Revisiting "Programmers' Build Errors" in the Visual Studio Context

A Replication Study using IDE Interaction Traces

Noam Rabbani
McGill University, Canada
nawras.rabbani@mail.mcgill.ca

Michael S. Harvey
McGill University, Canada
michael.harvey@mail.mcgill.ca

Sadnan Saquif
McGill University, Canada
sadnan.saquif@mail.mcgill.ca

Keheliya Gallaba
McGill University, Canada
keheliya.gallaba@mail.mcgill.ca

Shane McIntosh
McGill University, Canada
shane.mcintosh@mcgill.ca

ABSTRACT

Build systems translate sources into deliverables. Developers execute builds on a regular basis in order to integrate their personal code changes into testable deliverables. Prior studies have evaluated the rate at which builds in large organizations fail. A recent study at Google has analyzed (among other things) the rate at which builds in developer workspaces fail. In this paper, we replicate the Google study in the Visual Studio context of the MSR challenge. We extract and analyze 13,300 build events, observing that builds are failing 67%–76% less frequently and are fixed 46%–78% faster in our study context. Our results suggest that build failure rates are highly sensitive to contextual factors. Given the large number of factors by which our study contexts differ (e.g., system size, team size, IDE tooling, programming languages), it is not possible to trace the root cause for the large differences in our results. Additional data is needed to arrive at more complete conclusions.

ACM Reference Format:
Noam Rabbani, Michael S. Harvey, Sadnan Saquif, Keheliya Gallaba, and Shane McIntosh. 2018. Revisiting "Programmers' Build Errors" in the Visual Studio Context: A Replication Study using IDE Interaction Traces. In *MSR '18: 15th International Conference on Mining Software Repositories , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3196398.3196469

1 INTRODUCTION

The build system is a software development tool, which automates the process by which sources are compiled, linked, tested, packaged, and otherwise transformed into deliverables. The build system can incorporate several interacting technologies, e.g., *makefiles*, data loading scripts, dependency trackers, compilers, and debuggers. Modern *Integrated Development Environments* (IDEs) interface with build systems so that it can be executed to check the correctness of code changes as they are being produced.

MSR '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196469

Executing the build process (i.e., building) is at the heart of the development process. Indeed, since the 1970's, building has been a crucial stage in the four-step development process, where a developer must *think* of a solution, *edit* the codebase to implement the solution, *build* the codebase to propagate their changes to system deliverables, and *test* the newly created deliverables to ensure that the change had the desired effect [2]. Without at least one successful build, it is unlikely that a software project would yield a useful application that can be distributed to end users.

While ideally builds should be successful, they may also fail. These build failures help developers to identify problems in software systems early in the development process. For example, compiler warnings and test failures help to identify mistakes and faults in the codebase before they impact end users. On the other hand, build failures may also be a distraction, requiring developers to address these problems immediately in order to continue their work.

Previous studies have quantified the frequency and impact of team-level build failures. For example, Vassallo et al. [8] report a build failure rate of 26% at the ING financial organization. Rausch et al. [5] observe build failure rates of 14%–69% in 14 open source Java systems. Kerzazi et al. [3] estimate that between 893–2,133 hours are wasted due to the fixing effort that was needed to repair build failures in a large industrial system.

Recently, Seo et al. [6] were able to analyze build failures in C++ and Java developer workspaces at Google. These workspaces included Eclipse and IntelliJ for Java and ViM and Emacs for C++. In this study, builds were processed by a centralized build system.

To expand the scope of Seo et al.'s observations, we replicate their study in the Visual Studio context of the Mining Software Repositories (MSR) challenge. The build events in the MSR challenge dataset are of a similar granularity to those of the Google context, since they were extracted from traces of developer interactions with the IDE [4]. Builds in our context are assumed to have occurred using a local compiler in contrast to Google's centralized system. More specifically, in this paper, we revisit the following Research Questions (RQs):

- **RQ1: How often do builds fail?** We observe an overall build failure rate of 13.2% and a median developer-specific failure rate of 9.2%, which is 67%–76% lower than the failure rates that were observed in the Google context.
- **RQ2: How long does it take to fix a failing build?** Failing builds have a median resolution time of 2.7 minutes in our

study context, which is 46%–78% lower than the median resolution time of the study conducted at Google.

These results suggest that build failure rates are highly sensitive to changes in context. There are several notable differences between the MSR challenge context and the Google development context (e.g., system and team size, programming languages, IDEs, tooling). Additional context data is needed to identify the degree to which each of these contextual factors (and other potential confounding factors) affect the build failure rate.

Paper Organization. The remainder of the paper is organized as follows. Section 2 describes the design of our study, while Section 3 presents the results. Finally, Section 4 draws conclusions and proposes promising avenues for future work.

2 STUDY DESIGN

We structure our study of build failures by addressing two research questions (RQs) (Section 2.1). To address our research questions, we first filter and pre-process the MSR challenge dataset (Section 2.2) and then analyze it (Section 2.3). Figure 1 provides an overview of our study design while the text in the following subsections describes each step. Our data and scripts are available online.[1]

2.1 Research Questions

We begin our study by aiming to replicate the three research questions of Seo et al. [6]. Unfortunately, the second research question ("Why do builds break?") could not be answered using the MSR challenge dataset as the reason for failing build events is not recorded in the interaction traces. The MSR challenge dataset provides the necessary data to address the following two research questions:

- **RQ1: How often do builds fail?** Data about the frequency of failing builds will help us to understand the potential impact of failing builds on a software organization. For example, if builds fail frequently, steps to mitigate failing builds may have a positive impact on a software organization.
- **RQ2: How long does it take to fix a failing build?** Data about how long failing builds take to repair will deepen our understanding of the impact of failing builds on a software organization. For example, even if failing builds occur frequently, if they are fixed quickly, their impact on a software team may be acceptable.

2.2 Data Filtering

The MSR challenge dataset [4] is rich and diverse. In this section, we describe the steps taken to filter and preprocess the MSR challenge dataset to extract the necessary and suitable data for our study. Below, we describe each step in our data filtering process.

DF0: Select All Build Events. An initial inspection of the event types that are recorded in the MSR challenge dataset reveals that events of type *Build* are the only events that we need to replicate the study of Seo et al. [6]. Therefore, in our first step, we extract all of the build events from the MSR challenge dataset. For each extracted event, we include the associated user and session IDs.

DF1: Select (Re)build Action Events. Build events in the dataset have an *action* attribute, whose value is *Clean*, *Deploy*, *Build*, or *RebuildAll*. Clean actions restore the codebase to the state that it was in before any build steps were performed, while Deploy actions make the deliverables that were produced by a previous build event available for customers to install or interact with [1]. These events are not events that were tracked as builds in the Seo et al. study, so we filter them out of our dataset before performing our analysis.

Table 1 shows that Clean and Deploy actions are rare in the MSR challenge dataset, only accounting for 272 of the 13,584 (2%) of the total build events. After filtering Clean and Deploy actions, 13,312 build events remain in our dataset.

DF2: Screen Inactive Users. Developers who build infrequently need to be removed from the dataset, since they are not representative of typical developers [6]. The approach of the study under replication was to focus on only those developers who build at least once per day. In our setting, this filter is too aggressive, and would remove 19% of users. Instead, we opt to filter out the users whose number of builds is below the fifth percentile, which in our dataset is 5 builds. This filter removes an additional 12 builds, yielding a final sample size of 13,300 builds.

2.3 Data Analysis

After extracting and filtering the MSR challenge dataset, the 13,300 build events that are suitable for our replication study need to be analyzed. Figure 1 provides an overview of our data analysis approach, which is comprised of five steps. Below, we describe each step in our approach.

DA0: Identify Failed Builds. The build events in the MSR challenge dataset were not logged with a success indicator. Instead, each build is comprised of targets (stages), each of which may succeed or fail. In our context, we define a successful build as one where all of its targets are successful. Conversely, if any target within a build fails, we flag that build as failed.

DA1.1: Compute Failure Ratio. To address RQ1, we need to compute the rate at which builds fail. Thus, we compute the total number of failures and normalize it by the total number of builds. We use this measure to analyze (1) the overall rate of build failures; and (2) user-specific and session-specific rates of build failure.

DA2.1: Identify Build Sequences. More than one project may be built during an IDE session. Therefore, even after grouping build events by user and session, we must apply an additional step to identify *build sequences*, i.e., builds of the same project within a session. For each build event within a session, we extract the names of the targets that were included in that build. We consider a series of build events to be in a sequence if the names of their targets match above a Target Similarity Threshold of t.

Higher t settings yield fewer build sequences with lower rates of false positives, but may be susceptible to false negatives. On the other hand, lower t settings yield more build sequences with lower rates of false negatives, but may be susceptible to false positives. Since this t setting may impact our results, we analyze the data using four t settings, i.e., 70%, 80%, 90%, and 100%.

DA2.2: Identify Fail-Fix Pairs. To address RQ2, we need to identify pairs of builds within sequences where the earlier build in the pair represents a build failure and the later build in the pair is the first successful build along the sequence after this failure (i.e., its fix). To identify these *fail-fix pairs*, we iterate over each build

[1] https://dx.doi.org/10.6084/m9.figshare.5995447

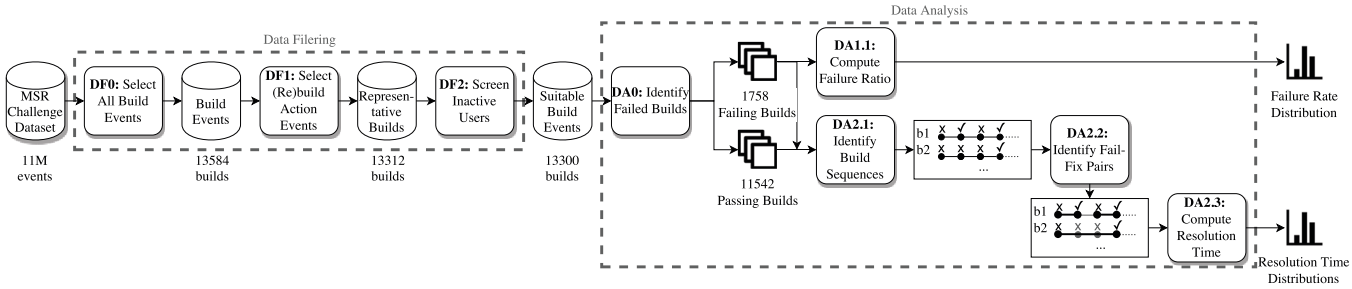

Figure 1: An overview of our approach to replicate the study of Seo et al. [6] using the MSR challenge dataset.

sequence to find a failing build that follows immediately after a successful build. This is identified as the failing build in the fail-fix pair. We then continue along the build sequence, skipping each additional failing build until we encounter a successful build. This build is identified as the fixing build in the fail-fix pair. This process is repeated until the end of the sequence is reached, and is repeated for each build sequence in the MSR challenge dataset.

DA2.3: Compute Resolution Time. For each fail-fix pair, we compute the resolution time using the definition provided by Seo et al. [6]. The resolution time is the elapsed time between the end time of the failing build and the start time of the successful build in the fail-fix pair. We do note that this definition of resolution time has limitations, e.g., one cannot know that the build has been fixed until the successful build has finished. However, in order to draw comparisons with the Seo et al. [6] study, we reuse this definition of resolution time.

Since each event in the MSR challenge dataset only contains a starting time, we must compute the end time of failing builds in fail-fix pairs. Fortunately, each build event has a duration attribute. Therefore, the resolution time $rt = t_{fix} - (t_{fail} + d_{fail})$, where t_{fix} and t_{fail} are the starting times of the fixing and failing builds, respectively, and d_{fail} is the duration of the failed build.

The resolution time is susceptible to noise, since developers may take breaks between the failing and fixing builds in a fail-fix pair. As was done by Seo et al. [6], we mitigate the impact of extreme cases by filtering out resolution times greater than 12 hours.

3 RESULTS

In this section, we present the results of our replication study with respect to the two research questions.

RQ1: How often do builds fail?

Results. Builds in our context fail less often than those of the Google context. In the original study, Seo et al. [6] observed median failure rates of 38.4% among C++ developers and 28.5% among Java developers at Google. The 9.9 percentage point discrepancy between the medians was in part due to Java developers' tendency to use IDEs (e.g., Eclipse, IntelliJ IDEA), while C++ developers tended to use text editors (e.g., ViM, Emacs). Text editors will offload most error detection to the build process. Many of these errors would typically be caught by an IDE.

Table 1 shows that, in our Visual Studio context, 1,758 of the 13,300 (13.2%) studied build events fail, which is 15.3 percentage points lower than the median build failure rate among Java developers at Google. Moreover, Figure 2 shows the distribution of user-specific build failure rates in our context. Our median user-specific failure rate is 9.2%; 19.3 percentage points lower than the median observed build failure rate among Java developers at Google.

The lower failure rates that we observe in the MSR challenge context may be due to several factors. First, Visual Studio may be detecting and suggesting fixes for a larger set of errors than Eclipse and IntelliJ IDEA. Those fixes may be preventing errors from becoming build failures. Second, lower failure rates may be due to a difference in build complexity in the contexts. The builds

Table 1: Build action types and their failure frequency.

Action	# Builds	# Failing Builds	% Failed Builds
Build	12,662	1,606	12.6%
RebuildAll	650	154	23.7%
Clean	263	2	0.8%
Deploy	9	4	44.4%
Total before filtering	13,584	1,766	13.0%
After filtering out Clean and Deploy actions	13,312	1,760	13.2%
After filtering out inactive builders	13,300	1,758	13.2%

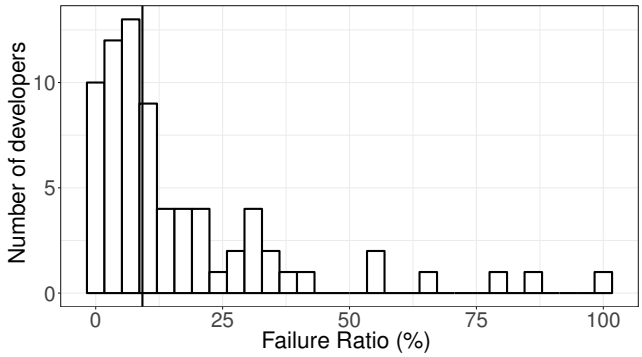

Figure 2: The distribution of user-specific build failure rates in our dataset with bin size 30. The vertical line shows the median value; 9.2%.

in the MSR dataset are likely to be less complex than the builds at Google, which are notoriously large [7]. Indeed, one of the largest builds in the replicated study was comprised of 2,858 build targets and used six programming languages. By comparison, the largest build in the MSR dataset has 350 targets. To better understand how much each factor contributes to the discrepancy in failure rate, project characteristic data is needed to track the context in which the builds in the MSR challenge dataset were performed. One user had a 100% failure rate. This user attempted 22 times to build the same single target over eight hours. These build durations ranged from 0.7 to 4.5 seconds. Although this user self-reported a positive two comfort level with C# programming and positive one comfort level with general programming (on scales from negative three to positive three), their number of builds fell within the 25th percentile and the number of days with builds logged fell in the seventh percentile. Therefore, we take this user and their builds as outliers.

***Observation 1**: Builds in the visual studio context of this study fail less frequently than those from the Google context.*

RQ2: How long does it take to fix a failing build?

Results. Builds in our context take less time to fix than those of the Google context. Developers at Google spend a median of five and twelve minutes fixing C++ and Java build failures, respectively [6]. Figure 3 shows the distribution of resolution time in the MSR challenge dataset for target similarity thresholds of 70%, 80%, 90%, and 100% (recall Section 2.3).

Figure 3 shows that, when choosing the most conservative target similarity threshold of 100%, build failures in the MSR challenge dataset take a median of 2.7 minutes to fix. The thresholds of 70%, 80%, and 90% have respective medians of 2.6, 2.7, and 2.7 minutes.

Regardless of the target similarity threshold that is selected, builds are generally fixed 46%–78% quicker in the MSR challenge dataset. Again, many factors may contribute to this discrepancy. First, the differences in work environments, languages, and tooling may impact resolution time. Second, the less complex builds of the MSR challenge dataset may also impact on resolution time. Similar to RQ1, additional data about the context of the MSR challenge builds is needed to better understand the impact that each factor has on resolution time.

***Observation 2**: Builds in our dataset are fixed 46%–78% quicker than those from the Google context.*

4 CONCLUSIONS

Software builds are a crucial part of the development process where source code and all relevant artifacts are combined to produce deliverables. Studies of failing builds yield insight about common mistakes and suggests modifications for their prevention, avoidance, and detection. While prior studies have been published describing build failures in other settings, to the best of our knowledge, this study is the first to conduct such an empirical study in the context of Visual Studio. By replicating two research questions of the Seo et al. [6] study from the Google context, we make the following observations:

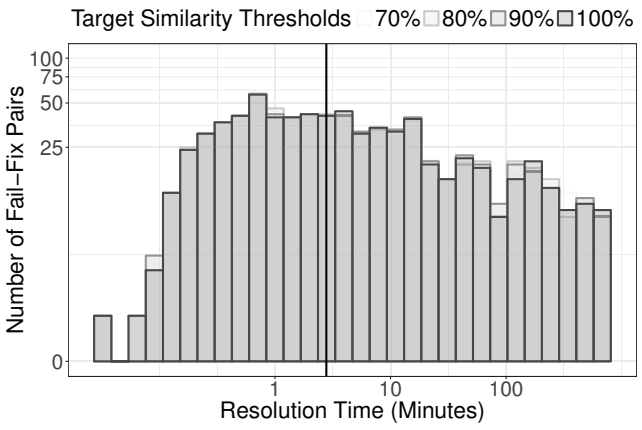

Figure 3: Resolution time of builds with different Target Similarity Thresholds. The vertical line shows the median value for 100% Target Similarity; 2.7 minutes.

- The median user-specific failure rate is 9.2%, which is much lower than the failure rates of 28.5–38.4% that were observed in the Google context.
- Failing builds take a median of 2.7 minutes to fix in our context, whereas failures in the Google context take a median of five to twelve minutes to fix.

These results lead us to conclude that build failure rates and resolution times are highly dependent on the study context. Since several contextual factors are in different between our study context and the Google study (e.g., system size, team size, programming languages, IDE tooling), it is not possible to pinpoint why we observe such a large difference in our results. Additional contextual data (which is not available in the MSR challenge dataset) is needed to perform a more detailed root cause analysis.

REFERENCES

[1] Bram Adams and Shane McIntosh. 2016. Modern Release Engineering in a Nutshell: Why Researchers should Care. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 78–90.
[2] Stuart I. Feldman. 1979. Make—a program for maintaining computer programs. *Software: Practice and Experience* 9, 4 (1979), 255–265.
[3] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. 2014. Why do Automated Builds Break? An Empirical Study. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. 41–50.
[4] Sebastian Proksch, Sven Amann, and Sarah Nadi. 2018. Enriched Event Streams: A General Dataset for Empirical Studies on In-IDE Activities of Software Developers. In *Proceedings of the 15th Working Conference on Mining Software Repositories*.
[5] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. 2017. An Empirical Analysis of Build Failures in the Continuous Integration Workflows of Java-Based Open-Source Software. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. 345–355.
[6] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' build errors: a case study (at google). In *Proceedings of the International Conference on Software Engineering (ICSE)*. 724–734.
[7] Mohsen Vakilian, Raluca Sauciuc, J. David Morgenthaler, and Vahab Mirrokni. 2015. Automated Decomposition of Build Targets. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 123–133.
[8] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. 2017. A Tale of CI Build Failures: an Open Source and a Financial Organization Perspective. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. 183–193.