WILEY Software: Evolution and Process

RESEARCH ARTICLE

There and back again: Can you compile that snapshot?

Michele Tufano[1] | **Fabio Palomba[2]** | **Gabriele Bavota[3]** | **Massimiliano Di Penta[4]** |
Rocco Oliveto[5] | **Andrea De Lucia[2]** | **Denys Poshyvanyk[1]**

[1]The College of William and Mary, Williamsburg, VA, USA

[2]University of Salerno, Fisciano, Italy

[3]Università della Svizzera italiana (USI), Lugano, Switzerland

[4]University of Sannio, Benevento, Italy

[5]University of Molise, Campobasso, Italy

Correspondence
Michele Tufano, The College of William and Mary, Williamsburg, VA, USA.
Email: mtufano@email.wm.edu

A broken snapshot represents a snapshot from a project's change history that cannot be compiled. Broken snapshots can have significant implications for researchers, as they could hinder any analysis of the past project history that requires code to be compiled. Noticeably, while some broken snapshots may be observable in change history repositories (e.g., no longer available dependencies), some of them may not necessarily happen during the actual development. In this paper, we systematically study the compilability of broken snapshots in 219 395 snapshots belonging to 100 Java projects from the Apache Software Foundation, all relying on Maven as an automated build tool. We investigated broken snapshots from 2 different perspectives: (1) how frequently they happen and (2) likely causes behind them. The empirical results indicate that broken snapshots occur in most (96%) of the projects we studied and that they are mainly due to problems related to the resolution of dependencies. On average, only 38% of the change history of the analyzed systems is *currently* successfully compilable.

KEYWORDS

broken snapshots, empirical studies, mining software repositories, software quality

1 | INTRODUCTION

A broken snapshot represents a snapshot from a project's change history that cannot be compiled.[*] The problem of having incompilable snapshots can have significant implications for researchers preventing the use of tools working on byte code as well as any sort of dynamic analysis. Other than representing problems for developers, as previous work has shown,[1-5] broken snapshots represent a strong limitation for researchers interested in running Mining Software Repositories (MSR) studies. To better illustrate the problem, let us consider the following scenarios.

Gabriele is a researcher studying the impact of refactoring activities on code smells. To this aim, *Gabriele* is using a smell detector, JDeodorant[6,7] and a refactoring detector, RefFinder.[8] All these tools work on byte code, hence require source code to be compiled. When trying to compile all 1000 revision snapshots of a project *A*, *Gabriele* realizes that there are 400 snapshots that failed to compile, hence, creating gaps in the observed project history, because JDeodorant and RefFinder cannot work on these snapshots. *Gabriele* feels that these gaps would seriously impact his analysis, especially because some of them occur in periods where release notes announced major

projects' restructuring. Certainly, *Gabriele* could decide to conduct the whole study at release level; however, refactoring actions can be precisely identified only at commit level granularity. Indeed, differencing 2 releases would introduce noise because multiple changes, not only refactorings, have likely occurred on the same artifacts. This represents a significant threat to validity for such analysis.[9]

A similar situation occurs to *Alex*, a researcher studying how the quality of test suites changes over time. *Alex* is interested in measuring the code coverage of the test suites in a project after each commit, to analyze the phenomenon at a fine-grained level (e.g., to understand the characteristics of commits that lead to a strong decrease in code coverage). To measure the code coverage *Alex* is using Cobertura.[†] Unfortunately, this tool also works on byte code, requiring the compilation of each project's snapshot that *Alex* is interested in analyzing. *Alex* is unable to compile about 450 out of the 900 projects' snapshots, i.e., half of them. He performs a deep investigation to identify the likely causes of lack of compilability and, for 100 snapshots, he realizes that the build script is unable to solve library dependencies, either because they refer to local files, or because the libraries are no longer available in the *Maven* central repository. He is able to solve the problem for 80 of them, by manually searching for the needed library version. However,

---

[*]In this paper we focus on broken snapshots based on compiling software, while other broken snapshots could be the result of test failures, deployment failures, *etc*

[†]http://cobertura.github.io/cobertura/

the whole investigation took 2 days of *Alex*'s effort, and still the achieved compilability rate is not enough for the analysis to be performed. As a consequence, *Alex* decides to change his plan, shifting the focus of the study to a coarser-grained level, only studying the quality of the test suite in each issued release (already compiled) of the analyzed project.

Clearly, the higher the rate of broken snapshots in the project, the higher the chances that *Gabriele* and *Alex* will not be able to derive useful (actionable) insights for the projects they are studying, because they will not be able to run the needed tools on those broken commits.

Paper contribution. While broken snapshots are significant research problems, there has been little effort in empirically exploring and understanding the root causes and implications of these problems. The goal of this paper is to fill this gap by systematically studying the compilability of Java project snapshots downloaded from Version Control Systems, with the purpose of understanding:

1. *how frequently* compile broken snapshots (in the following referred to as simply "broken snapshots") occur, and how long they last before the snapshot becomes compilable again in post-mortem analysis (noteworthy, we focus on compilability "here and now" for mining purposes, as opposed to compilability in the past for development purposes).
2. *why* broken snapshots occur, ie., what are the typical reasons behind their introduction.

The study has been conducted in the *context* of 100 Java open source projects by relying on *Maven* as an underlying build system. Overall, we downloaded, attempted to compile and analyze 219 395 snapshots of these 100 projects.

Motivations and Implications. The results of our study can be used from multiple *perspectives*. One straightforward perspective is the one of a researcher—or somebody interested in gaining actionable analytical insights for an evolving software project—mining change histories for various purposes.[10] Broken snapshots can hinder any analysis performed at commit level, because many such tools require byte code to work,[11-14] while specific analyses, e.g., related to dynamic analysis or test suite assessment, require the system to be executed. From this perspective, the higher rates of broken snapshots would either trigger tedious and costly repair activities (assuming they are even possible), or force the researcher to perform the analysis at a coarse-grained level, i.e., limit the analyses to compilable releases (and potentially loose useful data at commit level granularity). However, there are numerous examples in the research literature showing that the choice of granularity (commit *vs* release) is an important design decision, because analysis at different granularity may produce different or even contradictory results. For instance, the evolutionary study of code clone genealogies at commit level by Kim et al showed that a significant number of clones in software was short-lived.[15] However, another study of code clone genealogies at release level by Saha et al demonstrated contradictory results, finding that many code genealogies persisted in follow-up releases.[16,17]

More studies, related to fix-inducing refactorings[9] or to changes occurring on design patterns[18] have been performed by detecting refactorings and design patterns at release level. However, in the first case,[9] it would have been desirable to perform a more precise detection of refactorings at commit level, and in the second case,[18] to study how design patterns were introduced by developers in specific commits. Yet, the refactoring detector[8‡] and the design pattern detector[19] work on byte code. Similarly, a study on the evolution of test cases performed by Zaidman et al[20] could have been performed at fine-grained level aiming at observing specific activities triggering the need to augment test suites, as well as changes leading to better test suites achieving higher coverage. Of course, there are also cases (see the clone studies above) for which analysis at release level[16] can be easier and probably more meaningful.[17] All these considerations outline the need for the study presented in this paper: An empirical investigation into the phenomenon of broken snapshots is needed aiming at shedding some light into the reasons on *when* and *why* such changes happen.

Main results. The results of our study indicate that broken intervals (i.e., time periods including 1 or several consecutive broken snapshots) occur in 96% of the projects we analyzed with a median number of 4 in each system. Broken intervals are mostly small, with a median size of 3 subsequent broken snapshots. However, they can severely impact the compilability of software systems. As a matter of fact, on average, only 38% of the change history of a project system is successfully compilable today. The substantial number of broken snapshots shows their potential to hinder commit-based analysis of the project's change history when this requires compiling a snapshot of a system, (e.g., in the context of MSR-driven research). Moreover, broken snapshots are in most of the cases related to dependency resolution problems. Finally, snapshots' age highly impacts the likelihood of a successful compilation. Indeed, *Recent Snapshots*, (i.e., snapshots referring to commits performed recently in the change history) are almost 3.76 times more likely to compile with respect to *Early Snapshots*, (i.e., early snapshots in the change history of the project).

Paper structure. Section 2 describes the study definition, research questions, and planning. Results are reported and discussed in Section 3, while the threats to its validity are discussed in Section 4. Section 5 discusses the related literature on the analysis of build systems, while Section 6 concludes the paper and outlines directions for future work.

2 | EMPIRICAL STUDY DEFINITION AND DESIGN

The *goal* of the study is to analyze the change history of open source software projects, with the *purpose* of investigating the extent to which broken snapshots occur and which are the typical errors that a miner can observe when automatically compiling change history code snapshots. The *quality focus* of the study is on compiling projects' snapshots downloaded at commit-level granularity. The *perspective* is of researchers interested in performing mining studies over project histories using approaches and tools requiring to compile the project. The *context* consists of 100 Java open source projects using a Maven-based build infrastructure.

More specifically, the study aims at addressing the following **R**esearch **Q**uestions (RQs):

---

‡This tool is reported as an example, as refactoring detection tools working on source code also exist.

TABLE 1 A total of 100 projects considered in the study

No. of Classes	KLOC	No. of Commits	Mean history length	Min-max history length
4-6027	0-887	219 395	5.21	0-13

- **RQ$_1$:** *How many snapshots are currently compilable in the change history?* This research question aims at quantifying the relevance and magnitude of the observed phenomenon by investigating the frequency with which broken snapshots occur over time, and to what extent automatically compiling the complete change history of software project can be successful. We also investigate whether the size of the change history and the snapshot's age impact the likelihood of a successful compilation. We have chosen to study the effect of these 2 factors because it is possible that recent snapshots could be less affected by the problem, e.g., because dependencies are still available.

  The output of RQ$_1$ will provide a quantification of the possible problems researchers could experience in mining and compiling the change histories of open source systems.

- **RQ$_2$:** *Which types of errors can be observed by automatically compiling code snapshots?* The second research question sheds the light on the possible causes behind broken snapshots, by categorizing the compilation errors, we identified in our study. Knowing *why* a broken snapshot happens can help in understanding if it could be automatically/manually fixed with a reasonable effort while mining the software change history.

2.1 | Context selection

The *context* of our study consists of 100 open source projects from the Apache ecosystem.§ Our choice of focusing on these software projects is not random but aimed at considering systems that are (1) widely used in mining studies and, more generally, in studies carried out in the software engineering community,[21-24] (2) using state-of-the-art build systems, and (3) having a considerable change histories to analyze. Overall, to date the Apache ecosystem is composed of 249 software projects. Among those, we only selected the ones fully written in Java (187 projects) and relying on Maven¶ as an automated build tool. As we will explain later, the use of a fully automated build tool is a requirement for considering a system in our study, while the focus on systems written in Java is dictated by the fact that Maven is mainly used in the context of Java-based projects. As of today, Maven is one of the most popular build systems for Java projects. According to a survey, (performed in mid 2013)‖ Maven is being used by 71% of the interviewed developers, while 35% use Gradle** and 17% Ant.†† Note that the sum is beyond 100% as there are developers using multiple tools. In summary, we decided to focus on Maven not only because of its popularity (especially in the case of the Apache ecosystem[25]), but also because of its capability to resolve dependencies (e.g., a broken snapshot for an Ant-based project could just be due to the need for manually downloading a library and putting it in a specific location). Moreover, as it will be explained in Section 2.2, Maven generates error codes for different kinds of compilation errors, which allows us to address **RQ$_2$** without the risk of introducing a subjective or error-prone classification.

To identify which of the 187 Java projects make use of Maven, we cloned their *Git* repositories and verified the presence of `pom.xml` in the last systems' snapshot. This selection resulted in identifying 136 projects (72% of the total) using Maven. Because of the very computationally expensive analysis that we needed to perform to answer our 2 research questions, we decided to limit our study to 100 randomly selected projects. Table 1 reports the characteristics of the subject systems, and in particular (1) the size ranges in terms of the number of classes and kilo of lines of code (KLOC), (2) the overall number of commits analyzed, and (3) the average, minimum, and maximum length of the projects' history (in years) analyzed. All the considered projects are hosted in *Git* repositories. The list of 100 projects considered in our study can be found in our replication package (see Section 2.4).

In order to assess the extent to which the chosen set of projects represents a large body of open source Java projects (in terms of code size, team size, and project activity), we used the diversity metric proposed by Nagappan et al.[26] We matched the list of the systems in our study against the projects available in Boa,[27] and ended up with only 10 out of 100 projects that were matched by name. The diversity metric calculated on this subset was 0.4984, which means that only 10% of our dataset covers almost half of the open-source projects according to 4 dimensions: developers, project age, number of committers, and number of revisions. The scores for each dimensions are 0.89, 0.99, 0.92, and 0.91, respectively, indicating that our subset covers relevant dimensions for our analysis.

2.2 | Data extraction process

The first step needed to answer the 2 research questions is to identify, for each analyzed project P_i, the time interval T_b during which it used the Maven building tool. To identify T_b for each P_i, we mined P_i's complete change history looking for the commit C_{start} introducing the Maven build file `pom.xml`. Then, we verified that there was no commit in the P_i's history deleting the build file from the versioning system until the last commit considered in our study (we cloned the repositories during September 2014). Thus, T_b for each of the considered projects begins at the date in which C_{start} was performed and ends at the date in which the last commit was performed in the cloned repository. This process resulted in the exclusion, on average, of 0.79 years of change history across the 100 systems (i.e., $\approx$ 3 years). Overall, we analyzed 437 cumulative years of change history for all the projects.

Once the portion T_b of change history to analyze was defined for each project, the steps detailed in the following subsections have been performed to extract the data needed to answer our RQs. The overall data extraction process took 11 weeks despite being distributed across 3 Linux servers of which 2 were equipped with 8 10-core 3.2 GHz CPU (80 cores) and 192 Gb of RAM and 1 with 40 quad-core 2.13GHz CPU (160 cores) and 160 Gb of RAM.

§http://www.apache.org
¶http://maven.apache.org
‖http://arhipov.blogspot.it/2013/08/java-build-tools-survey-results.html
**https://gradle.org
††https://ant.apache.org

RQ$_1$ and RQ$_2$: Compiling Snapshots for each Commit. The basic data needed for answering the 2 research questions is the information about the compilability of each snapshot present in the change history (T_b) of each project P_i. To extract such information, we built a mining tool using Maven to automatically compile the system at each snapshot in T_b. During this phase, each snapshot S is classified as _compilable_ or _not compilable_.

If a snapshot S is _not compilable_, it is classified as a _broken snapshot_. We define a _broken interval_ as a series of continuous broken snapshots $S_i, S_{i+1}, \ldots, S_{j-1}, S_j$, such that S_{i-1} and S_{j+1} are either _compilable_ snapshots or not part of the change history (i.e, S_{i-1} and/or S_{j+1} do not exist). The snapshot S_i is the _broken interval starting point_ represented with BI_{start}, while the snapshot S_j is the _broken interval ending point_ represented with BI_{end}. We limited our analysis to the _master_ branch (including merges of other branches) for several reasons. First of all, according to the guidelines provided by _Git_,[‡‡] the master branch _always reflects a production-ready state_. Thus, it can be considered as the most stable one and, therefore, it is worth investigating the phenomenon of broken snapshots on such a branch. We also verified this conjecture applies to the analyzed projects by analyzing the documentation page of 10 randomly selected Apache projects. Secondly, most of the mining studies are conducted through the analysis of the master branch (e.g., some previous studies[19,20,28]). Because our goal is to analyze how the phenomenon of broken snapshots can impact the activities of researchers interested in compiling code snapshots, we decided to focus our attention only on this branch.

Also, note that for our perspective of researcher interested to perform fine-grained historical analysis of compilable commits, we are interested in the compilability of each single commit. This might not be the case when looking at the same phenomenon from a developer's perspective, for which what matters is the compilability before performing a push or pull request.

In order to compile all the 219 395 snapshots in our study, the tool that we built relies on Maven 3.2.3 (the latest version available as of September 2014, which ensures backward compatibility with previous versions), by programmatically invoking it through the Java API provided by Apache.[§§] More specifically, our tool creates an `Invoker` of type `InvocationRequest` setting `compile` as the building goal. The `InvocationResult` returned by the `Invoker` reports the compiling's results, including its success (i.e., the snapshot was compilable) or failure (i.e., the snapshot was not compilable).

RQ$_2$: Classifying Compile Errors. While the classification of snapshots as _compilable_ or _not compilable_ and the identification of _breaking intervals_ is sufficient to answer **RQ$_1$**, in the context of **RQ$_2$**, we need to analyze the issues observed when compiling _broken snapshots_. To this aim, we mapped the errors captured while compiling _not compilable_ snapshots into 4 possible categories related to different _building actions_ that may fail into 1 of these categories:

- _Resolution_: errors related to the resolution of artifacts (e.g., download of dependencies);

- _Parsing_: errors due to the parsing operation (e.g., malformed build files);
- _Compilation_: errors occurring during the compilation phase (e.g., syntactic errors in the source code);
- _Other_: generic errors that cannot be mapped to any of the above listed actions (e.g., Maven itself crashes during the run).

These 4 categories have been iteratively defined by 2 of the authors that manually analyzed the errors stored by our mining tool for each snapshot tagged as _not compilable_. The errors thrown by Maven during the snapshot compilation have been manually and independently analyzed by each of the 2 involved authors, who classified them into categories of errors related to the different compiling actions. Specifically, the 2 authors manually mapped the exceptions thrown by Maven into the corresponding _action_ categories. To accomplish this task, they read the Maven exceptions' documentation available online.[¶¶] After having independently mapped the errors into categories, the 2 authors met to resolve the conflicts and to refine their classification, leading to the 4 categories listed above.

2.3 | Data analysis

To answer **RQ$_1$**, we report descriptive statistics about the number of _broken intervals_ found while attempting to automatically compile the change history of the analyzed software systems and the length of such _broken intervals_. Also, we measure the length of _broken intervalsBI_ in terms of the number of _broken snapshots_ between BI_{start} and BI_{end} (considering BI_{start} and BI_{end} too).

Because we want to report absolute values as they are easier to be interpreted than percentages of commits over the total history length, we need to take into account the different size of the change histories of the subject systems. To this aim, we classified projects in short, medium and long history length, on the basis of the available, observed history. This also allows us to verify the impact of the history length on the likelihood of observing broken snapshots. To classify the projects into the above categories, we compute the first (Q_1) and the third (Q_3) quartile of the distribution representing the change history length of the subject systems. We classify systems into the following categories: (1) _short history_ length if they have a number of commits C_n lower than Q_1; (2) _medium history_ length if $Q_1 \leqslant C_n < Q_3$, and (3) _long history_ length for which $C_n \geqslant Q_3$. We report and discuss results on the number and size of broken intervals for the 3 categories separately as well as for all categories together. In order to provide an overall view of the extent to which _broken snapshots_ are found in the change history, we show the distribution of the _compilability_ (that is, the percentage of snapshots successfully compilable in the change history of each of the analyzed systems). This analysis complements the previous one performed with absolute numbers.

Moreover, we determine whether the number of broken intervals and project's compilability is correlated with the history's length. We show scatter plots, compute Kendall's rank correlation, and also check, using a Kruskal-Wallis test, whether the number of broken intervals differs across the 3 categories.

[‡‡]https://git-scm.com/doc
[§§]org.apache.maven

[¶¶]http://tinyurl.com/ja4ncal

TABLE 2 Number of broken intervals for projects having different length of change history

Projects	Min	1st Qu.	Median	Mean	3rd Qu.	Max	SD
Short history	1.00	2.00	3.00	3.12	4.00	7.00	1.56
Medium history	0.00	1.00	5.00	7.84	9.00	32.00	8.35
Long history	0.00	1.00	7.00	16.50	26.00	62.00	18.96
All	0.00	2.00	4.00	8.99	9.00	62.00	12.26

Qu, quartile.

Finally, we analyzed whether the *age* of the snapshot in a software project is an important factor characterizing its likelihood of successful compilability. We classified the snapshots of each project in accordance to their *position* in the change history. Similar to what previously was done for the change history length, we use quartiles to define the categories of snapshots. In particular, given a software project's change history $CH = \{S_1, S_2, \ldots, S_n\}$, we compute the first ($Q_1$) and third ($Q_3$) quartile of the distribution of the snapshots over time. We classify the snapshot of each project CH in the following categories:

- *Early Snapshots*: snapshots belonging to the early stage of the change history ($S_i | i < Q_1$). Representing the first 25% of the snapshots in the *CH*.
- *Intermediate Snapshots*: snapshots belonging to the middle stage of the change history ($S_i | Q_1 \leqslant i \leqslant Q_3$). Representing the middle 50% of the snapshots in the *CH*.
- *Recent Snapshots*: snapshots belonging to the recent stage of the change history ($S_i | i > Q_3$). Representing the last 25% of the snapshots in the *CH*.

We report and discuss the results on the distribution of compilability for such categories of snapshots. The intuition behind this analysis is that we expect that snapshots belonging to the early stage of a project's change history are less likely to be successfully compilable with respect to more recent snapshots, given the fact that references to dependencies might have been changed, missing or deprecated. To verify the statistical significance of such a hypothesis, also in this case, we apply the Kruskal-Wallis test.

For **RQ$_2$**, we report the results of classifying non-compilable snapshots according to the possible causes behind those errors (see Section 2.2).

2.4 | Replication package

The data set used in our study is publicly available at http://www.cs.wm.edu/semeru/data/breaking-changes/. Specifically, we provide: (1) the list of analyzed systems with links to their versioning system and GitHub page; (2) the raw data collected for the 2 research questions; and (3) the R scripts and working data sets used to run the statistical tests and produce the figures and tables presented.

3 | EMPIRICAL RESULTS

This section reports the analysis of the results aimed at answering 2 research questions formulated in Section 2.

3.1 | RQ$_1$: occurrence of broken snapshots

We divided the projects in 3 categories based on the length of the change history using the quartiles of such distribution as described in

Section 2.3. The quartile values are: $Q_1 = 235$ and $Q_3 = 1,968$ commits. Note that these quartiles refer to the portion of the change history in which we identified a build file. Because we are using Q_1 and Q_3 to define our sets, the number of projects for each category is 25, 50, and 25 respectively for projects in the *short, medium* and *long* history category.

Table 2 shows the number of *broken intervals* observed during the change history of the studied systems. We found that 96% of projects experienced at least one *broken interval* in their history. Also, the value for the first quartile (2.00) for all the projects indicates that at least 75% of the analyzed projects experienced at least 2 *broken intervals*. Also the median (4.00) and the mean (8.99) values confirm that, generally speaking, the phenomenon of broken snapshots cannot be ignored by researchers interested in mining (and compiling) the change history of a software system. When looking at the data for projects belonging to the 3 groups, on the one hand, we can notice that the median and mean number of broken intervals slightly increases in case of projects with medium or long history. On the other hand, the scatter plot in Figure 1A shows that there is no clear correlation between the size of the change history and the number of broken intervals found. This result is confirmed by the correlation value ($\tau = 0.29$), which indicates a low correlation between the phenomenons. Note that even though we obtained a low correlation value, this is definitely not negligible, therefore worth to further investigate using different dataset. Finally, a statistical comparison among the 3 groups performed using the Kruskal-Wallis test indicates a lack of statistically significant difference (p-value $= 0.06$).

To get a better idea of what these numbers mean in practical terms (i.e., to what extent these broken intervals affect the compilability of the subject systems), Table 3 shows the descriptive statistics summarizing the length of the broken intervals (i.e., the number of commits performed between a BI_{start} and its corresponding BI_{end} —see Section 2.3). The median length of broken intervals is 3.00, showing that at least half of the broken intervals only affect few commits (less than 4). However, relatively high values of the mean (110.50) and standard deviation (478.03) tell another part of the story, highlighting the presence of very long broken intervals. For instance, we identified in our dataset a very long breaking interval in the Apache Oozie system. The BI_{start} snapshot is dated September 2, 2011, and its corresponding BI_{end} is found on July 17, 2012, for a total of 408 broken snapshots. In order to understand the reason behind this long breaking interval, we manually analyzed what was the change that actually broke the compilation, finding that it was related to the update of a dependency, i.e., `org.slf4j`, that cannot be resolved. As for the end of the broken interval, we found that it corresponds to a commit in which a developer updated most of the dependencies of the project, likely resolving the `org.slf4j` dependency.
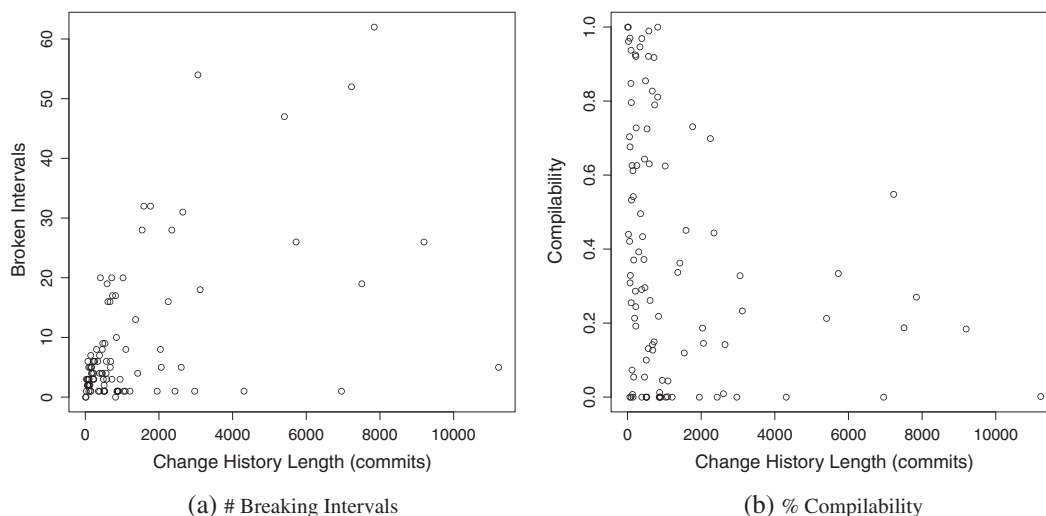
(a) # Breaking Intervals   (b) % Compilability

FIGURE 1 Scatter plots of (A) size of broken intervals per change history length; and (B) compilability rate per change history length

TABLE 3 Size of broken intervals

Projects	Min	1st Qu.	Median	Mean	3rd Qu.	Max	SD
Short history	1.00	1.00	5.00	18.96	16.00	139.00	33.20
Medium history	1.00	1.00	4.00	53.09	30.00	1212.00	150.39
Long history	1.00	1.00	2.00	184.00	26.00	6955.00	676.01
All	1.00	1.00	3.00	110.50	25.75	6955.00	478.03

TABLE 4 Percentage of compilable snapshots

Projects	Min	1st Qu.	Median	Mean	3rd Qu.	Max	SD
Short history	0.00	24.44	44.00	48.91	79.61	97.01	33.63
Medium history	0.00	5.44	29.60	39.46	72.50	100.00	34.69
Long history	0.00	0.00	18.54	23.78	33.26	100.00	28.38
All	0.00	5.43	29.60	38.13	66.00	100.00	33.79

The effect of broken intervals on compilability of the analyzed systems' snapshots is described in Table 4, which reports the percentage of compilable snapshots. The process adopted to verify the snapshots' compilability is the one described in Section 2.2. Surprisingly, the percentage of compilable snapshots is quite low (mean = 38%, median = 70%). This clearly poses strong limitations on the kind of analysis one can carry out on these non-compilable snapshots, for instance, the kinds of analyses outlined in the introduction. On 18 projects, we were not able to compile any of the snapshots available. We manually investigated such cases in order to understand the possible reasons behind this result. We found that all of them are actually sub-projects belonging to parent projects. Even if they have their own `pom.xml` file, their compilation depends on the compilation of the parent projects. As an example, `Log4j-receivers` is part of the `Log4j` project. In one of the first commits, a developer moved the build plugin configuration to the parent project,[||] making it impossible to compile any of the snaphots of the system. Note that we do not look at the relationships among the software projects because our goal is to systematically investigate the compilability of Apache projects, actually simulating the behavior of a researcher interested in performing a large scale mining study.

On the other hand, we also observed several exceptions to the general low compilability trend, like the Apache Quid-proton project that reported 100% compilability for all the snapshots despite the relatively long change history (1759 mined commits). Interestingly, during its change history, this project only established dependencies toward 2 libraries, i.e., `JUnit` and `org.mockito`. However, it is worth noting that developers always declared dependencies with major releases of such libraries.

Moreover, we investigated whether the length of the change history (negatively) correlates with the compilability of the project. The scatter plot in Figure 1B clearly shows a very low correlation, which is confirmed by the Kendall's $\tau = -0.25$.

Finally, we analyzed the impact of the snapshots' age on the compilability. Table 5 shows the distribution of compilability for the 3 categories of snapshots based on their age in the software project. The trend is clear—the age of the snapshots is an important factor characterizing the likelihood of a successfully compilation. Indeed, looking at the median values, *Recent Snapshots* are almost 2 times more likely

[||] http://tinyurl.com/hodp7fu

TABLE 5 Percentage of compilable snapshots based on snapshots' age

Snapshots' age	Min	1st Qu.	Median	Mean	3rd Qu.	Max	SD
Early Snapshots	0.00	0.00	12.39	36.20	83.93	100.00	41.06
Intermediate Snapshots	0.00	0.00	23.60	36.42	71.01	100.00	38.62
Recent Snapshots	0.00	0.46	46.56	43.89	75.69	100.00	38.51

to compile than *Intermediate Snapshots* and 3.76 times more likely to compile with respect to *Early Snapshots*. The result is statistically significant (p-value < 0.01) as highlighted by the Kruskal-Wallis test.

Summary for RQ$_1$. Among the analyzed systems, 96% exhibit at least 1 broken interval in their history, and this phenomenon is not necessarily more severe for projects with a longer change history. Most of the broken intervals only result in a few uncompilable commits, while the others persist in the system for a long time resulting in a high number of snapshots (median 30%) that can not be properly compiled or built. Snapshots' age in a software system represents an important factor characterizing the likelihood of a successful compilation, with recent snapshots being more compilable than early snapshots.

3.2 | RQ$_2$: likely causes for broken snapshots

Table 6 reports classification results of the causes behind the compiling errors that we identified in the subject projects (see Section 2.2). A large number of the errors (58%) raised during the *Resolution* of an artifact, usually representing a dependency to be solved (e.g., a library linked by the project under analysis). Indeed, the most frequently raised exceptions we found during the Maven building were `DependencyResolutionException` (32%) and `ArtifactResolutionException` (26%). The former is raised, for example, during the compilation of an Apache Oltu's snapshot S_B, because of the problem of downloading the OAuth library, responsible for the implementation of a secure authorization protocol via HTTP. This error has been fixed after 193 commits in a commit S_F by modifying the reference pointing to the OAuth library in the `pom.xml` file. It should be noted that the compilation error we identified during these 193 commits could be the result of 2 scenarios having different implications:

1. *The compilation error was actually present at the time T in which S_B was committed*. This means that the developers actually put a wrong reference to the OAuth library inside the build file. Thus, starting from *T*, it was not possible to compile the system for the subsequent 193 commits, when S_F fixed the issue. If this is the scenario behind S_B, this change clearly resulted in a long broken interval for Apache Oltu, with all possible consequent problems.
2. *The compilation error was not present when S_B was committed*. Suppose that the reference to the OAuth library present in the build file after S_B was valid at that time, and has remained so until S_F, in which the developers updated the OAuth's reference to a new one, given the unavailability of the previous reference. Clearly, by mining the OAuth's versioning system today, the old reference is no more available because the time in which S_F has been committed results in compiling issues during the 193 commits between S_B and S_F. In this scenario, the S_B commit did not cause any issues for developers during the broken interval we observed, but it only represents a problem for people interested in post-mortem analysis—compiling

TABLE 6 Likely causes behind broken snapshots

Action	%
Resolution	58
Parsing	10
Compilation	4
Other	28

snapshots of a system for each commit, which is a typical use case in the field of mining software repositories.

It is worth noting that it is not obvious to determine to what extent this phenomenon occurs, because we cannot determine whether the URL was valid when the pom.xml file was committed. However, it must be pointed out that on our interpretation this still constitutes a broken interval for researchers interested to analyze the past history of a project or use such data to build a recommender.

Parsing and *Compilation* errors account only for a small fraction of broken snapshots identified in the projects (10% and 4%, respectively), showing that the build and the source code files are generally correct from a syntactic point of view. One example of such broken interval is the one we observed in the Apache Jackrabbit system because of compilability problems in 1 of the libraries it exploited. Such a problem was fixed after 3 commits with a commit reporting the following message: "*Bundle plugin 2.3.5 requires Java 6. Downgrade to 2.3.4 that works with Java 5*". The problem was due to the different Java versions exploited by the client project (e.g., Apache Jackrabbit, requiring Java 5) and the specific version of the exploited library (e.g., Bundle plugin 2.3.5, requiring Java 6). The problem was solved by downgrading the dependency to a previous version of the library requiring Java 5. Unlike the example discussed for the Apache Oltu system, in this case, we can observe that the broken interval has been directly observed by the developers of the Apache Jackrabbit project.

The remaining 28% of errors fall in the *Other* category, containing errors that we were not able to map to a specific compile action.

Summary for RQ$_2$. The causes behind broken snapshots are often related to problems encountered during the resolution of dependencies (58% of cases), while we rarely observed errors because of syntactic errors in the build and source code files.

4 | THREATS TO VALIDITY

Threats to *construct validity* concern the relationship between theory and observation, and in this work, they are mainly related to the measurements we performed in our study. As discussed in Section 3.2, it could be the case that we observe missing dependencies because a library URL contained in a Maven pom.xml file is no longer valid. While

this can be considered as a false positive while interpreting the results from the perspective of somebody compiling the system at that time (e.g., a tester), as explained in Section 3.2, this is a true broken interval if one is interested to analyze the past history of a project, for various purposes. Imprecisions in our study can also be due, as explained, to the fact that we only observed the evolution history of a project by looking at the master branch. However, as explained in Section 2.2, this is intentional in order to identify a specific evolution timeline.

Threats to *internal validity* are related to factors, internal to our study, that can influence our results. The main threat to internal validity is related to cause-effect relationships we claim, especially in RQ_2, where we analyze the likely causes of broken snapshots. As explained earlier, we mainly rely on error messages raised by Maven, which may not necessarily capture the true underlying causes of a broken snapshot (this is why we refer to them as "likely causes"). Nevertheless, we must point out that our focus is to observe *what breaks the compilability today, rather than what broke compilability at development time*.

As explained in Section 2.2, we limited our analysis to the projects' history master branch, under the assumption that it is the main and stable development trunk. However, we could not exclude that, for some projects, the main development trunk occurs in different branches.

Threats to *conclusion validity* concern the relationship between experimentation and outcome. As explained in Section 2.3, wherever possible, we have used appropriate statistical procedures, including effect size measures, to address our research questions.

Threats to *external validity* concerns the generalization of our findings. Our results are clearly confined to a specific category of software projects, i.e., Java projects using Maven. Moreover, McIntosh et al[25] found that Maven adoption drops drastically in projects outside of the Apache foundation. We have explained in Section 2.1 why we decided to focus on such projects. Truly, nowadays other build tools such as Gradle[***] are becoming even more popular than Maven; however, our focus on Maven for a mining past history study like ours is motivated by its popularity in the past 5 years or so. Nevertheless, our study definitely needs to be extended on projects using other build tools and, above all, to projects developed in different programming languages, including C/C++ programs using Make. Last, but not least, although we have shown that the selected pool of projects achieve a good level of diversity,[26] future studies should look at projects deployed on different forges or belonging to different open source communities other than the Apache Software Foundation, because different communities may impose different promotion policies.

5 | RELATED WORK

Despite the significant research and practical importance of a problem of broken snapshots, there have been only few studies aimed at understanding this phenomena and its implications. While there is some work on studying continuous integration practices[3,29] and build systems,[30,31] there are no general recipes on how to handle broken snapshots.

Seo et al[2] presented an empirical study of 26.6 million builds from Google's cloud-based build system used internally for Google projects. The authors analyzed the log of the build system in order to understand (1) how often compiling actions fail; (2) why they fail; (3) how long it takes to fix them. Although our study appears to be similar to the one conducted by Seo et al, there is a fundamental difference between the contexts of these 2 studies. Indeed, while Seo et al investigated the *developers' building activity* (i.e., every time a developer attempts to build the project she is working on), our study analyzed the *repository's history* (i.e., the committed source code). Clearly, compiling activities do not match the activity performed on the repository. Indeed, *a developer could obtain several compile failures using the building tool, before performing a commit that does not contain any broken snapshot*. Also, while the reasons behind a compiling failure and a broken snapshot can be shared between these 2 studies, the results regarding the frequency of compile errors and how long does it takes to fix them cannot be compared because they are related to different contexts (building vs committing). Finally, it is worth noting that in our study, we perform our analysis relying on Maven as building tool, rather than using an internal Google's building system. This different choice allows us to directly analyze projects typically used in MSR studies. Thus, our study can be seen as strongly complementary to the one proposed by Seo et al.

The closest work to our is the study by Kerzazi et al,[1] who conducted an empirical study considering 3214 builds of an industrial system during a period of 6 months. They analyzed (i) the frequency of broken snapshots on one software project, (ii) under which circumstances builds usually breaks, and (iii) which are the factors impacting broken snapshots. Firstly, the results indicate that broken snapshots are frequent, i.e., they found a median of 8.3% of broken compilations per day. Secondly, they conducted semi-structured interviews involving 28 developers with at least 3 years of experience and authors of most of the breaking and fixing compiles they found. The outcome of the interviews showed that developers usually break the compilation when they forget to commit some files or update dependencies. This happens especially because they were active in several branches of a system at the same time. Finally, Kerzazi et al explored the factors that impact the rate of broken snapshots, demonstrating that the development timeline is the most important factor causing the breaks. While our study shares similar goals with the one by Kerzazi et al,[1] the context of our study is much more extensive as we conduct the study on 219 395 snapshots of 100 Java open source projects. In addition to quantitatively studying frequencies of broken snapshots, the goal of our study is to obtain and in-depth and systematic understanding of the phenomena into *how* and *why* broken snapshots happen. Moreover, it is important to remark that our study is performed at the *commit level*, which has important implications for the practical tools and research analyses that need to be performed at commit level granularity.[22,32–40] At the same time, our paper reinforces some of the findings provided by Kerzazi et al, in terms of broken snapshots' diffusion and time windows in which breaks are located.

McIntosh et al[4] studied, on the one hand, how frequently source code changes require changes to the build file, and, on the other hand, the proportion of developers responsible for build maintenance. The study involved 10 open source systems and showed that a good percentage of source code items (ranging between 4% and 27%) requires

---

[***]https://gradle.org

an accompanying build change. Moreover, about 79% of source code developers and 89% of test code developers are involved in build maintenance activities.

Cataldo and Herbsleb[29] studied major factors behind integration failures in distributed software projects, whereas Hassan and Zhang[41] defined a model for predicting the outcome of the compilation process. Unlike these works, in this paper, we do not provide methods that a researcher can use for assessing the output of the compilation process, but we have the main goal of systematically investigating to what extent the phenomenon of broken snapshots can impact the ability to conduct a mining software repository study.

Phillips et al[42] designed and executed the study aimed at studying how release engineers and managers make integration decisions. The work by Kwan et al[43] investigated the effect of socio-technical congruence (i.e., the alignment between work and social relationships between team members) on the outcome of the compilation process on a large industrial software project. Their findings suggest that congruence affects build success probabilities for different types of builds. Our work does not take into account social factors, but it is focused on the analysis of the likely causes behind broken snapshots.

Broken snapshots have been studied from a completely different perspective by Raemaekers et al.[5] Specifically, they investigated 7 years of library releases on the Maven central repositories, to understand the phenomenon of incompatibility caused by new versions of such libraries. They found that the "semantic versioning", i.e., the strict usage of major and minor release numbers do not provide developers with enough information about the compatibility of the library interfaces. Differently from our study, they looked at a very specific problem more related to the evolution of library interfaces, while we are interested in investigating various kinds of compilability problems, from dependencies to parsing errors.

6 | CONCLUSION

This paper presents a study aimed at investigating compile broken snapshots in 100 Java projects from the Apache Software Foundation, all of them relying on Maven as an underlying build tool. More specifically, the study aims to investigate (1) the magnitude of the phenomenon, i.e., frequency of broken intervals and their length in terms of the number of broken snapshots, (2) the likely causes of broken snapshots, as identified from the Maven's error messages.

In summary, the main results of the study are that the size of broken intervals is relatively short, with a median of 3 commits, although the long-tailed distribution highlights some cases in which the broken intervals can be very long.

Looking at the percentage of snapshots that could not be compiled (median 30%), we can easily understand that compilability breaks can seriously affect any kind of research activity studying the evolution of software projects and needing compiled snapshots. Also, similar problems arise for recommender systems requiring code to be compiled and/or executed and at the same time exploiting projects' past history.

The causes behind the majority of the compile broken snapshots can be mostly attributed to the problems with dependencies. While Maven is considered to be one of the most advanced build tools in terms of han-

dling dependencies, the problems still arise, especially when observing the past change history of a project, e.g., libraries are no longer available or URLs are outdated. Again, while this may not be a problem during actual development, this phenomenon can seriously affect any kind of post-mortem analysis either in research or practice.

Results of our study clearly highlight—at least within the investigated set of projects, language (Java), and build tool (Maven)—the major challenges of performing mining studies at commit level should such studies require source code to be compiled. One of the main reasons for that is to be found in dependency resolution problems. In some cases, dependencies are resolved through resources available online whose URLs are outdated at present time. For this reason, a possible way to circumvent this problem would be to resolve the dependency with an updated link to the missing resource. Therefore, researchers interested in performing software evolution analysis could tweak such references in order to successfully compile the project under analysis.

Finally, we found that the snapshots' age highly impacts the likelihood of a successful compilation. Indeed, *Recent Snapshots* are almost twice more likely to compile than *Intermediate Snapshots* and 3.76 times more likely to compile with respect to *Early Snapshots*.

In summary, if we observe one project's past change history today, it is likely that many of the snapshots at the commit level are likely to be incompilable. If compilation is required for our analyses, then we would have to seek the first compilable snapshot. Clearly, this could make our studies less precise because we are no longer observing all the changes (although, fortunately, systems are often not compilable just for few consecutive snapshots). Alternatively, compile broken snapshots (and outdated/missing dependencies in particular) have to be manually fixed, which makes the analysis effort-prone.

Researchers could maximize the number of snapshots successfully analyzed while minimizing the operational costs by focusing only on the recent snapshots of each system under analysis (if the desired analysis allows to discard part of the change history).

In our future work, we are planning on enlarging the study, especially with the aim of considering the impact of different types of build tools and, consequently, the projects developed in different programming languages (or even mixes of programming languages), e.g., C/C++ projects using Make as a build tool. Moreover, it is interesting to analyze whether the phenomenon of broken snapshots is different when analyzing branches different from the master branch. In addition, we are also planning on empirically assessing the impact of broken snapshots on reliability of observations when studying past change history of software projects. Last, but not least, we will investigate how the lack of a detailed, compilable history analysis can affect the accuracy of some previously conducted mining studies.

REFERENCES

1. Kerzazi N, Khomh F, Adams B. Why do automated builds break? An empirical study. *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, Victoria, BC, Canada; September 2014:41–50.

2. Seo H, Sadowski C, Elbaum S, Aftandilian E, Bowdidge R. Programmers' build errors: a case study (at Google). *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, Hyderabad, India; 2014:724–734.

3. Wolf T, Schroter A, Damian D, Nguyen T. Predicting build failures using social network analysis on developer communication. *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09. Washington, DC, USA: IEEE Computer Society; 2009:1–11.

4. McIntosh S, Adams B, Nguyen THD, Kamei Y, Hassan AE. An empirical study of build maintenance effort. *Software Engineering (ICSE), 2011 33rd International Conference on*, Waikiki, Honolulu, HI, USA; May 2011:141–150.

5. Raemaekers S, van Deursen A, Visser J. Semantic versioning versus breaking changes: a study of the maven repository. *14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28-29, 2014*, Washington, DC, USA; 2014:215–224.

6. Tsantalis N, Chatzigeorgiou A. Identification of move method refactoring opportunities. *IEEE Trans Software Eng.* 2009;35(3):347–367.

7. Fokaefs M, Tsantalis N, Stroulia E, Chatzigeorgiou A. Jdeodorant: identification and application of extract class refactorings. *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, may 21-28, 2011*, Waikiki, Honolulu, HI, USA; 2011:1037–1039.

8. Prete K, Rachatasumrit N, Sudan N, Kim M. Template-based reconstruction of complex refactorings. *26th IEEE International Conference on Software Maintenance (ICSM 2010)*, Timişoara, Romania; December 18, 2010:1–10.

9. Bavota G, Carluccio BernardinoDe, De Lucia A, Di Penta M, Oliveto R, Strollo O. When does a refactoring induce bugs? An empirical study. *12th IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM 2012, Riva del Garda, Italy; September 23-24, 2012:104–113.

10. Buse RPL, Zimmermann T. Information needs for software development analytics. *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12. Piscataway, NJ, USA: IEEE Press; 2012:987–996.

11. Ferenc R, Beszedes A, Gyimothy T. Fact extraction and code auditing with columbus and sourceaudit. *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, Chicago Illinois, USA; September 2004:513–.

12. Bakota T, Beszédes Á, Ferenć R, Gyimóthy T. Continuous software quality supervision using sourceinventory and columbus. *Companion of the 30th international conference on software engineering*, ICSE Companion '08. New York, NY, USA: ACM; 2008:931–932.

13. Ferenc R, Lango L, Siket I, Gyimothy T, Bakota T. Source meter sonar qube plug-in. *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, Victoria, British Columbia, Canada; September 2014:77–82.

14. Spinellis D. Tool writing: a forgotten art? *IEEE Software.* 2005;22(4):9–11.

15. Kim M, Sazawal V, Notkin D, Murphy G. An empirical study of code clone genealogies. *SIGSOFT Softw Eng Notes.* 2005;30(5):187–196.

16. Saha RK, Asaduzzaman M, Zibran MF, Roy CK, Schneider KA. Evaluating code clone genealogies at release level: an empirical study. *Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on*, Timişoara, Romania; September 2010:87–96.

17. Bettenburg N, Shang W, Ibrahim WM, Adams B, Zou Y, Hassan AE. An empirical study on inconsistent changes to code clones at the release level. *Sci Comput Program.* 2012;77(6):760–776.

18. Aversano L, Canfora G, Cerulo L, Del G Concettina, Di P Massimiliano. An empirical study on the evolution of design patterns. *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, September 3-7, 2007*, Dubrovnik, Croatia; 2007:385–394.

19. Tsantalis N, Chatzigeorgiou A, Stephanides G, Halkidis ST. Design pattern detection using similarity scoring. *IEEE Trans Software Eng.* 2006;32(11):896–909.

20. Zaidman A, Rompaey BV, Demeyer S, van Deursen A. Mining software repositories to study co-evolution of production & test code. *First International Conference on Software Testing, Verification, and Validation, ICST 2008, April 9-11, 2008*, Lillehammer, Norway; 2008:220–229.

21. Franke N, von Hippel E. Satisfying heterogeneous user needs via innovation toolkits: the case of apache security software. *Res Policy.* 2003;32(7):1199–1215. Open Source Software Development.

22. Mockus A, Fielding RT, Herbsleb JD. Two case studies of open source software development: Apache and mozilla. *ACM Trans Softw Eng Methodol.* July 2002;11(3):309–346.

23. Bavota G, Canfora G, Di Penta M, Oliveto R, Panichella S. How the apache community upgrades dependencies: an evolutionary study. *Empirical Software Eng.* 2014:1–43.

24. Rigby PC, German DM, Storey M-A. Open source software peer review practices: a case study of the apache server. *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, Leipzig, Germany; 2008:541–550.

25. McIntosh S, Nagappan M, Adams B, Mockus A, Hassan AE. A large-scale empirical study of the relationship between build technology and build maintenance. *Empirical Software Eng.* 2015;20(6): 1587–1633.

26. Nagappan M, Zimmermann T, Bird C. Diversity in software engineering research. *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, Saint Petersburg, Russia; 2013:466–476.

27. Dyer R, Nguyen HA, Rajan H, Nguyen TN. Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. *Proceedings of the 2013 international conference on software engineering*, ICSE '13. Piscataway, NJ, USA: IEEE Press; 2013:422–431.

28. Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D. Detecting bad smells in source code using change history information. *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, Palo Alto, California, USA ; November 2013:268–278.

29. Cataldo M, Herbsleb JD. Factors leading to integration failures in global feature-oriented development: an empirical analysis. *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11. New York, NY, USA: ACM; 2011:161–170.

30. Phillips S, Zimmermann T, Bird C. Understanding and improving software build teams. *Proceedings of the 36th international conference on software engineering*, ICSE 2014. New York, NY, USA: ACM; 2014:735–744.

31. McIntosh S, Poehlmann M, Juergens E, Mockus A, Adams B, Hassan AE, Haupt B, Wagner C. Collecting and leveraging a benchmark of build system clones to aid in quality assessments. *Companion proceedings of the 36th international conference on software engineering*, ICSE Companion 2014. New York, NY, USA: ACM; 2014:145–154.

32. Śliwerski J, Zimmermann T, Zeller A. When do changes induce fixes? *SIGSOFT Softw Eng Notes.* May 2005;30(4):1–5.

33. Zimmermann T, Kim S, Zeller A, Whitehead EJ, Jr. Mining version archives for co-changed lines. *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06. New York, NY, USA: ACM; 2006:72–75.

34. Zimmermann T, Weisgerber P, Diehl S, Zeller A. Mining version histories to guide software changes. *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04. Washington, DC, USA: IEEE Computer Society; 2004:563–572.

35. Zimmermann T, Diehl S, Zeller A. How history justifies system architecture (or not). *Proceedings of the 6th International Workshop on Principles of Software Evolution*, IWPSE '03. Washington, DC, USA: IEEE Computer Society; 2003:73–.

36. Sheoran J, Blincoe K, Kalliamvakou E, Damian D, Ell J. Understanding "watchers" on github. *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014. New York, NY, USA: ACM; 2014:336–339.

37. Qiu D, Li B, Su Z. An empirical analysis of the co-evolution of schema and code in database applications. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013. New York, NY, USA: ACM; 2013:125–135.

38. Wu R, Zhang H, Kim S, Cheung S-C. Relink: recovering links between bugs and changes. *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11*. New York, NY, USA: ACM; 2011:15–25.

39. Kim M, Cai D, Kim S. An empirical investigation into the role of api-level refactorings during software evolution. *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11. New York, NY, USA: ACM; 2011:151–160.

40. Sunghun K, Kai P, Whitehead EJ, Jr. Micro pattern evolution. *In Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR '06)*, ACM, New York, NY, USA; 2006:40–46.

41. Hassan A, Zhang K. Using decision trees to predict the certification result of a build. *Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on*, Tokyo, Japan; September 2006: 189–198.

42. Phillips S, Ruhe G, Sillito J. Information needs for integration decisions in the release process of large-scale parallel development. *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12. New York, NY, USA: ACM; 2012:1371–1380.

43. Kwan I, Schroter A, Damian D. Does socio-technical congruence have an effect on software build success? a study of coordination in a software project. *IEEE Trans Software Eng*. 2011;37(3):307–324.

How to cite this article: Tufano M, Palomba F, Bavota G, et al. There and back again: Can you compile that snapshot? *J Softw Evol Proc*. 2017;29:e1838. https://doi.org/10.1002/smr.1838