Empirical Software Engineering (2022) 27: 65
https://doi.org/10.1007/s10664-022-10117-6

Revisiting the building of past snapshots — a replication and reproduction study

Michel Maes-Bermejo[1] · Micael Gallego[1] · Francisco Gortázar[1] · Gregorio Robles[2] · Jesus M. Gonzalez-Barahona[2]

Accepted: 6 January 2022 / Published online: 17 March 2022

Abstract

Context Building past source code snapshots of a software product is necessary both for research (analyzing the past state of a program) and industry (increasing trustability by reproducibility of past versions, finding bugs by bisecting, backporting bug fixes, among others). A study by Tufano et al. showed in 2016 that many past snapshots cannot be built.

Objective We replicate Tufano et al.'s study in 2020, to verify its results and to study what has changed during this time in terms of compilability of a project. Also, we extend it by studying a different set of projects, using additional techniques for building past snapshots, with the aim of extending the validity of its results.

Method (i) Replication of the original study, obtaining past snapshots from 79 repositories (with a total of 139,389 commits); and (ii) Reproduction of the original study on a different set of 80 large Java projects, extending the heuristics for building snapshots (300,873 commits).

Results We observed degradation of compilability over time, due to vanishing of dependencies and other external artifacts. We validated that the most influential error causing failures in builds are missing external artifacts, and the less influential is compiling errors. We observed some facts that could lead to the effect of the build tool on past compilability.

Conclusions We provide details on what aspects have a strong and a shallow influence on past compilability, giving ideas of how to improve it. We could extend previous research on the matter, but could not validate some of the previous results. We offer recommendations on how to make this kind of studies more replicable.

Keywords Compilability · Buildability · Build failures · Software reconstruction · Software builds · Software maintenance · Software evolution

Communicated by: Gabriele Bavota

✉ Gregorio Robles
gregorio.robles@urjc.es

Extended author information available on the last page of the article.

1 Introduction

The problems for building the current snapshot from source code has been discussed in detail in the research literature (see Section 2), but the problems for **building past snapshots** have received less attention.

Compilability of past snapshots of the source code of a software product has been shown to be of interest both for researchers and practitioners (Nikitin et al. 2017; Reproducible builds 2017). Some examples of its uses are as follows: **(1) to search and find bugs** developers often run previous snapshots of the system in order to locate bugs and understand how they originated (Zimmermann et al. 2006); **(2) due to security reasons** users usually trust available binaries of a library, but a backdoor could have been introduced (de Carné de Carnavalet and Mannan 2014), so rebuilding it from the original source allows to compare the binaries and verify that it was not modified; **(3) to backport bug fixes** it is necessary to build an old version to apply a patch to that specific version of the software (Tian 2017)); and **(4) to reproduce the past state of a system**, for research purposes, it is useful to obtain a functional executable to verify the correct performance of the system (Manacero 2011), or to use the project history to predict future bugs (Zimmermann et al. 2008)).

To our knowledge, the most complete study on the compilability of all past snapshots is presented in Tufano et al. (2017) (from now on "the original paper" or "the original study"). It analyzes all past snapshots for 100 Java projects of one organization (the Apache Software Foundation, ASF), determining how many of them could be built, and the main causes of failure in building. Its main conclusions were: only 38% of snapshots could be successfully built, almost all projects contained snapshots that could not be built (96%), and the main cause of failure when building a snapshot was dependency resolution. We decided to revisit and extend this paper, with two main aims:

(1) To validate the results of the original study, by trying to build **in 2020** the same snapshots it considered **in 2014**, answering the original two research questions (although slightly rephrased):

RQ$_{1a}$ "How many snapshots from the change history are compilable?"
RQ$_{1b}$ "Which types of errors prevent snapshots from being built?"

We reproduced the conditions and methodology of the original study as much as possible, studying compilability of the same snapshots with the Maven tool, as they did. In addition, we also wanted to learn if compilability had degraded. We suspected that it could be the case because one of the main reasons for failed builds in the original study was availability of dependencies, which is known to degrade over time (Bavota et al. 2015). So we added the following research question:

RQ$_{1c}$ "Has compilability degraded since the original study?"

While answering the previous RQs, we stumbled upon some problems that lead us to an additional one:

RQ$_{1d}$ "Are the data in the reproduction package of the original study enough for a replication?"

(2) To explore the generalizability of the results, by conducting another study with the same methodology but on a set of Java projects with a more diverse background:

RQ$_{2a}$ "How many snapshots from the change history are compilable?"

RQ_{2b} “Which types of errors prevent snapshots from being built?”
RQ_{2c} “Are there differences in compilability depending on the building tool?”

With the first two questions, we check the extensibility of the results of the original paper to other Java projects. The last question is aimed to find out if some building tools perform better in terms of compilability than others, for example because of the amount of information they require about the construction process and the construction context.

In the rest of this paper, we refer to the study that addresses $RQ_{1[a-d]}$ as *replication study*, and *reproduction study* to the one answering $RQ_{2[a-c]}$. This terminology is based on Juristo and Gómez (2010) and Cartwright (1991), which distinguish between **replication** (performing the same experiment again) and **reproduction** (performing the same experiment but with other input/data). Very recently, this terminology has been reviewed,[1] however in this paper we use the traditional definitions for replication and reproduction studies.

Our replication study analyzes 79 projects from the set of 100 in the original study, and our reproduction study will be performed on a dataset of 80 FOSS (free, open-source software) Java projects. In addition, for the reproduction we will extend the build systems with Ant (very popular in the old days of long-running projects) and Gradle (a newer build tool) — the original study only considered Maven. In both studies we used our own software for checking compilability and analyzing the resulting logs (see Section 9).

The rest of the paper is structured as follows: Section 2 discusses previous research. Section 3 defines the main concepts. Section 4 presents the methodology used in the studies. The results of applying the methodology are reported in Sections 5 (replication) and 6 (reproduction). Section 7 discusses the results, and explores threats to their validity. Finally, Section 8 draws conclusions and presents further research.

2 Previous Research

The build process and the errors preventing correct builds, have been an active area of research during the last years. One of the most influential empirical studies in this area was authored by Seo et al., who examined 26.6 million builds from Google’s centralized build systems, analyzing compilation errors in failed builds. As a result, an error taxonomy was provided based on log patterns (Seo et al. 2014). Sulír and Porubän examined the builds of more than 7,000 Java projects, but only for their last commit (Sulír and Porubän 2016). Other investigations have also focused on errors related to build failures. Rausch et al. address specifically the reasons why builds fail in the context of CI environments (Rausch et al. 2017). Travis logs from 14 open-source projects were analyzed, finding that a significant fraction of errors corresponded to tests that failed because of a failure in a previous build. The study analyzed the build logs from the point of view of continuous integration systems (snapshot build and test execution), but it did not include a reproduction of the builds. Some authors have emphasized the importance of historic compilability to propose repair tools for failed builds. Using a taxonomy for the root causes of build failures found in 86 out of the 200 most popular Java projects in GitHub, it was demonstrated that 52 of these failures could be resolved in an automated manner (Hassan et al. 2017). And the `HireBuild` tool was able to fix 11 out of 24 reproducible build failures using fix patterns automatically generated from existing build script fixes and recommending fix patterns

[1] https://www.acm.org/publications/policies/artifact-review-and-badging-current

based on build log similarity (Hassan and Wang 2018). None of the previous studies considered historic compilability, which is the subject of our study. They were in general based on the analysis of logs: in comparison, our studies perform our own building processes, starting from scratch with the source code available in the analyzed snapshots.

A related area to compilability is build reproducibility: *"the ability to generate byte-to-byte identical binaries from the source code of a project version, no matter who builds the binary, when or in which machine"* (Reproducible builds in Debian 2018). Reproducible builds create a verifiable path from human readable source code to the binary code used by computers, and are gaining relevance (Cito et al. 2017; Maudoux and Mens 2018; de Carné de Carnavalet and Mannan 2014; Perry et al. 2014). Software compilations, such as Debian and other Linux-based distributions, have a strong interest in the build reproducibility (Reproducible builds 2017; Reproducible builds in Debian 2018). Obtaining reproducible builds in Debian has been addressed in Glukhova (2017) and Ren et al. (2018), which present tools to ensure reproducibility, and a framework for detecting and fixing packages with problems. However, they focus on the latest version, not dealing with past reproducibility.

The reproducibility of builds is also interesting from a security point of view. Some works focus on bringing security into the software development life cycle, considering build reproducibility as one of the main issues to be taken into account. Proposals have been presented to use reproducible builds in the context of security-critical open-source software (de Carné de Carnavalet and Mannan 2014), decentralized software-update frameworks including build verifiers (Nikitin et al. 2017), systems to ensure binary transparency (Hassan et al. 2017), or enhancing trust in software through reproducible builds (Skrimstad 2018). Even when our work is relevant to obtain reproducible builds of past versions of the software, we have not dealt with the details needed to ensure it.

Compilability of past versions of a program has been used instrumentally in research or industrial activities. This is the case for bug location (Śliwerski et al. 2005; Asaduzzaman et al. 2012; Murgia et al. 2010; Zimmermann et al. 2006; Zimmermann et al. 2008). When locating bugs, techniques like `git bisect` may be used to traverse the project history of commits back to the past, to find the change that introduced a bug (Spinellis 2012; Meneely et al. 2013). In these cases, the utility of the technique is limited to tools performing static analysis, except when automatic compilability of past snapshots can be ensured — then, the debugged system can be also analyzed dynamically. Some authors have proposed metrics to evaluate the stability of project builds over time (Raemaekers et al. 2012). Others have addressed the problem in an indirect way, for example when trying to run mutant tests in previous versions of several software projects (Just et al. 2014) provided by Defects4J (Just et al. 2014). However, none of those studies systematically addresses the analysis of the compilability of past versions of real systems.

Finally, we already mentioned Tufano et al.'s work as the direct precedent of the studies we present in this manuscript (Tufano et al. 2017). Thus, its methodology and results will be discussed in detail later in this paper.

3 Definitions

We derive the terminology used in this paper from Sulír and Porubän (2016). According to it, the build process of projects programmed with compilable languages consists of following steps: **(1) read** the project build (configuration) file, **(2) download** third party components defined in the build file, **(3) execute** the compiler to generate binary files from source code, and **(4) package** the program in a suitable format for deployment.

A specific project version is **compilable**[2] if these steps can be executed to generate a valid binary, with a success build status. Based on this background, we define:

- **Snapshot**: a version of the source code of a project, represented by a commit of its git repository. It will be identified by the unique hash of the commit.
- **Snapshot with build configuration**: a snapshot with configuration files for a build system.
- **Successful build**: a snapshot that was compilable (*we could build it*)
- **Failed build**: a snapshot that could not be built.
- **Error for a build**: a failing build (and its cause).

4 Methodology

For both our replication and reproduction studies we use a similar workflow, sketched in Fig. 1. We work with git repositories, which means that we can clone the whole repository locally, and that each code snapshot corresponds to a commit in the git history of the repository. We locate repositories to analyze, clone them, and try to find out all the commits of interest. If we cannot clone a repository, or we do not find all the commits of interest in it, we discard it. Then, for each remaining repository, we get its commits of interest, and for each of these commits we investigate if it uses a building system. If so, we try to build it.

There are some differences between the two studies, mainly on how we find repositories, which ones are our commits of interest in them, how we find those commits, and which build systems we consider (see details below). Table 1 shows some quantification (projects and commits) for the original study, with all its projects (*Original Pristine*), for the reduced version of it, with the 79 repositories in which we could find all the commits (*Original Reduced*), and for our replication (considering Maven builds only) and reproduction studies. The distribution of commits per project for the replication and reproduction studies is given as well.

4.1 Replication Study

For the replication study we followed the methodology of the original study as much as possible. Its authors considered 100 git repositories corresponding to Java FOSS projects from the ASF, all of them using a Maven-based build infrastructure. They retrieved in September 2014 all commits in their master branches, claiming to have analyzed a total of 219,395 commits. Then, they attempted to build all of them locally using Maven. The original paper comes with an accompanying reproduction package listing in detail which commits they considered for each repository. When reviewing that list, we found that the total number of commits referenced is 174,505. This is the reason why this is the number we include in the tables for the original study (see details in Section 5).

[2]Although Sulír and Porubän use the term *buildable*, for consistency we have used the term *compilable* instead, as in the study by Tufano et al., which we would like to replicate and reproduce.

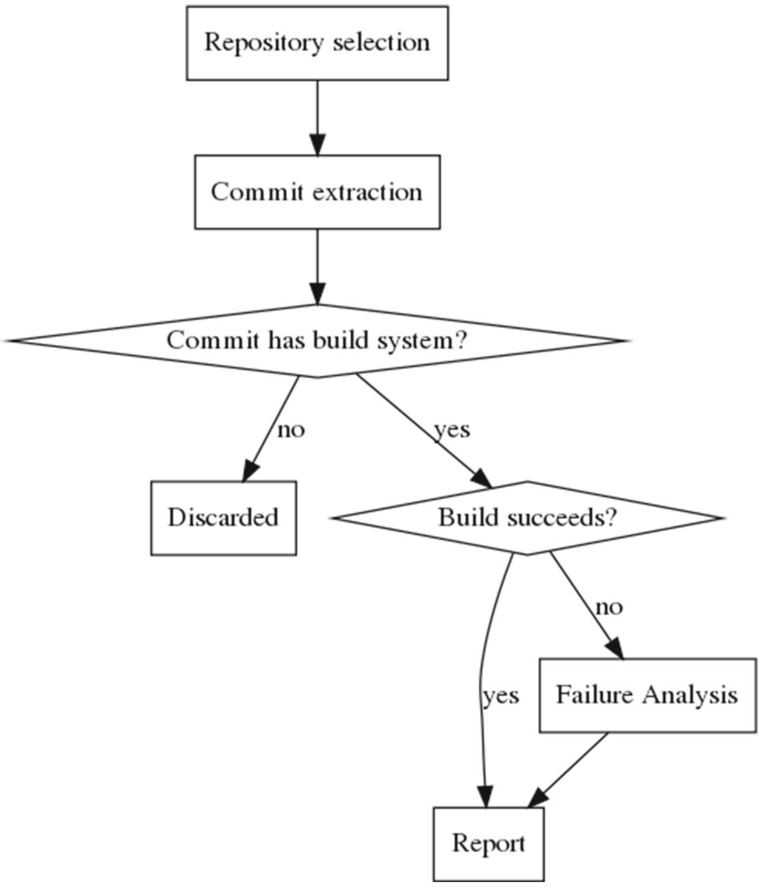

Fig. 1 Basic workflow for both studies

Table 1 Studies (main quantities)

Studies	Original Pristine	Original Reduced	Replic.	Reprod.			
Repositories	100	79	79	80			
Commits	174,505	139,389	139,389	300,873			
Commits (build conf.)	132,484	101,811	101,811	281,487			
Commits (build success)	31,696	22,737	14,664	98,488			
Commits/project	min	25%	50%	mean	75%	max	std
Replication	25	234	726	1,764	1,898	14,818	2,694
Reproduction	1,132	1,974	2,980	3,760	4,847	10,000	2,404

4.1.1 Subject Recovery

Our first step was to retrieve the git repositories to analyze. We wanted to clone all the repositories, to be able of checking out each specific commit, and analyze its compilability. We were interested in doing a replication as close as possible, so we decided to use only the repositories for which we could find all the commits of the original study. This way, we ensured that results would be comparable, and not influenced by a potentially biased sample of missing commits.

We started by using the list of git repositories from the original study to clone and check all of them. We noticed that some were not available or did not have all the commits considered in the original study. From a total of 100 repositories in the list, 6 were no longer available. Before discarding them, we tried to find them both in the ASF git repositories, and in the GitHub repositories that the foundation maintains as replicas of the original ASF-hosted ones. In addition, of those that we could clone, 19 did not have all of the commits considered in the original study (8 had none of them, 11 had only some of them). This resulted in a total of 75 repositories with the complete set of commits considered in the original study. We followed this procedure during February 2020.

To improve the number of repositories with all commits from the original study, we turned on to Software Heritage (SH), an initiative to collect, preserve and share all public source code in a universal software archive (Di Cosmo and Zacchiroli 2017; Di Cosmo 2018). SH tries to archive all commits, even if they are later removed from the original repositories. Therefore, it was an option for finding the missing commits. Although its API is still evolving, during March 2020 we could use it to retrieve some of the repositories with missing commits, or which we could not find.[3]

We found all the 25 remaining repositories in SH, but as we retrieved them using their API, 5 of them were empty or corrupt, and of the other 20, only 4 contained all the commits of the original study.

Therefore, the dataset that we used for our replication study consisted of 79 repositories out of the 100 in the original study, amounting for 139,389 commits from the total of 174,505 commits (79,9%). Even when this is only a fraction of the commits, we consider that the sample is large enough to conduct the rest of the study, as follows.

4.1.2 Building

Once we cloned all git repositories, we proceeded to replicate the experiment. For that, we checked out, one by one, all source code snapshots, each one corresponding to one commit, and tried to build them. In the original study they used some Java program to run the Maven tool, via its Java API, to build the code. However, we could not find the code in their reproduction package. Because of this, but also because we wanted to produce a tooling-independent replication, we developed a Python script that uses Maven through its command-line interface. The design of the script allowed to include other build systems, to be used in the reproduction study.

[3]In later conversations with representatives of Software Heritage, we learned that the part of the API for retrieving full repositories had been removed because it failed in some cases, which could explain our problems in retrieving some of the repositories.

For each commit of interest in each repository, the script runs the following procedure (see also Fig. 2):

1. Check out the intended commit, obtaining the snapshot of the source code to be built.
2. Find the configuration for Maven (usually a `pom.xml` file). If it is found, the build command for Maven is executed (*mvn clean compile -X*) in a Docker container spawned for this specific analysis.
3. Collect the success code and the log produced by the execution in a log file (the verbose option of the command is used to obtain the most detailed log). If a configuration for Maven was not found, it is also noted in the log file.

A further analysis of the log file allowed us to identify if the snapshot had configuration files for Maven, if the build was successful or not, and if not, the likely reason for the failure.

It is important to ensure the build environment is fully clean from results of previous builds, such as dependency modules or intermediate files that could influence the current one. To enforce this cleanup, besides the build command cleaning up local folders, the execution is encapsulated in a Docker container containing Maven and Java 8.

When checking for the presence of a Maven configuration file, we found an important replication issue: in all projects but three, those files were in exactly the same commits than the original study. But in those three, we found many more commits with Maven configuration (in the order of 6,000). We carefully inspected the checkout for a large sample of those commits, and our heuristics seem to work well. Unfortunately, this has some impact on the results of the replication, especially since a fraction of them are actually compilable. For having a more usable comparison with the results of the original study, we decided to analyze those commits separately (see details in Section 5), so that the results on how compilability "ages" are not influenced because of them. That is the reason why the number of "Commits (build conf.)" in Table 1 is exactly the same (101,811) in both the original study (reduced) and our study (for the 79 considered repositories).

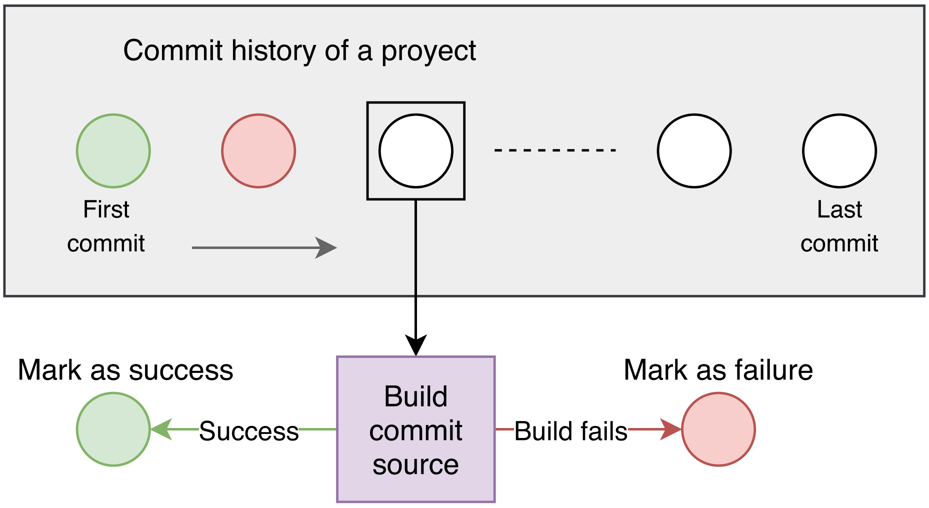

Fig. 2 Process for checking compilability status of commits in a repository. "Build commit source" includes checking out the commit, checking for Maven configuration files, and if found, running Maven to try to build the code

When searching for build files (pom.xml), we discovered that we were able to detect 4.72% more commits with build files than the authors of the original study. Detecting the presence of these build files is a relatively simple task, as they are easy to identify if they exist. Nonetheless, to check if this inconsistency was due to an error in our scripts, we randomly selected a dozen commits where we found build files but the original study did not, and performed a manual inspection. We found that our approach is the correct one. So, probably the discrepancy is caused by some bug in the scripts of the original paper — as these scripts are not publicly available, we cannot confirm this possibility.

To explore the compilability of all of the snapshots, we used two Ubuntu 18.04 machines, one of them with 16 cores and 8 GB of RAM and the other one with 8 cores and 16 GB of RAM. The software we built was capable of balancing the load between both servers to minimize execution time, building several snapshots in parallel. Once the repositories to analyze were ready, the building of all snapshots took about four weeks of wall time to run.

4.1.3 Obtaining Results

The results of the study are obtained from analyzing the log file. From it, we can know if a Maven configuration was found for a commit, or if the build was successful. For the analysis on the causes of build failures, we analyze the exception produced by Maven from the log for the failed builds. The exceptions that can be produced during the construction of the Maven project are well defined and limited.[4] For the reporting, we use the classification and mapping of exceptions in the original study, which defined four categories of errors: *Resolution* (related resolution of artifacts, such as downloading of dependencies), *Parsing* (such as malformed build configuration), *Compilation* (during the compilation phase) and *Other*. Note that the first three correspond to the first three steps in the building process described in Sulír and Porubän (2016), presented in Section 2.

For comparing our results with the original study, we also classified every commit according to how it behaved when building it in both studies: **(1) stable build**, the snapshot was built in both studies; **(2) new error**, the snapshot was built in the original study, but not in ours; **(3) same error**, the build failed in both studies for the same reason; **(4) different error**, the build failed in both studies but for a different reason; and, **(5) new build**, the build failed in the original study but not in ours.

We tried to reproduce the methods of the original study as much as possible, using the same classification to enable a comparison as is mandatory in a replication study. However, we did not do it manually but automated the procedure. We took therefore advantage of the description of the Java exceptions used in their classification by the original authors in their reproduction package.[5] As each exception is mapped to exactly a single category, our tool took this mapping and applied it automatically, avoiding the necessity for a manual classification. We think this highlights one of the main reasons for replication studies: to be able to detect and fix limitations in previous works. Our tool and data sources are publicly available in our reproduction package.

[4] https://cwiki.apache.org/confluence/display/MAVEN/Errors+and+Solutions

[5] http://www.cs.wm.edu/semeru/data/breaking-changes/

4.2 Reproduction Study

4.2.1 Subject Selection and Recovery

For our reproduction study, we generated a new dataset of repositories to analyze. Inspired by Sulír and Porubän (2016), we obtained a long list of repositories via the GitHub API, meeting the following criteria:

– *Java as the programming language*. We wanted to check Java building technologies, staying in the same domain of the original study.
– *At least 500 stars and 300 forks*. We wanted some indicator of relevance.
– *At least five years of development*. We wanted to have a long enough commit history, so that the analysis was more complete.
– *Active in January 2020* (at least one commit). We wanted projects with recent activity, to include current practices.
– *Use a build system*. We wanted to check compilability, so we checked that they were using Maven, Gradle or Ant (the three most popular build systems for Java) in the last commit.
– *Between 1,000 and 10,000 commits*. We wanted to avoid projects too small, which would have few snapshots to analyze, but also very large ones, which would consume too many resources for the analysis.
– *No Android projects*. Because of a practical limitation: building projects for Android is in general more complex, and requires specific procedures.

From the long list meeting all these conditions, we randomly selected repositories for our reproduction study and proceeded to their analysis. For each repository, we considered all commits in the master branch as the commits of interest for the study. A total of 80 projects have been selected for this reproduction study. The total number of commits was 300,873. When comparing the resulting list with the list of projects from the original analysis, in addition to variety (since the previous analysis was focused only in ASF projects), the main difference is that we focused in non-small projects with certain relevance.

4.2.2 Building

The process we followed to explore the buildability of snapshots for the repositories in our list was very similar to the one described above for the replication study. The differences were as follows:

– After checking out a commit, three systems are considered when searching for build configuration (see “Build File” in Table 2). In the replication study only Maven was considered.
– When building the snapshot with more than one build system, we tried the build systems in order: first Maven or Gradle, and if it failed, then Ant. We did not find snapshots with both Maven and Gradle. Having Ant and one of Maven or Gradle is usually due to the project transitioning from the former to the latter, thus still having the old Ant configuration and a new configuration for Maven or Gradle. Our order for testing systems considers that if the Maven or Gradle configuration worked, the project had likely already transitioned to them — if it did not, the project was still with Ant.
– For each build configuration, we executed the build command defined by the build configuration (see “Command” in Table 2).

Table 2 Build systems considered in the reproduction study

Build System	Command	Build File
Maven	mvn clean compile -X	pom.xml
Gradle	./gradlew build -x test	build.gradle
Ant	ant compile	build.xml

For this reproduction study, the Docker container we used included Java 8, Ant and Maven, while Gradle was run standalone (since it works self-contained). The building of all snapshots took about two weeks of wall time, in the same environment we used for the replication study.

4.2.3 Obtaining Results

As we did for the replication study, the results of the reproduction study are obtained by analyzing the logs of the attempts to build each considered commit from our list of repositories. The analysis is the same that we described already, with the difference that in the reproduction study we considered not only if the snapshot could be built or not, but also which build system was used. We also had to extend the mapping of exceptions in error logs to one of the four categories of errors in the original study (*Dependency*, *Parsing*, *Compiling* or *Other*). The new error mapping will be shown in the following sections,as well as being available in the reproduction package.

5 Replication Study: Results

5.1 Data for Replication

The original paper comes with a reproduction package, which we have found tremendously useful. It includes the complete list of commit hashes for all the analyzed repositories, and counts per repository of the main results. This allowed us to compute counts for our reduced set of 79 repositories. This way, we could do a fair comparison between the replication and the original results.

After computing our results for the 79 repositories in the original study, we found a discrepancy in the total number of commits. According to the original paper, 219,395 commits were analyzed. But computing from the list of commit hashes, we counted 174,505. To be consistent with other data in our paper, which was extracted from that reproduction package, we used the second number in our tables. This fact does not impact the results, since we focus on the analysis of the reduced set of 79 repositories.

The scripts used for the original study are not available in its reproduction package. This is not a problem in itself, since we created our own software. But we could not determine the reason for some discrepancies, e.g., on the number of snapshots with build configuration (see details below). Maybe the heuristics coded into the original software took into account something that we missed, or maybe it didn't detect that configuration information in some cases.

Detailed logs of the execution are also not available in the reproduction package of the original study, making it difficult to explain some differences found in the compilability of a large number of commits in three repositories (see details below): we do not know if there was some error in the execution of the original study, or if we are missing something in our own.

RQ$_{1d}$: "Is the data in the reproduction package of the original study enough for a replication?" There is enough data for comparing results, even at the repository level. However, there is not enough data to exactly reproduce the original study: a copy of the git repositories, as they were when the analysis was performed, is missing; some relevant software is also missing (such as the one to detect the Maven configuration in a snapshot); and the logs of their execution for all snapshots is also missing, which makes it difficult to understand the reasons of some discrepancies when replicating the study.

5.2 Compilability

To answer RQ$_{1a}$ and RQ$_{1c}$ in detail, we follow the same process as the original study, classifying repositories in three categories according to their number of commits: short history (number of commits in the first quartile, Q1), medium history (Q2 and Q3) and long history (Q4). Repositories classified as short (20) have less than 203 commits, medium (39 repositories) have between 203 and 1,148 commits, and long repositories (20) have more than 1,148 commits. Then, we analyze the compilability of those snapshots for which we found configuration for Maven, assuming the rest could not be built because developers were not using automatic tools for building.

Tables 3 and 4 shows the results for compilability in the original study, and of our replication study. For the original study we computed, thanks to the details in its reproduction package, results for the reduced list of 79 repositories that we have analyzed. However, the results for the pristine study (with all its 100 repositories) are very similar to those of the reduced one. For example, the average compilability of the reduced study is 37.19, while for the pristine study it is 38.13, according to the original paper. For completeness, we run Wilcoxon's test on the compilability results of all projects of the previous and our replication studies, obtaining a *p*-value of 1.9077e-6. This confirms statistical significant differences between both cases.

RQ$_{1a}$: "How many snapshots from the change history are compilable?" 8.74% or less of the commits were compilable for half of the analyzed projects. The result for the original study was 28.64%. Thus, results are very different, likely affected by the time passed (as we will discuss when answering RQ$_{1c}$).

RQ$_{1c}$: "Has compilability degraded since the original study?" A much smaller fraction of the commits with build configuration can be built (22.33% in the original study versus 14.66% in our replication). When analyzing repositories by size, medium projects are more affected by this decrease in compilability, with half of them having a compilability of 1.70% or less, while in the original study that number was 36.24%. Short projects are also affected, and the less affected are long projects, but still they move from a median of 16.48% to 9.44% in our replication.

We found significant differences in three repositories when checking for the existence of Maven configuration files (for all the others, results are exactly the same). In those three repositories we found 6,583 extra commits with a Maven configuration. When trying to

Table 3 Compilable snapshots — Distribution (top: Reduced Original Study, bottom: Replication Study)

Projects	Fraction built (in %)						
	Min	1st Qu.	Median	Mean	3rd Qu.	Max	SD
Short history	0.00	23.67	48.64	49.85	82.73	97.01	35.08
Medium history	0.00	4.99	36.24	38.43	68.42	100.00	34.03
Long history	0.00	0.00	16.48	22.14	25.67	100.00	29.33
All	0.00	2.91	28.64	37.19	66.00	100.00	34.25

Projects	Fraction built (in %)						
	Min	1st Qu.	Median	Mean	3rd Qu.	Max	SD
Short history	0.00	0.55	31.49	40.62	82.72	99.01	39.97
Medium history	0.00	0.00	1.70	20.99	38.24	95.23	29.52
Long history	0.00	0.00	9.44	17.56	19.17	100.00	28.31
All	0.00	0.00	8.74	25.1	41.64	100.00	33.07

build them, we could build 442 extra snapshots, which would increase the number of compilable snapshots to 15,106 (from a total of 108,394), or 13.93% compilability, slightly less than 14,40 as shown in Table 4.

We also checked all snapshots for configuration files of Ant and Gradle (we only found for Ant), finding 2,586 additional snapshots that could be built, leading to a total of 17,692 commits built. However, these results are not comparable with the original study, since they did not consider Ant.

Table 5 summarizes the results of all replications, with the different cases. It should be noted that Table 3 extends the information of the results reported in the "Replication" row of Table 5, as it is the most comparable scenario between the original and the replication study.

Table 4 Compilable snapshots — Totals (top: Reduced Original Study, bottom: Replication Study)

Projects	Total commits		
	With build conf.	Build success	Fraction built
Short history	2,311	1,129	48.85%
Medium history	26,903	9,602	35.69%
Long history	72,597	12,006	16.54%
All	101,811	22,737	22.33%
Short history	2,311	876	37.91%
Medium history	26,903	6,049	22.48%
Long history	72,597	7,739	10.66%
All	101,811	14,664	14.40%

Table 5 Original and replication studies (Summary of results). Replication$_a$: use same build configs, Replication$_b$: use all detected Maven config files, Replication$_c$: use all detected Maven and Ant config files

	Repos	All Commits	Commits w/ build conf	Commits built	Built total	Built mean of projects
Pristine Original	100	174,505	132,484	31,696	23.92%	37.74%
Reduced Original	79	139,389	101,811	22,737	22.33%	37.19%
Replication$_a$	79	139,389	101,811	14,664	14.40%	25.09%
Replication$_b$	79	139,389	108,394	15,106	13.93%	25.42%
Replication$_c$	79	139,389	117,124	17,692	15.10%	24.85%

5.3 Failure Analysis

To answer RQ$_{1b}$, and provide some insight on the reasons for the answer to RQ$_{1c}$, we analyzed build failures following the categorization of the original study. Table 6 reports the classification results for the previous study and for our replication, in both cases for the 79 repositories of the reduced study.

Table 7 shows the changes we observed when building snapshots with respect to the original study, according to the classification provided in Section 4. We could build 14,480 (63.68%) of the successful builds in the original study (*stable builds*). We could also build some snapshots that could not be built in the original study. In general, these correspond to external artifacts that were not available when they run their experiments, but were available when we run ours. These *new build* snapshots amount to 184 (0.13%).

For 67,298 of the snapshots that failed to build, we reported the same exception as in the previous experiment (*same error*). Therefore, adding these to *stable build*, 81.778 (80.66% of the snapshots with build configuration) behaved exactly as in the original study. On the other hand, for 13.30% of the snapshots that failed for us, we found a different error than in the original study (*different error*), while 9.48% of the snapshots that failed for us did not fail in the original study (*new error*).

Both in *different error* and in *new error*, one of the main causes for failed builds is *Resolution*, and specifically, the resolution of external artifacts needed for the build. We can affirm that one of the main reasons for the degradation in compilability with time is the impossibility of recovering more and more of the dependencies of these projects as time passes. On the other hand, *Compilation* errors only appear in *same error*, since they only depend on using the same compiler (so if they failed in the replication, they should have

Table 6 Causes of failed builds

	Original study		Replication study	
Categories	#	%	#	%
Resolution	47,832	60.49	55,299	63.45
Parsing	9,595	12.13	10,350	11.88
Compilation	2,807	3.55	2,544	2.92
Other	18,839	23.83	18,954	21.75
All errors	79,073	100.00	87,147	100.00

Table 7 Changes in builds (original to replication)

Categories	Errors						Builds	
	Same		Different		New		Stable	New
	#	%	#	%	#	%	#	#
Resolution	42,826	49.14	7,605	5.39	4,868	5.59		
Parsing	9,485	10.88	694	0.79	171	0.20		
Compilation	2,544	2.91	0	0.00	0	0.00		
Other	12,443	14.27	3,293	7.10	3,218	3.69		
All	67,298	77.22	11,592	13.30	8,257	9.48	14,480	184

failed in the original study). Comparing the number of *Compilation* errors in the replication and in the original study, it decreased from 2,807 to 2,544. Some inspection raised the cause: new exceptions mask previous building errors.

We studied in some detail the *Resolution* category. One of the most recurrent cause was experiencing timeouts when accessing some external artifact, usually a dependency (5.12% of total errors). This type of error appears only as *different error* or *new error*. Given that no long network failures were observed during the execution of our study, this is due to access to artifacts that are no longer available, which is consistent with the previous comment on dependencies.

RQ$_{1b}$: "What types of errors prevent snapshots from being built?" The proportion of the different kinds of errors is similar to the original study. Most of the errors (63.45%) are caused by resolution of dependencies and other external artifacts (60.49% for the original study), these being the errors that contributed the most to the degradation of compilability. Parsing errors convey 11.88% of the failures we detected, close to the 12.13% reported in the original study. The same can be said for Compilation errors (2.92% in our replication *vs.* 3.55% in the original study). About 80% of the snapshots behaved exactly like in the previous study (they built, or they failed with the same error).

6 Reproduction Study: Results

6.1 Compilability

To answer RQ$_{2a}$, we repeated the same analysis of the replication study, but now for the 80 projects of our reproduction study. Of them, 20 projects were classified as having a short history (Q1, less than 1,974 commits), 40 as medium history (Q2 and Q3), and 20 as long (Q4, more than 4,847 commits). Compilability results are shown in Table 8.

For all categories, and for all the projects together, median and mean are close, meaning the distribution is not very skewed: to some extent we were successful in avoiding bias when selecting projects to analyze. The standard deviation shows a large spread in the values (compatible with the values observed for the different quartiles). The fraction of snapshots that could not be built in all categories is higher than (or very close to) 50%, which shows that automatic compilability of past snapshots can certainly not be given for

Table 8 Reproduction study — Compilability results

Projects	Fraction built (in %)							Total commits		
	Min	1st Qu.	Median	Mean	3rd Qu.	Max	SD	With build conf.	Build success	Fraction built
Short history	0.00	37.43	55.42	58.21	82.27	99.38	28.12	28,312	15,948	56.32
Medium history	0.00	10.56	34.94	40.35	68.60	100.00	33.34	114,716	44,688	38.96%
Long history	0.00	4.27	12.80	27.76	48.04	93.52	31.76	138,459	37,852	27.33%
All	0.00	10.56	36.25	41.66	70.62	100.00	33.18	281,487	98,488	34.98%

granted. Projects with a short history have in general less compilability, and compilability increases as we move to projects with a medium and large history.

It should be noted that in the original study, we found a great diversity in the size of the projects according to their number of commits (from projects with 25 commits to 14k). The problem with smaller projects (<203 commits) is that in many cases they have not had a natural development (the initial commit imports code from another source, as can be seen for example in following commit).[6] This means that part of the project's development history is lost, which is a limitation of the original study. In the selection criteria for the repository mining we have taken this limitation into account, setting a lower limit of 1000 commits. Because of this difference between experiments, the separation between small, medium and large projects is different and the groups cannot be compared between experiments. Nonetheless, for the sake of completeness, we offer a comparison using the same quartiles to classify the projects into the three categories in a table that can be found in the replication package.

RQ$_{2a}$: "How many snapshots from the change history are compilable?" Less than half of the snapshots with build configuration could be built. Compilability increased with the size of the commit history of projects. For all history size categories, and all projects together, mean and median compilability was less than 51%, although variability is high (standard deviation is over 28% in all cases).

6.2 Failure Analysis

To answer RQ$_{2b}$, we analyzed build failures, as shown in Table 10. Most of the errors are of type *Resolution*, due to missing dependencies and other external artifacts. *Parsing* and *Compilation* errors are almost negligible.

The fact that most errors are due to missing external objects is a good indicator of further decline of compilability in the future: as time passes, more and more artifacts will disappear, which will lead to more and more commits not building. The low numbers for *Parsing* and *Compilation* errors are interesting: they could be indicating that the use of automatic tools to check the compilability of snapshots (and therefore, the consistency of building configuration, and the correctness of compilation) before merging the corresponding commits.

[6] https://github.com/apache/maven-app-engine/commit/b77ad9e82d7cc94627d0a000214aeb6c4b8f7738

Table 9 Reproduction study — Compilability by build system

Build	Compilability		
System	#Commits	#Build success	Fraction built
Maven	191,063	80,310	42.03%
Ant	25,735	4,441	17.26%
Gradle	64,095	13,737	21.43%
All	280,893	98,488	35.06%

RQ$_{2b}$: "What types of errors prevent snapshots from being built?" Most of the errors observed (56%) are due to missing dependencies and other external artifacts. The impact of these errors in the overall compilability is thus very important: almost one third (36.43%) of all snapshots with building configuration could not be built because of them. On the contrary, parsing and compilation errors are few, meaning that most snapshots were carefully checked for these errors before committing.

6.3 Building System

To answer RQ$_{2c}$, we performed an analysis by build system. When we were detecting build configuration files, we observed some projects starting to build with Ant, then switching to Maven or Gradle, and some others using Maven and Gradle for their entire history. From the total number of commits with build configuration, 68.02% are for Maven, 22.82% for Gradle, and 9.16% for Ant. The results of the analysis by build system are shown in Table 9.

The first interesting result is that compilability for snapshots with Maven is much higher (42.03%) than for Ant or Gradle (17-22%). With respect to specific kinds of errors, Maven is dominated by *Resolution*, as Gradle and Ant is dominated by *Other*. However, the sample of snapshots with Ant and Gradle is relatively small, which means our results for them are less reliable (Table 10).

The detailed classification of errors for all building systems can be found in the Table 11 (The table is limited to the 20 most frequent errors. The full table can be found in the reproduction package). From this classification, it is worth mentioning:

– From the 15,732 *other errors* for Ant, 8,615 are failures because they do not have the "compile" task defined (which is a standard in the development of projects with Ant),

Table 10 Reproduction study — Errors by build system

Build	Resolution Err.		Parsing Err.		Compilation Err.		Other Err.	
System	#	%	#	%	#	%	#	%
Maven	75,806	68.45%	1,967	1.78%	0	0.00%	32,980	29.78%
Ant	4,485	21.06%	0	0.00%	1,077	5.06%	15,732	73.88%
Gradle	22,260	44.20%	0	0.00%	2,373	4.71%	25,725	51.08%
All	102,551	56.22%	1,967	1.08%	3,450	1.89%	74,437	40.81%

Table 11 Classification of errors for all build systems

Build System	Error	Action	Count	%
Maven	ArtifactResolutionException	Resolution	74,950	41.09
Maven	MojoFailureException	Other	12,426	6.81
Maven	MojoExecutionException	Other	11,562	6.34
Ant	Target “compile” does not exist	Other	8,615	4.72
Gradle	IOException: Server returned ...	Resolution	8,057	4.42
Gradle	Could not resolve	Resolution	7,057	3.87
Gradle	Could not resolve all dependencies	Resolution	6,899	3.78
Maven	ConnectException	Other	5,524	3.03
Gradle	Task “test” not found	Other	5,215	2.86
Gradle	Execution failed for task	Other	4,726	2.59
Gradle	Cannot run program	Other	4,540	2.49
Ant	UnknownHostException	Resolution	3,918	2.15
Gradle	Other Gradle error	Other	3,849	2.11
Ant	Unable to find property file	Other	3,693	2.02
Gradle	Permission denied	Other	2,295	1.26
Gradle	Compilation failed	Compilation	2,288	1.25
Maven	PluginDescriptorParsingException	Parsing	1,591	0.87
Gradle	Could not find	Other	1,405	0.77
Ant	Other Ant error	Other	1,181	0.65
Gradle	git-lfs is required to build	Other	1,083	0.59
Ant	Compile failed	Compilation	1,077	0.59

and about 4,610 because a specific binary or config file is expected in the path, instead of being included as a dependency.

– From the 25,725 *other errors* for Gradle, 5,215 of them fail because of a test that cannot be automatically ignored. A common error (4,726) is that some of the tasks, defined as scripts in the configuration file by the developer, fail to build the project. Other less common errors are the absence of a secret key (which is not usually stored in a public repository), an error in the generation of documentation or the lack of permissions for certain system folders. These errors, although a priori generic, are usually concentrated in a single project and are not significant.

Although all of these snapshots could not be built automatically, they could be built with minimal human intervention, or with a slightly improved set-up of the build configuration for the projects affected. Given the relatively small sample of snapshots for these two systems (Ant and Gradle, compared to Maven), considering these snapshots as errors or not would change dramatically the figures for the *other errors* category for both of them. In the reporting of results, we have decided to consider those cases as errors, which is a cause for the lower compilability reported for Ant and Gradle.

RQ$_{2c}$: "Are there differences in compilability depending on the building tool?" The compilability observed in commits whose build system is Maven is twice as high as that obtained for Gradle and Ant. Resolution errors are dominant for Maven (68.45%), parsing errors are very low (1,96% for Maven and 0% for Ant and Gradle), and compilation errors seem to be only relevant for Ant (5.06%, 4.71% for Gradle, 0% for Maven). We note that both Ant and Gradle use user-written tasks or scripts that are subject to errors, mainly because they do not follow tool standards, which negatively affects their compilability.

7 Discussion

When performing our studies, we wanted to learn about the validity of the results presented in Tufano et al. (2017), by checking how they had changed after about six years, and about their generality, by doing a similar analysis on a different set of projects. As a side result, we also expected to learn about conditions for the replication of this kind of studies, based on its very detailed reproduction package. We discuss our findings in the next subsections.

7.1 Reproduction Package

The data in the reproduction package was not enough for a complete replication, although it helped substantially. The lists of repositories analyzed, commit hashes, and results per project were certainly the most useful data for us. With the list of repositories we could search for them in the ASF git repositories, in GitHub, and in Software Heritage. The list of hashes was fundamental for checking if those commits were still recoverable from any of these sites and the list of results per project was fundamental to compare with the original results once we decided to analyze only a part of the repositories in their analysis.

However, we also missed some data. In our search, we could not find some repositories. For 25% of them we could not find all commits. Since git repositories may disappear, or commits in them can be removed, having a copy of the git repositories as they were analyzed would have allowed us for a complete reproduction. For the number of repositories analyzed, this is a massive dump, but small enough to be preserved in sites like Zenodo, which admits very large datasets. Software Heritage is also a very interesting option. At the time of the original study, they were still starting. But now, they are already archiving probably all code of interest for this kind of studies, and they are open to archive more if needed. Since they preserve all changes to the repositories they track, all commits studied will be available for future research. Unfortunately, these types of problems we have encountered pertain to any study involving the analysis of a project's history. Kalliamvakou et al. (2016) identified many perils that can be encountered when mining software repositories that are important to consider when conducting MSR-like studies. However, they do not include the perils that we have encountered, which negatively affect the replication of these studies. Future lists of perils should include those that affect the integrity of the repository over time:

- The repository may no longer be accessible (it has been deleted or made private).
- Commits have been deleted

We also missed the software used for building the snapshots. Even when the methodology is clearly explained, we expect some details to be only available by reading the software. The lack of it prevented us from determining with certainty if the discrepancies in the number of builds for some repositories were due to errors in the original software, or in ours. For the same reason, we missed the logs of the original experiments. With them, we could have determined with more detail some of the discrepancies we found.

In summary, we recommend to include the following items in the reproduction package for this kind of studies: references to the repositories, list of commits, complete copy of the repositories, or references to the analyzed commits in Software Heritage, results at the finer grain possible (at least at the level of repositories), a copy of the software used (at least for the building of the commits), and a copy of the logs of executing it. Our reproduction package (see "Acknowledgments and reproducibility", at the end of this paper) includes all of these artifacts, plus the software used for the calculation of the results (as Python notebooks), with intermediate results. All of this is provided for both the replication and the reproduction studies.

7.2 Replication Study

We have found less compilability than the original study. Some dependencies cease to be available with time, and as time passes, more and more of them are missing, impacting on the compilability of more and more snapshots. This depends on the kind of dependencies: official packages, in the case of Maven, are usually preserved. However, we have found many cases of specific links, or interim packages, that are much less reliable. We think this fact shows the importance of building strong configuration files for the build tools, that are designed to be resilient to the passing of time. Including references to artifacts that will be preserved (including, but not limited to, dependencies) as much as possible is an important part of it. However, other errors due to compilation, parsing, etc., have remained stable, as it seems reasonable, because we are using basically the same tools for building the source code.

The length of the history in commits seems to have an impact on the changes from the original study to our replication. Projects with medium histories went from a median of about 36% of built commits, to almost 0% (1.7%). We do not know if this is due to something specific in the sample of projects, or if there is some effect worth researching. It is also noticeable how the standard deviation of the distribution of built commits per project remains very stable.

The overall numbers for compilability are rather small. For half the projects analyzed, less than 10% of the commits can be built, which for practical purposes means most of the history of the project cannot be built automatically. This leads to an interesting detail: we have tried to build snapshots automatically, but maybe a manual approach —looking at the specific errors and trying to fix them– could reach a much better overall compilability. However this approach, although interesting, is not of much practical use in many of the common scenarios in industry.

7.3 Reproduction Study

The main reason for the reproduction study was to explore if the results of the original study were extensible to other, more diverse, Java programs. From the point of view of compilability, our results show only a partial extensibility. Mean compilability of projects is relatively similar (41% in our reproduction, 37% in the original study). Maybe the main conclusion is that a large number of snapshots in both cases are not compilable (clearly more than half, in both cases), showing the difficulties of past compilability.

When considering the kinds of errors leading to failed builds, it is clear in both studies the importance of *Resolution* errors: about 60% of all errors in the original study, 56% in our reproduction. Both studies offer similar results for *Compilation* errors: about 4% and 2%. We can conclude that the most influential errors, at least in the case of Java, are indeed *Resolution*, and that *Compilation* errors are not significant. Therefore, if a project wants to work on improving their past compilability, they should very likely work on improving the preservation of external artifacts, and on using references with guaranteed future availability. Clear rules could be derived, to ensure better configuration files for building tools, not degrading compilability as time passes. There is, however, little to do by the compiler to improve the situation, maybe due to the good work that Java does in ensuring backwards compatibility, or to the pre-merge testing that is usually done before producing new commits. These results are also in line with those discussed in the original paper.

We also studied the impact of different build systems on past compilability, which was not done in the original analysis. In our sample, we can say that snapshots using Maven show in general better compilability (about 42%, as opposed to 17% and 21% for Ant and Gradle). The numbers for Ant could be underestimated in our study, as we explained, but the numbers for Gradle seem clear, despite the smaller sample of Gradle snapshots. A more extensive study should be performed to know if this is really a difference between the systems, or just a particularity of our sample.

7.4 Mitigation Measures

The large number of errors that occur when trying to build commits from the past leads us to think about what possible measures we can take to reduce their occurrence. In the literature we find a large amount of Automatic Repair work that could solve some of the problems we are facing. In this section we will highlight two works that directly address build errors in previous snapshots, which will help us to make some recommendations to mitigate build errors.

Vassallo et al. (2020) summarize the reasons why Maven builds fail and suggest possible solutions. The work includes a survey on how do developers approach different types of build failures. They consider dependency resolution issues to be the most difficult to address. Among the most difficult bugs to address are infrastructure bugs (entirely related to the execution context) and dependency resolution bugs. The former are not a problem in our case, as most projects that are mined from GitHub tend to be programming libraries. The latter are the most numerous in our experiments and highlight that the logs usually do not give much information about the error.

Macho et al. (2018) propose a more specific tool focusing on problems related to Maven dependencies. Their study on how developers tend to fix dependencies themselves shows that in most cases it is usually enough to make a version number increase or promote a version with the -SNAPSHOT suffix to a stable version. Among the strategies of the tool we can find: Version Update (based on the usual behavior followed by the developer), Dependency Delete (deleting a problematic dependency that is no longer needed for the project), and Add Repository (in many cases, the configuration of the repository to which the artifacts are to be requested is not registered in the code repository, a possible fix being to add this artifact repository to pom.xml).

Based on the information from previous work and our experience in analyzing the results of the experiments we have conducted, we state possible mitigations for the problems that prevent a snapshot from being built. We want to approach these mitigations from two points of view: (i) *the developer* (who must consider that in the future he may need to rebuild a particular version of his code, for example to include a patch), and (ii) *the researcher* (who performs experiments on the commit history of a project):

- **Dependency resolution**: These errors represent a significant number for all three building systems. In some cases the dependencies are not from an artifact repository, but require manual installation within the project, or we may find authentication errors as the repositories are private. Developers should ensure that changes uploaded to the code repository include stable versions of the libraries for subsequent download from the artifact repositories. In the case of needing an additional resource (e.g., a library compiled directly to a JAR executable file), this should be versioned along with the rest of the code so that the code repository is self-contained. So, we have found SNAPSHOT errors because a project is multi-module and some modules depend on build versions from other modules (as it is the case for io.spring.initializr:initializr-generator-spring:jar:0.8.0.BUILD-SNAPSHOT). Researchers can mitigate some of the errors, for example, by removing the -SNAPSHOT suffix or replacing the library version with the version set by the next commit history where the build does work.
- **Parsing**: The main mitigation measure for this type of errors is usually the encoding. We have found non-English characters in code comments (usually author names) that prevent compilation. Developers should make sure that the encoding is present in the project configuration file, so that it is taken into account when building the project. Researchers can check if this configuration exists and overwrite it if it is not correct; or if they cannot find it, they can try to use the most inclusive encoding possible.
- **Compilation**: These errors are often due to code that has been commented out and that simply cannot work and needs to be manually inspected, sometimes with some knowledge of the project's operation. In previous work, Tufano et al. considered that there were builds that were broken from conception; compilation bugs often fall into this category. Therefore, we have not established any mitigation measures for this type of bugs.
- **Other**: Errors in this category include all those that do not fit into any of the above categories. In the case of Gradle and Ant, this is the predominant category. In these build systems, unlike in Maven, it is the developer who defines through tasks or scripts the steps to be carried out to build the project. This can make it very difficult to automate the build if the developer has not used the standard tasks of the build system and has defined his own tasks (which can be subject to errors). Some of the most recurring errors encountered are: *Target "compile" does not exist* (Ant), *Task 'test' not found* (Gradle), and *Execution failed for task X* (Gradle). Developers should use standard task names. It is common that after a while another developer has to build a past version of the code and tries to build the project using standard tasks. Researchers should consider that at some points in the project, standard task names may not be used. The documentation can help in understanding how the project should be built, and if that fails, Continuous Integration (CI) configuration files can be inspected as they usually contains these build commands correctly versioned alongside the code.

7.5 Threats to Validity

Our studies are subject to construct validity issues, mostly due to how we define compilability of a snapshot (the snapshot is automatically compilable now, using only the available source code in the snapshot). The usual compilability is defined not for past commits, but for the current ones, but we are interested specifically in the case of building commits in the history of the project. In addition, it could be argued that a snapshot could be built with more advanced techniques (e.g., by extracting the build command from the documentation or, failing that, from the CI files), or by fixing by hand some of the errors. However, we were specifically interested in the automatic case, and we think the techniques we used for building are the usual ones.

Our results are also subject to internal validity issues, mainly because of our interpretation of what an error is, and to the specific heuristics that we use to detect and classify build failures. Also, to any kind of bug in our execution or analysis software. We make this information available in the reproduction package, so that these issues can be inspected.

Threats to conclusion validity could come from our interpretation of the results. We think that our statistical methods and interpretations are the usual ones. Just in any case, we share the Python notebooks we have used, so that they can be inspected.

Finally, we can also have external validity issues. We think we have at least partially mitigated those mentioned in the original paper, by generalizing results replicating the analysis, and performing the reproduction analysis with a more diverse and different dataset. Still, we have concerns of how general these results are for Java projects, even when they are coincident with the original study. Extension to other languages is still a matter of further research.

8 Conclusions and Future Work

In this paper we have shown a replication and a reproduction study by Tufano et al. (2017) about the compilability of the history of past commits of a project. In the first one we have repeated their analysis, with those repositories for which we found all commits, and in the second one we extended its generality by using the same methodology with a different, more diverse set of Java projects, and considering also Ant and Gradle in addition to Maven.

The main contributions of this paper are:

- A discussion and guidelines on reproduction packages for studies on the compilability of past snapshots.
- A dataset and software, usable by other researchers, to study long-term degradation of compilability.
- A partial validation of the results of the original study. In particular, results about frequency of errors causing build failures have been validated and extended.
- Evidence on how compilability degrades over time, and how it could be mitigated by ensuring future availability of dependencies.
- Evidence on the compilability of a different, more diverse set of Java projects, showing some differences with the original study.
- Evidence on how the building tools affect future compilability.

In summary, we wanted to shed some more light on to which extent past snapshots of projects are compilable “as such”, because that is the basis to know how much build-repair techniques are needed if past artifacts of a project need to be reproduced from source code. Since our study was a replication and a reproduction, a part of our results could be expected, but still they add more detail and evidence to the original study. In addition, we also found some differences, generalized evidence by analyzing a more diverse set of projects, and produced a tool to automate the analysis of any Java repository, which could be used in further studies by any researcher.

Still, more research is needed to draw general conclusions on the compilability of past snapshots, especially for languages other than Java, to get a more precise knowledge about how compilability degrades over time, and how this degradation could be mitigated.

9 Reproducibility

A reproduction package is available.[7] It includes a link to an extra package in Zenodo (due to size limitations in GitHub), with raw results and a copy of all the git repositories at the time of the analysis.

Author contribution All authors contributed to the study conception and design. Material preparation, data collection and analysis were performed by Michel Maes-Bermejo. The first draft of the manuscript was written by Michel Maes-Bermejo and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Declarations This research has not involved human participants and/or animals.

Competing interests This work has been supported by the Regional Government of Madrid (CM) under project EDGEDATA-CM (P2018/TCS-4499) cofunded by FSE & FEDER and by the Spanish Government under project BugBirth (RTI2018-101963-B-I00) cofunded by the Ministry of Economy and Competitiveness, FEDER & AEI.

References

Asaduzzaman M, Bullock MC, Roy CK, Schneider KA (2012) Bug introducing changes: A case study with Android. In: Proceedings of the 9th IEEE working conference on mining software repositories, MSR '12. IEEE Press, Piscataway, pp 116–119. http://dl.acm.org/citation.cfm?id=2664446.2664463

Bavota G, Canfora G, Di Penta M, Oliveto R, Panichella S (2015) How the apache community upgrades dependencies: an evolutionary study. Empir Softw Eng 20(5):1275–1317

Cartwright N (1991) Replicability, reproducibility, and robustness: comments on Harry Collins. Hist Polit Econ 23(1):143–155

Cito J, Schermann G, Wittern JE, Leitner P, Zumberi S, Gall HC (2017) An empirical analysis of the Docker container ecosystem on github. In: 2017 IEEE/ACM 14Th international conference on mining software repositories (MSR). IEEE, pp 323–333

de Carné de Carnavalet X, Mannan M (2014) Challenges and implications of verifiable builds for security-critical open-source software. In: Proceedings of the 30th annual computer security applications conference, ACSAC '14. ACM, USA, pp 16–25. https://doi.org/10.1145/2664243.2664288

[7] https://github.com/BuildabilityResearcher/BuildabilityStudy

Di Cosmo R (2018) Software Heritage: why and how we collect, preserve and share all the software source code. In: 2018 IEEE/ACM 40Th international conference on software engineering: Software engineering in society (ICSE-SEIS). IEEE, pp 2–2

Di Cosmo R, Zacchiroli S (2017) Software Heritage: Why and how to preserve software source code

Glukhova M (2017) Tools for Ensuring Reproducible Builds for Open-Source Software. Master's thesis, Lappeenranta University of Technology. http://lutpub.lut.fi/bitstream/handle/10024/135304/MariaGlukhova_ToolsForEnsuringReproducibleBuildsForOpenSourceSoftware.pdf?sequence=2

Hassan F, Mostafa S, Lam ES, Wang X (2017) Automatic building of Java projects in software repositories: A study on feasibility and challenges. In: 2017 ACM/IEEE International symposium on empirical software engineering and measurement (ESEM). IEEE, pp 38–47

Hassan F, Wang X (2018) Hirebuild: An automatic approach to history-driven repair of build scripts. In: 2018 IEEE/ACM 40Th international conference on software engineering (ICSE), pp 1078–1089

Juristo N, Gómez O. S. (2010) Replication of software engineering experiments. In: Empirical software engineering and verification. Springer, pp 60–88

Just R, Jalali D, Ernst MD (2014) Defects4j: a database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 international symposium on software testing and analysis, ISSTA 2014. ACM, USA, pp 437–440. https://doi.org/10.1145/2610384.2628055

Just R, Jalali D, Inozemtseva L, Ernst MD, Holmes R, Fraser G (2014) Are mutants a valid substitute for real faults in software testing? In: Proceedings of the 22Nd ACM SIGSOFT international symposium on foundations of software engineering, FSE 2014. ACM, USA, pp 654–665. https://doi.org/10.1145/2635868.2635929

Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2016) An in-depth study of the promises and perils of mining github. Empir Softw Eng 21(5):2035–2071

Macho C, McIntosh S, Pinzger M (2018) Automatically repairing dependency-related build breakage. In: 2018 IEEE 25Th international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 106–117

Manacero A (2011) Using binary code to build execution graph models for performance evaluation of parallel programs graph models

Maudoux G, Mens K (2018) Correct, efficient, and tailored: The future of build systems. IEEE Softw 35(2):32–37

Meneely A, Srinivasan H, Musa A, Tejeda AR, Mokary M, Spates B (2013) When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In: 2013 ACM/IEEE International symposium on empirical software engineering and measurement. IEEE, pp 65–74

Murgia A, Concas G, Marchesi M, Tonelli R (2010) A machine learning approach for text categorization of fixing-issue commits on CVS. In: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '10. ACM, New York, pp 6:1–6:10. https://doi.org/10.1145/1852786.1852794

Nikitin K, Kokoris-Kogias E, Jovanovic P, Gailly N, Gasser L, Khoffi I, Cappos J (2017) Ford, B.: {CHAINIAC}: Proactive software-update transparency via collectively signed skipchains and verified builds. In: 26Th {USENIX} security symposium ({USENIX} security 17), pp 1271–1287

Perry M, Schoen S, Steiner H (2014) Reproducible builds. moving beyond single points of failure for software distribution. In: Chaos communication congress

Raemaekers S, van Deursen A, Visser J (2012) Measuring software library stability through historical version analysis. In: 2012 28Th IEEE international conference on software maintenance (ICSM), pp 378–387. https://doi.org/10.1109/ICSM.2012.6405296

Rausch T, Hummer W, Leitner P, Schulte S (2017) 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 '17. IEEE Press, Piscataway, pp 345–355. https://doi.org/10.1109/MSR.2017.54

Ren Z, Jiang H, Xuan J, Yang Z (2018) Automated localization for unreproducible builds. In: Proceedings of the 40th International Conference on Software Engineering, ICSE '18. ACM, USA, pp 71–81. https://doi.org/10.1145/3180155.3180224

Reproducible builds (2017) https://reproducible-builds.org/ (Accessed May 20, 2021)

Reproducible builds in Debian (2018) https://wiki.debian.org/ReproducibleBuilds (Accessed May 20, 2021)

Seo H, Sadowski C, Elbaum S, Aftandilian E, Bowdidge R (2014) Programmers' build errors: A case study (at Google). In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014. ACM, USA, pp 724–734. https://doi.org/10.1145/2568225.2568255

Skrimstad Y (2018) Improving Trust in Software through Diverse Double-Compiling and Reproducible Builds. Master's thesis, University of Oslo. http://urn.nb.no/URN:NBN:no-68006

Śliwerski J., Zimmermann T, Zeller A (2005) When do changes induce fixes? In: Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR '05. ACM, USA, pp 1–5. https://doi.org/10.1145/1082983.1083147

Spinellis D (2012) Git. IEEE software 29(3):100–101

Sulír M, Porubän J (2016) A quantitative study of java software buildability. In: Proceedings of the 7th international workshop on evaluation and usability of programming languages and tools, PLATEAU 2016. ACM, USA, pp 17–25. https://doi.org/10.1145/3001878.3001882

Tian Y (2017) Mining software repositories for automatic software bug management from bug triaging to patch backporting. Ph.D. thesis, Singapore Management University

Tufano M, Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D (2017) There and back again: Can you compile that snapshot? J Softw Evolution Process 29(4):e1838

Vassallo C, Proksch S, Zemp T, Gall HC (2020) Every build you break: Developer-oriented assistance for build failure resolution. Empir Softw Eng 25(3):2218–2257

Zimmermann T, Kim S, Zeller A, Whitehead EJ Jr (2006) Mining version archives for co-changed lines. In: Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR '06. ACM, USA, pp 72–75. https://doi.org/10.1145/1137983.1138001

Zimmermann T, Nagappan N, Zeller A (2008) Predicting Bugs from History. Springer, Berlin, pp 69–88. https://doi.org/10.1007/978-3-540-76440-3_4

Affiliations

Michel Maes-Bermejo[1] · Micael Gallego[1] · Francisco Gortázar[1] · Gregorio Robles[2] · Jesus M. Gonzalez-Barahona[2]

Michel Maes-Bermejo
michel.maes@urjc.es

Micael Gallego
micael.gallego@urjc.es

Francisco Gortázar
francisco.gortazar@urjc.es

Jesus M. Gonzalez-Barahona
jesus.gonzalez.barahona@urjc.es

[1] Department of Computer Science, Universidad Rey Juan Carlos, Móstoles, Spain

[2] Department of Signal Theory and Communications and Telematics Systems and Computing, Universidad Rey Juan Carlos, Fuenlabrada, Madrid, Spain