being able to test the new version of the system. In particular, we observed that the Integrator and Testers are more aware of broken builds than other roles, likely because this role is responsible for the aggregation of separate parts of source code coming from different branches.

For example, one developer said: "*I switched off the Build notifications to avoid any distraction. I check the results of my builds only in predefined time windows within the day.*", which was elaborated upon by another developer: "*Usually, I am noticed for a broken build by QA because they are the persons who upload builds to the test environments. When other team members broke the build related to the branch that I am working on, I warn them to watch out.*"

On the other hand, the perspective and needs of an Integrator & Release Manager were completely different: "*Mainly, I am interested by integration builds triggered after code merges or integrations. However, I'm always aware about the build notification, especially the red ones. When a build notification for a branch shows up, I examine not only that build, but also the history of build failure on that branch. This is a good predictor of the integration effort that I will spend to merge back this branch into Trunk. On my side, major issues are coming from the integration of different parts of the code, especially when refactored.*"

Q2. **On average, it requires 3 hours (171 minutes) to become aware of a broken build.** On the one hand, this seemingly lax attitude towards build breakage can be explained as a consequence of code isolation in branches, since branches by definition are meant to shield other teams from experimentation or invasive changes to the code. On the other hand, as pointed out by other researchers [3], this protection can become an issue when the branch's changes are merged back into the other branches, since this is the moment when all changes (especially invasive ones) are revealed at once.

Nevertheless, two of the interviewed developers claimed to use an IDE plugin that warns them about build breakages when checking their pending changes, for example by signaling that the latest build is broken. It also provides an indication of who is responsible for the breakage. Another tester used a more implicit approach: "*As part of our manual tests related to a given branch, we have to upload the result of automated builds to the test server. With our upload tool, we can see only successful builds. When we see in the work items system tracker that the developer is done and we do not see the build result, then we have to communicate with the concerned developer to verify where the latest code to test is.*"

These findings suggest that Brun et al.'s [15] Crystal approach would be useful to identify breakages long before the branch's changes are merged back, i.e., during regular development.

Q3. **On average, a build breakage takes 57 min to fix**. This confirms our findings in *RQ1*. The time (effort) required to fix a broken build depends on the role, the characteristics of the branch and the nature of the build (triggered after merges or source code submission). Indeed, we have observed that it takes less time for the Integrator role to fix a broken build than for the developer. We can explain this observation by the fact that the different roles do not have the same perspective of the source code. For instance, while developers focus on a feature or bug, the integrators focus on an overall consistent system. Second, the integrator resolves the source code conflicts in an impartial way, i.e., with the project's global quality in mind rather than personal preferences.

Q4. **Roughly all participants (96%) agreed that working with a small, or at least tightly knit, team within a branch reduces build breakage.** They admitted that with an automatic build, collaboration becomes more transparent. For instance, it is easy to know who introduced the defects. However, since Q1 showed that most of the people ignore the automatic notifications, it seems like there are still some hurdles that keep people from realizing the full potential of automatic awareness support. One tester mentioned that "*when someone breaks the build and other team members continue to check-in their code, the build fix becomes a challenging task. Build server should warn people before committing changes.*"

Circumstances

Q5. **We identified three main factors impacting build breakage.**

All participants mentioned that missing files are the most probable underlying cause of build breakage. When checking in pending changes, developers might miss some files or libraries that are referenced in their local workspace. Although the system builds correctly in their local workspace, it does not on the build server. This incorrect behavior is due to the common human error of missing to inform the version control system of new files or dependencies.

The Integrator and Front-End Developer roles shed light on two more complex explanations for this factor. First, Integrators have to precompile the source code to improve the system performance before going to the staging environment (i.e., the environment right before production). In contrast to a normal build that compiles only packages referenced in the application,

a precompilation process builds all the contents of a folder, for example the *.aspx* files even not referenced in any project. As such, the precompilation can expose more errors than a normal build carried out by developers. Second, Front-End Developers experience many issues related to generated files. For instance, *.css* files are generated from .less files (through a mixture of translation and aggregation). This requires an adjustment of the automatic build process to generate the *.css* files before building the actual solution. One solution adopted by developers is the use of package management tools like *NuGet*[2] to incorporate such dependencies.

Mistakenly checking in work-in-progress is a second important factor. All Developers answered that they are using several workspaces (one for each branch that they use). As such, many of them acknowledged that, for example by forgetting to switch to the correct workspace (branch), they accidentally checked in files or file revisions that should not have been checked in, or files related to a different branch. Since this can be a rather subtle error (same file name, wrong revision), it can take a while to detect and fix those errors, especially if the person checking in the wrong file is not aware of the build breakage (Q1 and Q2).

Transitive dependencies between libraries are not specified in the build files and hence cause unforeseen inconsistencies. The interviewees have observed a difference in behavior when building using the automatic CI system compared to building manually within the IDE: "*It's always building on my machine before I checked-in. However, the server build does not behave the same way as Local build. The major problems that I have relates to library dependencies.*" For instance, assume three libraries *A*, *B* and *C*, with *A* referencing *B*, which in turn references *C*. Since *A* just needs *B* (directly), the developer will specify this dependency in her IDE, which will cause the build of *A* to succeed. However, since the transitive dependency between *B* and *C* did not need to be specified in the IDE, the automatic build system is not aware of this and might end up with a different version of *C* during the build than the one used by the developer on her local machine. This in turn will fail the build.

Since many IDEs do not offer support for detecting and/or specifying transitive dependencies (between *B* and *C*), this forces Architects to add the missing explicit reference from *A* to *C* in the source code to fix this build issue. Although this technically resolves the build issue, this of course has other negative consequences for the maintainability and comprehension of the source code, which only aggravates the likelihood of future build breakage.

It is worth noting that the organization currently uses the MSBuild build tool. A tool such as Maven (for Java systems) would be able to manage such transitive dependencies more efficiently. For instance, the front-end development team uses NuGet to support bundling the referenced files.

***Q6.* All participants answered that they are working on multiple (~ 4 to 15) branches in parallel**. In general, working on different branches by itself is not the main issue, due to the virtues and protection offered by parallel development. In contrast, the main issue are the differences in organization of workspaces. Some developers map all the branches on which they are working to a single workspace, while others map each branch to a separate workspace. This can cause confusion, since it is very easy to forget in which branch the developer currently was working, especially under pressure to fix an urgent security issue or build breakage: "*I am a nomad moving from a branch to another. I map one workspace by branch to avoid check-in work in progress related to other branch.*"

Curiously, we have observed during the interviews that developers that have different folders on their machine for different branches are more likely to break the build, compared to developers that have only one shared physical structure for all branches. This is because multiple folders leads to a higher chance of having inconsistent copies of the same file across different locations.

Furthermore, apart from having multiple inconsistent copies of files across folders, the need to switch context (for example, the work item being worked on or the state of the system) when moving from one branch to the other is time and effort consuming. For instance, a team member engaged in feature development might need to suspend her current work to fix a bug happening in production or blocking other colleagues. The interviewees unanimously agreed that such ad-hoc switches are error-prone: "*when I joined this team, I was assigned bugs on a specific project within the overall solution. I unload all other projects from the solution to focus only on one project. The code builds fine on my machine, but not on the server because of missing dependencies.*"

Best Practices

***Q7.* Nearly all (92%) contributors replied that they do not perform any additional formal instructive steps before checking in their pending changes.** However, some developers are more diligent when checking modified files: "*Before I check-in, I always compare diligently my pending changes with the latest version from the server. In this way I am not wasting team members' time with broken builds.*"

***Q8.* 12% of participants claimed to have their own routine for submitting their changes, driven by their previous experience and common sense.** In the majority of cases, this personal routine (not formal instructions) aims to check the files going into a check-in in order to ensure that the right files are submitted to the right branches.

Another suggestion from the interviewees was that new developers joining a development project should be required to learn the basic practices of code committing, as part of the process known as onboarding [16]. In this case, the newcomers should be supported at the process level when trying to submit their edits.

> Only 28% of the team members are aware of build breakage, taking on average 3 hours to be detected. Breakages typically take an hour to be fixed and mostly

[2] http://docs.nuget.org/. *NuGet* is an IDE extension that makes it easy to add and update libraries and tools within projects.

are caused by missing files, accidental commits or missing transitive dependencies. These root causes of build breakage might be related to the frequent context switches of developers (between branches).

C. *RQ3. What are the factors impacting build breakage?*

1) Motivation

In *RQ1*, we observed that build breakages have a large impact on the cost of software projects. From interviews, in *RQ2*, we have learned that only 28% of contributors are aware of build breakages, with the circumstances surrounding breakage depending on the point of view of each role. In this research question, we investigate the factors impacting build breakages using statistical models. Such quantitative analysis of potential impact factors will help stakeholders to identify specific factors that affect the occurrence of build breakages, giving them the possibility to reduce build breakages by monitoring and controlling these factors.

2) Approach

Using data collected over a period of 6 months from the large commercial web application, and based on the answers to the interview questions of RQ2, we hypothesize that:

- H1. Certain roles are more likely to break the builds than others. Our independent variable is 'Role': the role that performed the build (i.e., Integrator, T-Lead, F-Dev, B-Dev, or Architect);
- H2. A larger number of changed lines (churn) or files is linked with lower build success probability;
- H3. Integration Builds break more often than Continuous Builds;
- H4. The number of individuals involved in the development of a piece of software (i.e., contributors within a branch) correlates with build results;
- H5. The changes related to feature-related work items are more likely to break the build than those related to bug fixes, mainly because of the amount of code modified. Our independent variable is the type of work item associated with the build (i.e., Feature, Bug, or Integration).
- H6. Broken builds are more likely to happen in certain periods of the development process. Our independent variable is the date on which the build occurred, discretized by the release periods (P1 to P4), see Figure 1-b.
- H7. Build breakage is more common on certain working days and working hours. We broke down builds by the day of the week and hour when they were triggered;
- H8. The geographical distance of team members might be related to build breakages (i.e., local vs distant team).

For each categorical independent variable (H1, H3, H5, H6 and H8), we use a Pearson's chi-square test with Yates' continuity correction [17] with as dependent variable "breakage/no breakage" to check whether or not there is a statistically significant difference between the proportions of build failures in the different categories. For continuous independent variables (H2, H4 and H7), we apply Mann-Whitney test [17] to assess the difference between the mean values of the variable for broken and successful builds. We apply each test following the commonly used confidence level of 95% (p-value < 0.05) [17]. We choose these non-parametric statistical methods because they make no assumption about the distributions of the assessed variables.

To compute the importance of each independent variable, we then build a Random Forest (*RF*) [18] model using all the independent variables. RF is a non-parametric recursive regression method, which combines multiple regression trees (in our case 1,000) built using bootstrap samples of the data set, as training set, to quantify the importance of each variable in explaining build breakage [18]. We use this technique because *RF* is one of the most accurate learning algorithms available [19], and *RFs* run efficiently on large sets of variables and data without linear relationships. We measure the importance of variables in the random forest models using the normalized comparison of the increased mean square error (*%IncMSE*) of model predictions with predictions generated using randomly permuted predictor values. If a variable is important in the model, then randomly assigning other values for that variable should have a negative influence on the accuracy of the prediction. In other words, large changes in *%IncMSE* indicate important variables.

3) Results

This section presents the results of our statistical analysis.

H1. Build breakage depends on the role. The interviews (see *RQ2*) revealed that team members with different role viewed the concerns of build breakage differently. For example, participants invoke different circumstances that cause the build breakage (*Q5*). This observation led us to examine quantitatively the rate of build breakage by role. Figure 5 shows the distribution of build breakage across different roles. The integrators have the highest median score (9 breakages per month compared to 8 for B-Dev, 3.5 for F-Dev and Architect, and 1.0 for T-Lead). The chi-square test revealed a statistically significant difference in build breakage rates across different role groups, with $\chi^2 = 87.53$, p-value <0.001. A medium effect size is observed, with Cramer's V = .20. For the overall period studied, χ^2 shows that 59.2% of the Architects' builds were broken, while 15.3%, 12.4%, 12.1%, and 9.8% of B-Devs, T-Leads, Integrators, and F-Devs builds were broken respectively.

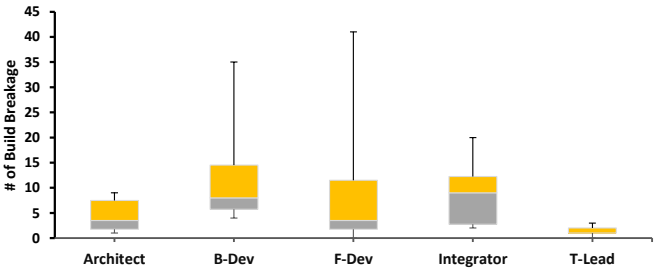


Fig. 5. Monthly distribution of build breakage according to roles (medians).

H2. Build breakage depends on churn. The Mann-Whitney Test revealed a significant difference in the build breakage regarding the size of changes measured by the number of changed files (U=207148, z= -7.11, p<.001 with a medium effect size (σ= -0.15). The average number of churned files for successful builds is 77.53, while it is 176.95 for failed builds. Similar test with code churn also reveal a significant difference (U=206329, z= -7.07, p<.001, σ= -0.15). Hence, the likelihood of build breakage increases as the number of files (or code

churn) involved in a build increases. We attribute this finding to the fact that large streams of code can make it difficult for developers to track all dependencies that should be updated. Figure 6 shows the amount of files impacted by source code merges along with the churn metric. On average, 958 (SD ± 1912) files are merged from the main streamline to the branches.

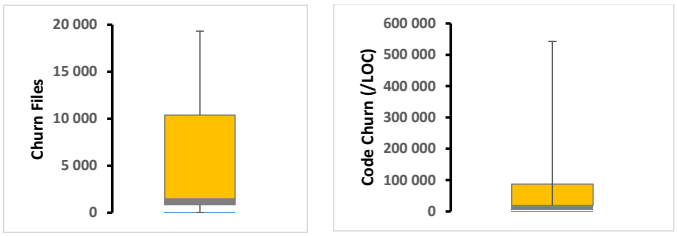


Fig. 6. Distribution of number changed files (left) and lines (right; churn).

H3. Integration Builds (*IB*) have more build breakage. The chi-square test revealed a significant difference in the build breakage probability regarding the Integration builds and Continuous builds; χ^2 (1, df = 2098) = 12.94, p-value < 0.001. 21.8% of IBs ended up failing, compared to 13.5% of CBs failing. A likely cause of this variance might be explained by two facts. First, when merging the source code from different branches, conflicts are the norm rather than the exception [14]. Second, integration builds typically contain more changes, and hence end up more likely with integration failures. The likelihood of build breakage increases for integration builds. The effect size Phi coefficient is -0.08, which is considered a very small effect using Cohen's [20] criteria (0.10 small, 0.30 medium, 0.50 large effect).

H4. Larger teams have more breakage. The Mann-Whitney Test revealed a significant difference in the build breakage regarding the number of coworkers in a branch (U=235834, z= -4.10, p<.001, σ= -.080). Successful builds are characterized by a mean of 7.6 contributors within a branch. The likelihood of build breakage increases by 10% when the number of branch's coworkers exceed 15. This finding is coherent with churn metrics, since more coworkers means larger churn metrics.

H5. Integration work items have more breakage. The chi-square test indicates a significant association between the type of work items and build results; χ^2 = 43.87, p-value <0.001. The effect size is medium, Cramer's V = 0.145. A proportion of 19.7% of integration work items breaks the build, followed by Features, and Bugs respectively at 17.4% and 8.3%. We can explain this result by the fact that the amount of files modified during integration work items is much larger than those modified when fixing bugs (i.e., Bug) or implementing a new feature (i.e., Feature).

H6. P1 sees more build breakage. The chi-square test reveals a statistically significant difference in the build breakage rate across different periods (χ^2= 9.77, p-value = 0.021). This observation corroborates results from previous work [21] that highlights the impact of time pressure on software quality. Nan et al. [22] propose to measure time pressure as the relationship between the planned delivery date and client requested delivery date. Figure 7 illustrates the distribution of build breakage across the four periods of development. A high rate of breakage occurs at the beginning of development period (P1). The most likely cause of this is source code refactoring at the beginning of each project. Two of the developers advocated that: "*Refactoring at the beginning of the project is one of the best practices. It is not only useful for refurbishing the internal structure of the system, but it's easy to justify such an effort to managers at the project's beginning*".

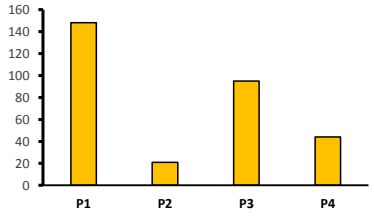


Fig. 7. Amount of build breakage according to the time periods.

H7. No link between working hour and build breakage. The analysis of the time of the day when a build is performed reveals that there is no significant difference in failure rates for different working hours (χ^2 = 20.84, df = 22, p-value = 0.531 >0.05). Commits submitted after normal working hours (referred to as late-night commits) are not significantly different in terms of build breakage than those carried out during normal hours. However, we observed a large downtime for branches affected by broken builds related to late-night commits.

H8. Local teams see more build breakage. We investigate whether teams delocalization might be a cause for build breakage. The chi-square test reveals a significant association between the geographical distance of team members and the build results; χ^2 = 11.06, p-value = 0.001 with a very small effect size (Phi coefficient = -0.05). The analysis shows that local teams break builds more often (15.1%) than delocalized teams (4.9%).

Which of the aforementioned independent variables are most related to build breakage?

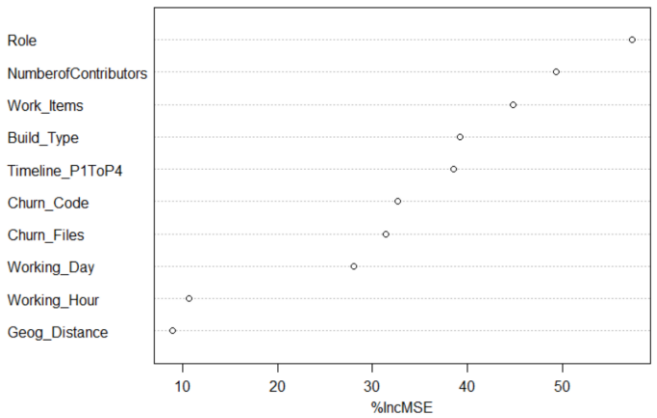


Fig. 8. Random Forest importance of independent variables.

Figure 8 shows that the role variable has the highest importance score (%*IncMSE*=58.74) among all independent variables, suggesting that the association between roles and build breakage is the strongest one, followed respectively by the number of contributors in a branch, the type of work item and build, and the development period (%*IncMSE* values of 53.71, 45.82, 39.56, and 38.59). Work time and geographical distance are of less importance (12.63 and 10.15, respectively).

Stakeholder role, the number of contributors in a branch, the type of work item and build, and the development timeline are the most important factors related to build breakage.

V. Threats to Validity

We cannot assume that the results of any empirical study generalize beyond the specific environment in which they were conducted. Since we studied one industrial system in depth, generalizing conclusions from our study is difficult because of a large number of contextual variables. However, with this longitudinal study across more than 6 months, we believe that our results contribute to increasing the existing body of knowledge [23], providing (for the first time) concrete cost estimations for build breakages, and raising a number of new questions regarding efficient parallel development. Note that we limited ourselves to the direct cost of build breakage on everyone involved directly with the offending branch. The real cost is higher, since teams working on dependent branches eventually will be impacted and hence lose time. However, such dependencies are hard to model accurately, hence we leave this as future work.

At the time of our study, there was only one person in the company who acted as software Integrator and release Manager. We observed that this role dealt mainly with the integration builds triggered after merging and integrating the source code from different branches (22.6 % of his builds break). The interview reveals that this role requires a thorough insight on all ongoing builds within branches to detect potential errors earlier. That is, the vision of parallel development seems to be in contrast with continuous integration principles. Future works should investigate this aspect.

VI. Conclusion

The work reported in this paper has three important contributions to the literature on Continuous Integration. First, our results provide one of the very first estimation of the build breakage cost. Indeed, 17.9 % of broken CI builds is a surprisingly high percentage that has to be investigated in more detail. Second, our qualitative investigation showed that the typical circumstances under which a build breaks are missing referenced files, mistakenly checking in work-in-progress, and transitive dependencies. Third, our quantitative analysis explored factors impacting the rate of build failures and the results revealed that the type of role, the number of simultaneous contributors in the branch, the nature of the work (Feature, Bug fix, etc.), the build type (*IB* vs. *CB*), and the period of the project are the most important factors related to build breakage.

Finally, the ultimate managerial purpose of this type of analysis is to reduce the amount of broken builds, and thus, decrease the wasted time. Our results aim at reduction on three levels: the costs, the circumstances, and the factors of build breakages. Based on our findings, the number of contributors in a branch and the period during which a change is made are actionable factors that can be manipulated to reduce build breakage. The findings reported in this paper can help steer further work on upstream development practices, CI practices for researchers, practitioners, and tool makers.

References

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

[2] B. Adams,H. Tromp,K. De Schutter, *et al.*, Design recovery and maintenance of build systems. in *Int Conf on Soft Maintenance, ICSM'07*, 114-123, 2007.

[3] Y. Brun,R. Holmes,M.D. Ernst, *et al.* Early Detection of Collaboration Conflicts and Risks. *IEEE Trans on Soft Eng, 39* (10): 1358-1375, 2013.

[4] M. Cataldo and J.D. Herbsleb. Factors leading to integration failures in global feature-oriented development: an empirical analysis. ACM ed. *33rd Int Conf on Soft Eng*, Honolulu, HI, USA, 161-170, 2011.

[5] J. Holck and N. Jørgensen Continuous Integration and Quality Assurance: A Case Study of Two Open Source Projects. *Australasian Journal of Information Systems, 11* (1): 40-53, 2004.

[6] S. Phillips,T. Zimmermann and C. Bird. Understanding and Improving Software Build Teams *The 36th International Conference on Software Engineering, ICSE'14*, India, 2014.

[7] A.E. Hassan and K. Zhang, Using Decision Trees to Predict the Certification Result of a Build. in *Automated Software Engineering, ASE '06.*, 189-198, 2006.

[8] I. Kwan,A. Schroter and D. Damian Does Socio-Technical Congruence Have an Effect on Software Build Success? A Study of Coordination in a Software Project. *IEEE Trans on Soft Eng, 37* (3): 307-324, 2011.

[9] S. McIntosh,B. Adams,T.H.D. Nguyen, *et al.*, An empirical study of build maintenance effort. in *The 33th International Conference on Software Engineering, ICSE'11*, 141-150, 2011.

[10] S. Phillips,G. Ruhe and J. Sillito, Information needs for integration decisions in the release process of large-scale parallel development. in *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, (Washington, USA), 1371-1380, 2012.

[11] A. Mockus and D.M. Weiss. Understanding and predicting effort in software projects *Proc of the 25th Int Conf on Soft Eng*, Portland, USA, 274-284, 2003.

[12] J.W. Creswell *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches, 3rd ed.* Sage Publications, Inc., 2009.

[13] B. Kitchenham and S. Pfleeger. Personal Opinion Surveys. in Shull, F., Singer, J. and Sjøberg, D.K. eds. *Guide to Advanced Empirical Software Engineering*, Springer, 2008, 63-92.

[14] Y. Brun,R. Holmes,M.D. Ernst, *et al.* Proactive detection of collaboration conflicts *Proc. of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, Szeged, Hungary, 168-178, 2011.

[15] Y. Brun,R. Holmes,M.D. Ernst, *et al.* Proactive detection of collaboration conflicts *13th European conf on Foundations of software engineering*, Szeged, Hungary, 168-178, 2011.

[16] T. Fritz,J. Ou,G.C. Murphy, *et al.* A degree-of-knowledge model to capture source code familiarity *32nd Int Conf on Software Engineering*, Cape Town, South Africa, 385-394, 2010.

[17] A. Liaw and M. Wiener Classification and Regression by randomForest. *R News, 2* (3): 18-22, 2002.

[18] J. Romano,D.J. Kromrey,J. Coraggio, *et al.*, Appropriate statistics for ordinal level data: Should we really be using t-test and cohen's d for evaluating group differences on the nsse and other surveys? in *Annual Meeting of the Florida Association of Institutional Research*, 1-33, February 2006.

[19] G. Biau Analysis of a random forests model. *Journal of Machine Learning Research, 13* (1): 1532-4435, 2012.

[20] D.J. Sheskin *Handbook of Parametric and Nonparametric Statistical Procedures (4 ed)*. Chapman & Hall/CRC, 2007.

[21] J. Eyolfson,L. Tan and P. Lam. Do time of day and developer experience affect commit bugginess? *8th Conf on Mining Software Repositories*, ACM, Waikiki, HI, USA, 153-162, 2011.

[22] N. Ning and D.E. Harter Impact of Budget and Schedule Pressure on Software Development Cycle Time and Effort. *IEEE Trans on Software Engineering, 35* (5): 624-637, 2009.

[23] V.R. Basili,F. Shull and F. Lanubile Building Knowledge Trough Families of experiments *IEEE Trans on Soft. Eng., 25* (4): 456-473, 1999.