Empirical Software Engineering (2021) 26: 32
https://doi.org/10.1007/s10664-020-09926-4

The nature of build changes

An empirical study of Maven-based build systems

Christian Macho[1] · Stefanie Beyer[1] · Shane McIntosh[2] · Martin Pinzger[1]

Accepted: 17 December 2020 / Published online: 16 March 2021

Abstract
Build systems are an essential part of modern software projects. As software projects change continuously, it is crucial to understand how the build system changes because neglecting its maintenance can, at best, lead to expensive build breakage, or at worst, introduce user-reported defects due to incorrectly compiled, linked, packaged, or deployed official releases. Recent studies have investigated the (co-)evolution of build configurations and reasons for build breakage; however, the prior analysis focused on a coarse-grained outcome (*i.e.*, either build changing or not). In this paper, we present BUILDDIFF, an approach to extract detailed build changes from MAVEN build files and classify them into 143 change types. In a manual evaluation of 400 build-changing commits, we show that BUILDDIFF can extract and classify build changes with average precision, recall, and f1-scores of 0.97, 0.98, and 0.97, respectively. We then present two studies using the build changes extracted from 144 open source Java projects to study the frequency and time of build changes. The results show that the top-10 most frequent change types account for 51% of the build changes. Among them, changes to version numbers and changes to dependencies of the projects occur most frequently. We also observe frequently co-occurring changes, such as changes to the source code management definitions, and corresponding changes to the dependency management system and the dependency declaration. Furthermore, our results show that build changes frequently occur around release days. In particular, critical changes, such as updates to plugin configuration parts and dependency insertions, are performed before a release day. The contributions of this paper lay in the foundation for future research, such as for analyzing the (co-)evolution of build files with other artifacts, improving effort estimation approaches by incorporating necessary modifications to the build system specification, or automatic repair approaches for configuration code. Furthermore, our detailed change information enables improvements of refactoring approaches for build configurations and improvements of prediction models to identify error-prone build files.

Keywords Maintenance · Build systems · Software quality

Communicated by: Kelly Blincoe

✉ Christian Macho
christian.macho@aau.at

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

1 Introduction

Large software projects use build systems, such as MAVEN, GRADLE, or ANT, to automate the compilation, testing, packaging, and deployment processes of their software products. The configuration of such build systems can often be complex (McIntosh et al. 2012), which also complicates their maintenance. Seo et al. (2014) showed that up to 37% of the builds at Google fail, stating neglected build maintenance as the most frequent cause. The development team is then blocked and obliged to fix the build first. Kerzazi et al. (Kerzazi et al. 2014) found a similar ratio of up to 18% of build breakage and estimated the total costs of breakages in their study context to be more than 336 person-hours.

As a software system evolves, changes are applied to the source code. Furthermore, development teams also need to maintain the build specification and hence, subsequent changes need to be applied to the build configuration. Adams et al. (2007a) and McIntosh et al. (2012) found evidence of a co-evolutionary relationship between source and build code. Hence, omitting changes to the build configuration that are needed to remain synchronized with the source code, can lead to build breakage. To that end, it is important to know when build changes should be applied. In our previous work (Macho et al. 2016; McIntosh et al. 2014), we proposed machine learning models that can predict whether source code changes should have accompanying build changes. However, these models lack the detailed information about the type of build change that is needed to understand and resolve the issue.

Recent studies have shown that the build specification is changed frequently (Macho et al. 2016; McIntosh et al. 2011). As the build system plays a crucial rule in the software development lifecycle, build system maintenance is also an important area to study. Although research showed an increasing interest on studying build systems and their configurations in the recent years, little is still known about the maintenance (Shridhar et al. 2014) of such systems and configurations. Moreover, a deeper knowledge about the kind of changes is important to understand the evolution and maintenance because such changes differ in their complexity and priority (Shridhar et al. 2014). Many studies that investigated the changes to source code files have found that, with a deeper knowledge about the changes, research can better understand these changes. Using this understanding, we can better tackle various problems that ultimately aim at decreasing the number of failing builds. Contrary to source code changes, changes to the build specification have not gained the attention they deserve.

In this study, we address this gap and investigate changes to the build configuration in detail. We are interested in which types of changes are typically made to the build configuration and when they are performed. Prior studies (McIntosh et al. 2014) consider the build configuration to be changed if the build file changes but do not investigate the detailed type of the change, *e.g.* inserting a new dependency to a third party library. A more detailed view on build configuration changes can improve studies of the build system and its configuration. Thus, we introduce BUILDDIFF, an approach to extract detailed build changes from MAVEN build files. Our approach is inspired by ChangeDistiller (Fluri et al. 2007; Gall et al. 2009), which extracts source code changes from Java source files. We also propose a taxonomy of build changes consisting of 143 build change types and five categories that our approach can extract. We evaluate BUILDDIFF in a manual investigation of 400 randomly selected build changing commits and find that it yields an average precision, recall, and f1-score of 0.97, 0.98, and 0.97, respectively.

Armed with an approach to extract build changes from MAVEN build files, we extract build changes from 144 open source Java projects from different vendors, of different sizes, and with different purposes. We study the extracted data in two ways. First, we study the frequency of build changes. We explore which change types are the most frequent ones and which change types are rarely applied. We analyze the frequencies also in terms of change categories. Furthermore, we study the co-occurrence of build changes to explore whether particular change types are frequently co-occurring with other change types. With these studies, we answer the first research question:

(RQ1) Which build changes occur the most frequently?
We divide this research question into two parts:

RQ1.1 *What are the most frequently occurring build change types?*
We find that changes to version numbers and dependency declarations are more frequent than other changes. In particular, the build change types PROJECT_POSTFIX_VERSION_UPDATE, DEPENDENCY_INSERT, and DEPENDENCY_INSERT are among the top-10 most frequent change types. Concerning build change categories, we found that the most frequently occurring change category is `General Changes`, followed by `Dependency Changes`, and `Build Changes`. Furthermore, the top-10 most frequent build changes account for 51% of the build changes.

RQ1.2 *What are the most frequently occurring build change patterns?*
We find that the three parts of the source code management system are usually changed together. Version numbers of the parent project and the respective project itself are normally changed together and the change type tends to be the same (*e.g.* a major version increase). Moreover, the declaration of dependencies tends to co-change with the dependency management part of the build specification.

By answering RQ1, we know which changes are performed to the build specification file. However, we do not know when these changes are performed. Knowing when particular changes are performed can be crucial because change types might be perceived as critical in certain periods. Hence, we also study the time at which build changes have been recorded and investigate how build changes are distributed over a project (*e.g.* consistently or bursty). More specifically, we answer the second research question:

(RQ2) When are build changes recorded?
We hypothesize that build changes are not equally distributed over time, but instead tend to cluster around important project events (*e.g.* official releases).

RQ2.1 *How are build changes distributed over time?*
Build changes are not equally distributed over the projects' timeline. There are particular phases that contain significantly more build changes than others. We observe that especially around releases, the frequency of build changes is high.

RQ2.2 *Which build changes happen before, during, and after releases?*
In particular, the change categories `General Changes` and `Dependency Changes` occur more frequently around release days than other categories. Furthermore, we find that critical changes, such as plugin configuration updates, are performed shortly before or on the release day.

The results of this study enable researchers to study build systems in much more detail. Our approach to extract build changes from Maven that we present in this paper can help to study and understand build configurations and their evolution. For example, our original work (Macho et al. 2017) helped us to identify common strategies to automate the build repair process (Macho et al. 2018). Our work also supports developers, for example, when integrated in the code reviewing process. First, BUILDDIFF can help to visualize build changes more specific because it can extract only actual changes without considering moved items or white spaces. Second, the assignment of code reviewers can be improved with our findings, for example by adding an additional reviewer from a build or release engineering department if late or critical changes to the build configuration are performed closely to an upcoming release. Furthermore, with our results, developers can profit prospective tooling that can, for example, detect missing build co-changes, based on the frequent build change patterns that we observed in this work.

This paper is an extension of our earlier work (Macho et al. 2017) and provides the following new contributions:

– An improved data preparation process that includes a larger sample of projects to increase the generalizability of our findings (Section 3)
– An improved version of BUILDDIFF and its taxonomy that allows a more detailed extraction of changes to version declarations (Section 4)
– An improved evaluation that demonstrates the applicability of the BUILDDIFF approach under a broader range of usage scenarios (Section 5)
– An extended investigation of the frequency of build changes replicating our original study (Section 6.1) with 114 additional projects.
– An additional analysis of build change patterns (Section 6.2)
– An extended investigation of when build changes are recorded replicating our original study (Section 7.1) with 114 additional projects.
– An additional study that investigates the occurrence of build changes on a fine-grained build change level (Section 7.2)

The remainder of the paper is organized as follows: Section 2 situates the paper with respect to the related work. Section 3 elaborates on the data preparation process. Section 4 presents our BUILDDIFF approach and describes the improvements compared to the original work. Section 5 evaluates the performance of BUILDDIFF, and discusses its strengths and weaknesses. Section 6 presents the study on the frequency of build change types and patterns. Section 7 shows the study on when build changes occur. Section 8 discusses the implications of our results and threats to validity. Section 9 concludes the paper and discusses promising avenues for future work.

2 Related work

In this section, we situate the paper with respect to related work. We first discuss studies that are related to build maintenance and we then elaborate on related research that deals with extracting changes from software artifacts.

2.1 Build maintenance

Related work on build maintenance includes the co-evolution of build systems with other artifacts of the development process. For instance, Adams et al. introduced MAKAO

(Adams et al. 2007b), a framework for re(verse)-engineering build systems. They studied the co-evolution of the Linux build system (Adams et al. 2007a) using MAKAO and found that the build system itself evolves and its complexity grows over time. Furthermore, they identified maintenance as the main factor for evolution. Tamrawi et al. (2012) proposed SYMake, a tool that creates a symbolic dependency graph that represents dependencies among files. They use this graph to detect smells and errors in makefiles, and to support refactoring of makefiles. Moreover, they showed that developers understood the makefiles better and to detect more smells in makefiles when using SYMake. McIntosh et al. investigated the evolution of the ANT build system (McIntosh et al. 2010) from a static and a dynamic perspective. They defined a metric for measuring the complexity of build systems and found that the complexity of ANT build files evolves over time, too. In follow-up work, McIntosh et al. investigated Java build systems and their co-evolution with production and test code (McIntosh et al. 2012). The results of a large-scale study showed a relationship between build technology and maintenance effort (McIntosh et al. 2015). The studies dealing with co-evolution were performed using coarse-grained measures. However, with our approach, these studies can go into more detail and relate co-evolution of particular change types, such as source code changes with build changes, similar to the work of Marsavina et al. (2014). In addition to these studies, Hardt and Munson developed Formiga, a tool to refactor ANT build scripts (Hardt and Munson 2013; 2015). With our work, we provide the basis to offer similar approaches to refactor build specifications based on our empirical findings.

Concerning the co-evolution of build configurations with other software artifacts, existing studies investigated models to predict build co-changes based on various metrics. For instance, McIntosh et al. (2014) used code change characteristics to predict build co-changes within a software project. Xia et al. (2015) extended this study by building a model for predicting build co-changes across software projects. In our previous work (Macho et al. 2016), we showed that we can improve both studies by using fine-grained source code changes. Furthermore, Xia et al. (2014) investigated missing dependencies in build files using link prediction. They showed that their algorithm outperforms state-of-the-art link prediction algorithms for this problem.

Other studies about build maintenance deal with, for example, understanding the build process to improve it. Lebeuf et al. (2018) conducted a design study investigating cognitive challenges when using build systems. They also propose BuildExplorer, a tool that helps understanding, optimizing, and debugging distributed build sessions. In a study at Microsoft, the tool was shown to help during these tasks. Wen et al. (2018) developed BLIMP Tracer, a build impact analysis system that identifies critical code for review. BLIMP Tracer helped to improve the speed and the accuracy of deliverables that are impacted by changes. In their recent work (Wang et al. 2018), Wang et al. investigate dependency conflicts and present DECCA, a tool that can identify critical dependency conflicts. Using DECCA, they contributed 20 new issues of dependency conflicts to the studied projects. Vassallo et al. (2018) present BART, an approach to suggest repair actions for broken builds. Using BART, developers felt that they could understand the build breakage better and that they can repair it faster. Going one step further, we provided an approach that can automatically repair dependency-related build breakage (Macho et al. 2018). This approach is already using the original version of BUILDDIFF and will profit from the improvements of this work. Especially these four works (Macho et al. 2018; Vassallo et al. 2018; Wang et al. 2018; Wang et al. 2019), show the emerging need of understanding build specifications in much more detail as the state-of-the-art build maintenance research can provide. With our work, we can provide a tool that is capable of contributing to study build changes with the aim of understanding why builds break, and how they are repaired. Ren et al. (2018) propose RepLoc, an approach to identify files that lead to an unreproducible build. With this approach, they can drastically reduce the scope of problematic files in terms of build reproducibility. Bezemer et al. (2017) studied four open source projects and found that unspecified dependencies are common. They identified six common causes of unspecified dependencies.

2.2 Change extraction

Many previous studies used changes that were extracted from different versions of source files to investigate various aspects of the evolution of software systems. Miller et al. (1985) and Myers et al. (1986) performed their studies by counting the number of added or deleted lines of text. One advantage of these approaches is that they avoid the complexities of parsing files to output the differences between versions of source files. However, one important shortcoming is that these approaches have difficulty mapping the changed lines of text to their syntactic or semantic meaning, such as the change of the return type of a method or the addition of an else branch. We address this issue in this work by recognizing the semantic of the changes which allows further work to study build changes in more detail.

Modern approaches overcome this issue by performing the differencing on the level of Abstract Syntax Trees (ASTs). For instance, Hashimoto and Mori (2008) developed Diff/TS, which operates on the raw AST created by parsing two versions of a source file. An example of this approach is ChangeDistiller (Fluri et al. 2007) which extracts differences from two consecutive versions of a Java file and maps the differences to 48 change types (Fluri and Gall 2006). Falleri et al. (2014) improved upon ChangeDistiller by combining approaches that match equal subtrees. These works significantly improved the possibility to study changes of source code which can be seen by the high number of other works that used this approach for the studies. This is a main motivation for our work because there is no approach that extracts build changes on this level of granularity. However, other studies (Shridhar et al. 2014) also describe this level of granularity as needed to better study build specifications.

One approach towards refining changes to build specification files by Désarmeaux et al. (2016) mapped line-level changes to MAVEN lifecycle phases and investigated the maintenance effort of each phase. They found out that the compile phase accounts for most of the maintenance. Again with our work, we give the possibility to study this in more detail as we can extract build changes in more detail.

The closest work to ours is the work of Shridhar et al. (2014). In this paper, the authors qualitative investigate build changes of 18 open source Java projects. They categorize these changes into six categories, *i.e.* adaptive, corrective, perfective, preventive, new functionality, and reflective, with the aim to measure churn in more detail. They found that corrective, adaptive, and new functionality, are the most frequently performed changes in the studied projects and consequently, produce the most churn. Furthermore, they found that these change types are also more invasive. Other changes are performed during development. They conclude their work with the insight that further research needs to incorporate detailed information about the type of build changes. At this point, our work comes into play. We address this research gap by providing an approach that refines build changes and offers research the needed detailed view on the changes of build specifications.

In summary, we find that several approaches exist to study build maintenance, build systems, and their configuration. However, these studies are primarily based on coarse-grained metrics and coarse-grained interpretations of changes. Previous work showed that

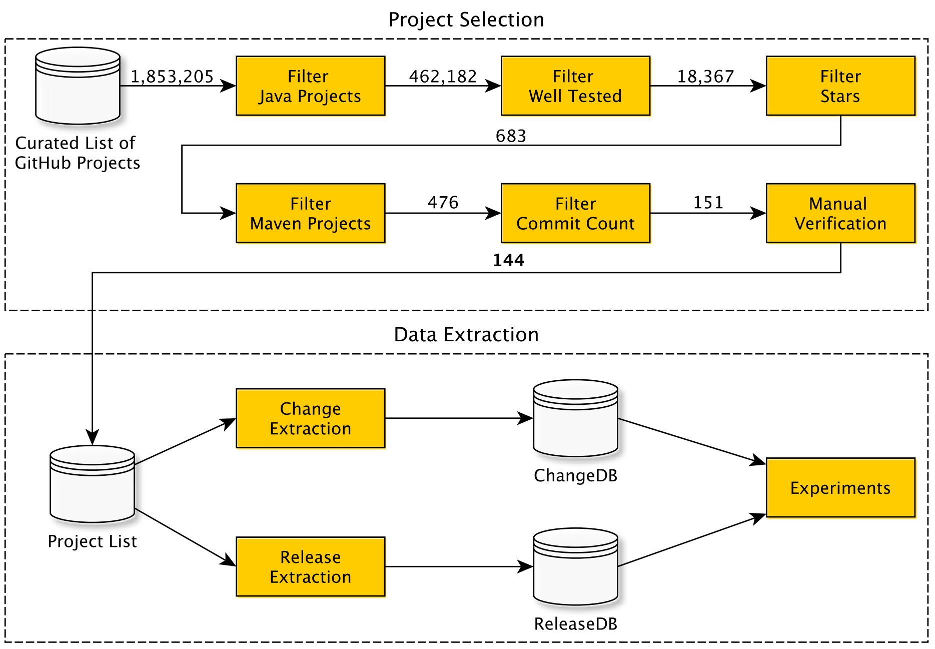


Fig. 1 Overall data preparation process. The values above arrows represent the number of projects

for programming languages, such as Java, the usage of a finer granularity of changes can help to improve several studies, *e.g.* works on prediction models (Giger et al. 2012; Giger et al. 2012; Giger et al. 2011; Romano and Pinzger 2011) or support the understanding of (co-)evolution (Fluri et al. 2009). To that end, it is important to also investigate build changes on a fine-grained level. Furthermore, in their work, Shridhar et al. (2014) explicitly name the need of having detailed build changes. In this paper, we address this gap by presenting BUILDDIFF, our approach to extract fine-grained changes from Maven configuration files.

3 Data preparation

In this section, we describe the relevant data and our process for retrieving it. The overall process is depicted in Fig. 1. The numbers on the arrows in Fig. 1 represent the incoming and the outcoming number of projects of each filter, respectively.

We start our data retrieval process with the curated data set of projects that is presented by Munaiah et al. (2017). We use this data set because it represents a list of engineered (non-trivial) projects. The data set has been created using a tool called *reaper* to measure seven dimensions, which capture the degree of engineering of projects. We use the data set[1] containing 1,853,205 GitHub projects as the starting point for our data retrieval process. In the following, we describe each step of the filtering process.

Java projects The tools that we use in this study (*e.g.* BUILDDIFF) are designed for Java projects. To address this, we only keep the projects that use Java as their main language.

[1]https://reporeapers.github.io/static/downloads/dataset.csv.gz

We consider a project as Java project if the *language* attribute of a project in the data set is “Java”. Applying this filter, we keep 462,182 projects.

Unit testing One investigated engineering dimension of the data set reflects the amount of unit testing that a project contains. This metric is calculated as ratio between the source code lines in test files and total source code lines. The higher the ratio the more the project is tested. We keep all projects that show a larger ratio than 0.5. Applying this filter, we keep 18,367 projects.

Stars As stated in Munaiah et al. (2017), stars reflect the popularity of a project. The rationale for using stars being that it is unlikely for popular GitHub projects to not be engineered. Empirically, Munaiah et al. found that a project having at least 1,123 stars is very likely to be engineered. However, the opposite is not always true, meaning that there are also projects that have less than 1,123 stars but are engineered. We decided to use a lower threshold, in particular we choose 10, to include more projects that are engineered. We believe that this choice is not problematic because we filter projects also by other criteria that reflect the amount of engineering. Furthermore, we manually check our final project selection, which removes projects that do not meet our minimal criteria. After this step, 683 projects that have more than 10 stars remain in our data set.

Maven projects As already mentioned, we use BUILDDIFF, which we introduced in our previous work (Macho et al. 2017) for our analysis. BUILDDIFF extracts the build changes that were performed between two consecutive Maven build specification files (`pom.xml`). We determined whether a project uses Apache Maven as build tool by querying the projects GitHub repository and verifying the presence of a `pom.xml` file in the current commit on the default branch. At this point, we add the 30 projects that were used in our original study (Macho et al. 2017).[2] Finally, we are left with 446+30=476 projects.

Commit count/recency From the remaining projects, we only keep those that are sufficiently large and actively developed. We measure the size in commits and keep the projects that have at least 100 commits. After this step, we keep 295 projects. Recency is measured by checking the date of the last commit. We consider a project as actively developed if the last commit was performed on 01.01.2019 or later. After this filter, our data set consists of 151 projects.

Manual verification With the final 151 projects that we retrieved after applying all criteria, we perform a manual analysis. We read the description that is given on GitHub and verify whether it contains any indication of being a toy project, a sample project, or an abandoned project. We excluded another 7 projects, such as `goldmansachs/gs-collections` as the authors state that this project is no longer active, and `pires/obd-java-api` because it is stated that this project is no longer maintained. After the manual verification of the projects, we are left with a total of 144 projects that we use for our experiments.

[2] 20 projects were selected using GitHub to retrieve an initial list of projects that have more than 1000 stars. Second, we focused on larger projects filtering out projects with less than 3500 commits and those that are not using Maven. The resulting list of project was sorted by the sum of stars and commits to balance both metrics. Lastly, we added the 10 projects that we used in our prior work (Macho et al. 2016).

Change extraction For each project, we extracted the build changes as follows: First, we cloned the repository and iterated over each commit, including commits on branches. Second, we checked for modifications in MAVEN build files (`pom.xml`) indicated by Git. For each of the modified build files, we determined its preceding version and passed both versions of the build file to BUILDDIFF to extract the MAVEN build changes. The extracted changes were stored in a database called the ChangeDB.

Release extraction For our experiments on when the build changes are performed, we need information on when the releases of each project were created. Hence, we extract the release information that is provided by GitHub. We query the REST-API of GitHub that provides access to the tags of a repository for each project to retrieve all tags of a project. For each retrieved tag, we determine whether it is an actual release or only a git tag. We consider a tag being a release if it contains a version number that matches at least one of the following patterns: (1) “MAJOR.MINOR.PATCH”, or (2) “MAJOR.MINOR”, both with optional suffixes (*e.g.* 1.0.0-Alpha or 1.0GA). Raemaekers et al. (2014) found that these two patterns are the most common version patterns in their study of the MAVEN repository. A preliminary investigation of our data set suggests that most of the projects use this pattern for versioning as well. Hence, other tags, such as development tags that only contain text, are not considered a release tag. All release tags are stored in a database called the ReleaseDB.

The experiments of this paper are based on the ChangeDB and the ReleaseDB, both of which are available online.[3] Table 1 shows the top-50 projects sorted by the number of commits of the resulting data set with descriptive statistics. The number of commits within the top-50 projects varies between 2,556 and 65,374. However, the number of commits that contain a build change only ranges between 63 and 12,543 which results in build change ratios between 0.01 and 0.40. Furthermore, the number of releases that we could retrieve varies between 2 and 5,322. The last commit of each of the projects was after 01.01.2019 and before the data retrieval date on 03.07.2019.

4 Extracting build changes with BUILDDIFF

In this section, we describe our approach to extract build changes from build files. Currently, we focus on the extraction of MAVEN build files. MAVEN build files are named `pom.xml` following the naming convention of MAVEN. First, we define a taxonomy of build changes and provide our rationale for the defined changes. Second, we describe BUILDDIFF, our approach to extract build changes of two consecutive MAVEN build file revisions.

4.1 Taxonomy

MAVEN build files are specified using a special type of XML. Hence, we can easily read, parse, and transform their content into a tree that corresponds to the MAVEN schema[4] that defines the various XML elements and attributes used for configuring a MAVEN build. Having the content of a MAVEN build file represented as a tree, we then can use tree differencing algorithms, such as ChangeDistiller (Fluri et al. 2007) or GumTree (Falleri et al. 2014), to extract the differences between two build files. We use the modified version of the GumTree

[3]`https://zenodo.org/record/4153674#.X5rNXlNKhTY`

[4]http://maven.apache.org/xsd/maven-4.0.0.xsd

Table 1 Top-50 Java projects sorted by the number of commits used for evaluating BUILDDIFF and for studying the evolution of build files plus descriptive statistics of #BCC (Number of commits with Build Change), BCCR (ratio of BCC), #R (Number of extracted Releases), #BC (Build Changes), BCPC (Build Changes Per Commit), and LC (Last Commit)

Name	#Commits	#BCC	BCCR	#R	#BC	BCPC	LC
apache/hadoop	65374	3244	0.05	315	32247	0.49	2019-07-03
neo4j/neo4j	62303	8801	0.14	273	76151	1.22	2019-07-02
apache/camel	49686	12543	0.25	144	191315	3.85	2019-07-03
apache/hbase	45836	3428	0.07	699	22571	0.49	2019-07-03
deeplearning4j/deeplearning4j	39879	3169	0.08	52	23351	0.59	2019-07-02
apache/wicket	32937	1789	0.05	271	16814	0.51	2019-07-02
languagetool-org/languagetool	31421	389	0.01	28	6039	0.19	2019-07-03
Alluxio/alluxio	30966	1217	0.04	52	7695	0.25	2019-07-03
hazelcast/hazelcast	29948	1181	0.04	178	6147	0.21	2019-07-03
jenkinsci/jenkins	29875	4484	0.15	659	23889	0.80	2019-07-03
wildfly/wildfly	29545	5067	0.17	91	45059	1.53	2019-07-03
SonarSource/sonarqube	29287	1886	0.06	173	21262	0.73	2019-07-02
eclipse/eclipse.jdt.core	28990	238	0.01	5322	1197	0.04	2019-07-03
orientechnologies/orientdb	23031	1227	0.05	145	7927	0.34	2019-07-03
spring-projects/spring-boot	22631	5243	0.23	129	47421	2.10	2019-07-02
apache/flink	21747	2191	0.10	94	20578	0.95	2019-07-03
eclipse/jetty.project	20104	3031	0.15	331	86371	4.30	2019-07-03
apache/karaf	18478	6600	0.36	80	65044	3.52	2019-07-02
Graylog2/graylog2-server	16940	2210	0.13	193	7660	0.45	2019-07-03
prestodb/presto	16122	1602	0.10	237	18542	1.15	2019-07-02
stanfordnlp/CoreNLP	15796	882	0.06	2	3508	0.22	2019-07-03
netty/netty	15114	1703	0.11	203	13337	0.88	2019-07-03
google/closure-compiler	14557	166	0.01	159	434	0.03	2019-07-02
apache/storm	13623	1238	0.09	39	14941	1.10	2019-07-01
apache/activemq	11993	1871	0.16	63	14242	1.19	2019-06-21
naver/pinpoint	11091	954	0.09	30	7770	0.70	2019-07-03
druid-io/druid	10526	1756	0.17	427	19967	1.90	2019-07-03
hibernate/hibernate-search	10120	1991	0.20	153	14336	1.42	2019-07-03
spring-projects/spring-roo	6624	642	0.10	39	10123	1.53	2019-06-05
google/guava	6580	318	0.05	87	1312	0.20	2019-06-30
OpenHFT/Chronicle-Engine	6209	850	0.14	162	1170	0.19	2019-01-24
apache/commons-lang	5933	393	0.07	87	602	0.10	2019-07-01
GluuFederation/oxAuth	5104	480	0.09	20	2384	0.47	2019-07-03
square/okhttp	4392	336	0.08	67	2623	0.60	2019-07-03
christophd/citrus	4196	615	0.15	31	4936	1.18	2019-06-10
alibaba/fastjson	3801	331	0.09	114	622	0.16	2019-06-28
cometd/cometd	3768	906	0.24	82	11458	3.04	2019-06-11
CloudSlang/cloud-slang	3692	811	0.22	225	5529	1.50	2019-06-30
cloudera/sentry	3564	823	0.23	100	9600	2.69	2019-04-12
ThreeTen/threetenbp	3446	87	0.03	21	305	0.09	2019-05-24

Table 1 (continued)

Name	#Commits	#BCC	BCCR	#R	#BC	BCPC	LC
jqno/equalsverifier	3145	251	0.08	82	458	0.15	2019-05-16
apache/maven-surefire	2996	1021	0.34	70	6704	2.24	2019-06-10
spring-projects/spring-data-gemfire	2719	491	0.18	139	1224	0.45	2019-06-28
jmock-developers/jmock-library	2667	174	0.07	171	951	0.36	2019-07-01
junit-team/junit	2569	63	0.02	23	108	0.04	2019-06-21
joel-costigliola/assertj-core	2533	370	0.15	45	497	0.20	2019-06-30
kaazing/k3po	2503	413	0.17	130	5033	2.01	2019-06-25
jankotek/MapDB	2498	294	0.12	84	575	0.23	2019-04-09
sonatype/sisu-guice	2294	362	0.16	32	2664	1.16	2019-01-31
jooby-project/jooby	2274	701	0.31	66	11387	5.01	2019-07-03

implementation of Dotzler and Philippsen (2016) to extract edit operations that transform one tree into the other. In the remainder of the paper, we refer to this implementation as GumTree. We describe the extraction procedure in more detail in Section 4.2.

Similar to ChangeDistiller, we defined the change types of our taxonomy based on the edit operations extracted by the tree differencing algorithm whereas the structure of the tree and its different elements correspond to the MAVEN schema. For defining the taxonomy, we started with the top level elements of the MAVEN schema and moved down the schema until we reached the bottom-most child elements. For each element (*i.e.* XML tag), we defined change types for inserting (*_INSERT), deleting (*_DELETE), and updating (*_UPDATE) that element. For some particular tags, such as `artifactId` and `groupId`, we only created the *_UPDATE change type because they are mandatory for the definition of particular tags, such as `dependency`, and we assume that they are inserted and deleted with their parent tag. This is further described in detail in Section 4.2. The resulting taxonomy currently comprises 143 build change types that we validated with two persons. The two persons are PhD students with the focus in Software Engineering who have been using MAVEN for at least 10 years in practice and who are actively researching in the area of build systems and MAVEN in particular. We argue that they have the necessary expertise to perform the evaluation because of their regular and in-depth usage of MAVEN. Compared to the original paper of this study (Macho et al. 2017), the taxonomy was refined concerning version changes. We asked the two persons to adapt the existing taxonomy to include the new change types, separately. In a second step, they then discussed differences in their assignments to finally reach consensus about the taxonomy. This also explains the increase of the number of different build change types from 95 to 143.

For all of the change types, we also extract the content that has been changed. For example, if a version has been updated from 1.0.0 to 1.0.1, we also extract 1.0.0 and 1.0.1 and refer to them as the old and new value, respectively. This information is important for further analyses because more information about the changes can be used. For example, the precise type of a version update is determined using these values which is then used in Section 6 to distinguish the type of a release. We found in another study (Macho et al. 2018) that version changes play a crucial role in build specifications. To address this finding, we extended BUILDDIFF and added a finer extraction of versions changes. We replaced the change types representing version changes (*i.e.* change types ending with "_VERSION_UPDATE"), such as DEPENDENCY_VERSION_UPDATE, with nine fine-grained change types for each old

Table 2 Fine-grained version change types

Old Value	New Value	Change Type
1.0.0	**2**.0.0	MAJOR_VERSION_INCREASE
2.0.0	**1**.0.0	MAJOR_VERSION_DECREASE
1.**1**.0	1.**2**.0	MINOR_VERSION_INCREASE
1.**2**.0	1.**1**.0	MINOR_VERSION_DECREASE
1.0.**1**	1.0.**2**	PATCH_VERSION_INCREASE
1.0.**2**	1.0.**1**	PATCH_VERSION_DECREASE
1.0.0-**Alpha1**	1.0.0-**Alpha2**	POSTFIX_VERSION_UPDATE
1.0.0	version1	UNKNOWN_VERSION_UPDATE
1.0.0	1.0.0	NO_VERSION_UPDATE

build change type. The fine-grained change types are derived from the difference in the version value. Table 2 shows examples of the new change types.

We also grouped the change types into categories. We retrieved the categories and the respective change type assignments by performing card sorting (Nielsen 1995). First, we gave the list of change types to the two selected persons who validated the changes types separately and asked them to group the change types. Second, we asked the developers to assign names to the created groups. In a third step, we asked both developers to discuss their categories and assignments to arrive at a common categorization. If the developers assigned a change to different categories they discussed their assignments to reach consensus.

The two developers identified the following 5 categories: (1) `Dependency Changes` contain all changes that are related to dependencies of the project, (2) `Build Changes` cover the changes that directly affect or modify the build process, (3) `Team Changes` comprise all modifications to the list of team members, (4) `Repository Changes` contain changes that are performed to the distribution and repository locations, and (5) `General Changes` contain changes that are made to the general items of a MAVEN project. Table 3 shows an excerpt of the taxonomy with examples of change types for each category. The full taxonomy can be found online.[5]

In the following, we provide four examples of frequently occurring build changes. The first example depicted in .- 1 (old version of the build file) and Listing 2 (new version of the build file) shows a change of the version of a dependency to the `spring-core` library from `4.2.5.RELEASE` to `4.2.6.RELEASE`. This change refers to a version change of a dependency. BUILDDIFF extracts this change as DEPENDENCY_PATCH_VERSION_-INCREASE because the PATCH part of the version value changed from 5 to 6.

The second example depicted in Listing 3 (old version of the build file) and Listing 4 (new version of the build file) shows the insertion of the `maven-jar-plugin` plugin. This change is extracted and classified by BUILDDIFF as PLUGIN_INSERT.

The third example shows a change in the configuration of a plugin. In particular, Listing 5 shows the configuration of a source and target version of 1.7. In the new version in Listing 6 these values have been updated to 1.8. BUILDDIFF extracts these changes as PLUGIN_CONFIGURATION_UPDATE.

The forth example concerns the definition of properties. Properties in Maven are used as placeholders similar to variables in programming languages. A very common usage of

[5]https://zenodo.org/record/4153674#.X5rNXlNKhTY

Table 3 Excerpt of our Taxonomy of Build Changes

Category	Change Types (Excerpt)
`Dependency Changes`	DEPENDENCY_INSERT DEPENDENCY_VERSION_UPDATE MANAGED_DEPENDENCY_DELETE
`Build Changes`	PLUGIN_INSERT PLUGIN_CONFIGURATION_UPDATE TEST_RESOURCE_DELETE
`Team Changes`	DEVELOPER_INSERT CONTRIBUTOR_DELETE
`Repository Changes`	PLUGIN_REPOSITORY_INSERT DIST_SNAPSHOT_REPOSITORY_UPDATE REPOSITORY_DELETE
`General Changes`	MODULE_INSERT PARENT_VERSION_UPDATE GENERAL_PROPERTY_DELETE

```
<dependency>
 <groupId>org.springframework</groupId>
 <artifactId>spring-core</artifactId>
 <version>4.2.5.RELEASE</version>
</dependency>
```

Listing 1 Example of a Dependency Version Update - Old Version

```
<dependency>
 <groupId>org.springframework</groupId>
 <artifactId>spring-core</artifactId>
 <version>4.2.6.RELEASE</version>
</dependency>
```

Listing 2 Example of a Dependency Version Update - New Version

```
<build>
<plugins>
</plugins>
</build>
```

Listing 3 Example of a Plugin Insertion - Old Version

```
<build>
 <plugins>
  <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-jar-plugin</artifactId>
   <version>2.6</version>
  </plugin>
 </plugins>
</build>
```

Listing 4 Example of a Plugin Insertion - New Version

properties is to extract version numbers. For example, in Listing 7 the property `maven-assembly-plugin.version` is assigned the value 2.3 and in Listing 8 this value is updated to 2.4. BUILDDIFF extracts this changes as GENERAL_PROPERTY_UPDATE.

4.2 Approach

This section presents our BUILDDIFF approach to extract 143 types of changes from MAVEN build files. Our approach is mainly motivated and inspired by the work of Gall et al. (2009) and Fluri et al. (2007), who showed that detailed information on source code changes can aid in understanding the evolution of software projects, and our previous work (Macho et al. 2016) that showed that this information can be used for computing models to predict when build configurations should be updated.

Concerning changes in build configuration files, in particular MAVEN build files, the finest level of analysis that has been performed was on line level. Désarmeaux et al. (2016) mapped lines of a MAVEN `pom.xml` to the respective build lifecycle phase. To the best of our knowledge, BUILDDIFF is the first approach to extract changes in MAVEN build files on the level of MAVEN configuration elements, that we refer to as fine-grained build changes. Compared to our previous work (Macho et al. 2017), we extend BUILDDIFF and refine the extraction of version changes.

BUILDDIFF first reads two versions of a MAVEN build file (*i.e.* `pom.xml`) and represents each version as a tree. Then, it uses the GumTree (Falleri et al. 2014) implementation of Dotzler and Philippsen (2016) to extract the differences between the two trees in terms of edit operations to transform one tree into the other. The list of edit operations is then mapped

```
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-compiler-plugin</artifactId>
 <version>3.5.1</version>
 <configuration>
  <source>1.7</source>
  <target>1.7</target>
 </configuration>
</plugin>
```

Listing 5 Example of a Plugin Configuration Update - Old Version

```
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-compiler-plugin</artifactId>
 <version>3.5.1</version>
 <configuration>
  <source>1.8</source>
  <target>1.8</target>
 </configuration>
</plugin>
```

Listing 6 Example of a Plugin Configuration Update - New Version

to the 143 change types that are defined in our taxonomy. In the following, we present each step in detail:

Preprocess build files The first step of BUILDDIFF preprocesses the two versions of a MAVEN build file. MAVEN build files are descriptive, meaning that the order of the elements in the file can be changed without changing its semantics. We observed that GumTree can match elements of the same level more accurately if they are sorted. Hence, BUILDDIFF first sorts the elements on the same level according to their content.[6] For example, the tag `<module>MySubmodule</module>` appears before `<module>TheModule</module>`. Furthermore, BUILDDIFF removes comments and attributes. Attributes, such as `combine .children` and `combine.self` for plugin configuration inheritance, affect the build configuration at execution time. We only analyze the build configuration from a static point of view and hence, we remove attributes.

Extract edit operations Next, BUILDDIFF parses the two preprocessed versions of a MAVEN build file into two trees and passes them to the GumTree differencing algorithm. GumTree provides a TreeGenerator for XML files. Unfortunately, this implementation does not handle values of tags in XML documents. Therefore, we implemented our own Tree-Generator that transforms XML files into GumTree trees. We use the prominent Java XML library *jdom* to read the XML file, and methods provided by GumTree to create the tree.

GumTree then uses a `Matcher` instance to find mappings between two trees. BUILD-DIFF extends the GumTree's default matcher by adding a mechanism to ensure that only tags with the same name will be matched, and by modifying the similarity calculation of two nodes. Tags that have a child tag named `id` are matched if the Levenshtein similarity of the `id` value exceeds a threshold t. The matcher chooses the node with the highest similarity exceeding the threshold. Tags that have the MAVEN triplet (`groupId`, `artifactId`, `version`) as child nodes are matched by applying the Levenshtein distance for `groupId` and `artifactId`. Two nodes are matched if the Levenshtein similarity exceeds a threshold t. The matcher chooses the node with the highest similarity exceeding the threshold. Experiments with different t values suggest that $t = 0.65$ yields the best performance.

Given the matcher, GumTree outputs a list of tree edit operations comprising added, deleted, updated, and moved elements in the tree that transform the source tree (previous version of the MAVEN build file) into the target tree (subsequent version of the MAVEN build file).

[6]Strings are sorted alphabetically and numbers in ascending order

```
<properties>
 <maven-assembly-plugin.version>2.3</maven-assembly-plugin.
     version>
</properties>
```

Listing 7 Example of a General Property Update - Old Version

Sort edit operations BUILDDIFF considers a particular order to process the changes in MAVEN files. We process the operations of the edit script in a top down order according to their level in the build file (parent nodes first). BUILDDIFF applies this order to prevent the extraction of additional changes that result from the insertion and deletion of composite MAVEN tags that also insert or delete their children at the same time. For instance, when a new dependency is inserted, BUILDDIFF only records a DEPENDENCY_INSERT, skipping the insertion of the child tags of that dependency (*e.g.* `groupId`, `artifactId`, and `version`).

Map build changes In this step, BUILDDIFF maps the tree edit operations that are generated by GumTree to the 143 change types of our taxonomy. We consider insertions, deletions, and updates. We do not consider moves, since MAVEN build files are descriptive, meaning that the order of the elements in the file can be changed without changing its semantics.

To map edit operations to change types, BUILDDIFF iterates over the sorted edit operations mapping each edit operation to at most one change type. Changes to child elements are handled by first checking whether the change is part of an insertion, deletion, or update of its parent. In that case, the change to the child element is not mapped, since it is already part of the parent change. For instance, the insertion of a dependency is mapped only to the change DEPENDENCY_INSERT while the insertions of its child elements `groupId`, `artifactId`, and `version` are skipped.

For version updates, the old and new values are investigated to retrieve the fine-grained change type of the version change. We first parse both values and detect the format of the version value. If at least one fails the parsing, UNKNOWN_VERSION_UPDATE is extracted (*e.g.* when changing from “1.0” to “version1”). Otherwise, we compare the components of the version definitions. First, we compare the MAJOR version. If the old value differs from the new value we extract MAJOR_VERSION_INCREASE or MAJOR_VERSION_DECREASE depending on which value is larger. If they are equal, we proceed with MINOR and eventually repeat the procedure with PATCH. If all three are equal but the POSTFIX differs, we extract POSTFIX_VERSION_UPDATE. For the postfix, we cannot extract a direction of the value, because postfixes usually are Strings.

As a result, BUILDDIFF outputs a list of build changes that have been performed between two versions of a MAVEN file.

```
<properties>
 <maven-assembly-plugin.version>2.4</maven-assembly-plugin.
     version>
</properties>
```

Listing 8 Example of a General Property Update - New Version

5 Evaluating BuildDiff

In this section, we describe the evaluation of BuildDiff. We present the evaluation of our prototype implementation of the BuildDiff approach in terms of precision and recall. Finally, we discuss examples of correctly and incorrectly extracted changes.

We evaluated BuildDiff by performing a manual evaluation with 400 build-changing commits that were randomly selected from the build files of the 144 open source Java projects. Compared to our previous work (Macho et al. 2017), we added an additional evaluation for the tool and included 114 more projects in the evaluation. Moreover, we present more details of the evaluation than in the previous work.

In our original paper (Macho et al. 2017), we invited two PhD students with profound Maven knowledge to evaluate 400 build-changing commits to show that BuildDiff is also working on real world projects and that it can extract build changes as they are understood by software developers. This paper contributes 48 new types of build changes to the BuildDiff tool which need to be evaluated. To that end, we repeat the evaluation on the data set described in Section 3 to evaluate the performance of BuildDiff. This time, we asked one additional software engineering PhD student to perform the evaluation. Compared to our previous work (Macho et al. 2017), we only refined the existing change types that concern version changes. Hence, we believe that one additional evaluation of the new version of the approach is sufficient to show the validity of BuildDiff. The PhD student received a set of 400 `pom.xml` pairs containing 3685 build changes. To ensure that the results of the evaluation are not affected by fatigue or any other effect that may impact the evaluation, we calculated the inter-rater agreement of 400 randomly selected build changes that were then also evaluated by the second author of the study. This inter-rater agreement calculated with Cohen's κ (Cohen 1968) showed an agreement of 0.982 which shows that the evaluators strongly agree.[7]

Prior to the experiment, we briefly explained our taxonomy of build changes to her. The `pom.xml` files under investigation were provided side-by-side and changed lines were highlighted, similar to the diff view available on code hosting platforms, such as GitHub. Other tools to support the participants were not allowed. In the experiment, we then asked her to label the changes in the `pom.xml` pairs according to our change taxonomy and compared the output with the output of BuildDiff.

Data selection Table 1 shows an excerpt of the list of open source Java projects from which we randomly selected 400 commits that contained changes to a `pom.xml` build file. We calculated the sample size based on a population size of 153,321 commits that contain build changes (Table 1, sum of the full column #BCC), with a margin of error of 5% and a confidence level of 95%. The minimum sample size is 384 commits[8] and we finally decided to randomly select 400 commits to exceed the minimum sample size.

The total amount of build changes in the data set is 4,016,838. By randomly selecting the commits for the evaluation, we missed to evaluate 55 of the 143 change types (or 38%). However, the analysis of these missing change types in our subject systems showed that these changes only represent 1.0% of the total build changes (41,097 out of 4,016,838). In our original work, we covered a similar ratio of change types (44%) with a slightly lower

[7] The seven disagreements were caused by reading errors by one of the evaluators.

[8] https://www.checkmarket.com/sample-size-calculator/

ratio of missing build changes (0.9%). However, we can still safely assume that our sample set sufficiently covers the majority of build changes in our data set.

Evaluation For each of the 400 commits, we provided the study participant with the original and modified version of the build file. We asked the participant to go through all of the selected commits and assign each change to the corresponding change type from our taxonomy. We then compared the changes that were assigned by the participant with the list of build changes extracted by BUILDDIFF. With the results of the participant, we calculated *precision*, *recall*, and *f1-score* to measure the performance of our approach. Following the approach of Fluri et al. (2007) used for evaluating ChangeDistiller, we calculated precision, recall, and, furthermore, f1-score as:

$$\text{precision} = \frac{\text{\#relevant changes found}}{\text{\#changes found}}$$

$$\text{recall} = \frac{\text{\#relevant changes found}}{\text{\#changes expected}}$$

$$\text{f1-score} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

Precision measures how many of the changes that were extracted by our approach were also detected by a study participant. *Recall* measures how many of the changes that a study participant has found have also been found by our approach. *F1-score* combines *precision* and *recall* in one value by calculating the harmonic mean of both. Similar to the evaluation of Dintzner et al. (2014), we are able to evaluate the correctness and the completeness of our approach with these performance measures. The results of the evaluation in the original study showed high precision, recall, and f1-score of 0.9513, 0.9796, and 0.9652, respectively for Participant 1. For Participant 2, the results show a precision of 0.9601, a recall of 0.9844, and an f1-score of 0.9721. Averaging the values of both participants, we obtained a mean precision of 0.9557, a mean recall of 0.9820 and a mean f1-score of 0.9687. The additional evaluation of this work that was performed on 400 randomly selected revisions shows a precision, recall, and f1-score of 0.9697 0.9796, and 0.9746, respectively. Comparing these results with the original evaluation, we can see that the performance of our tool was not affected by our extension. This means that the new version of BUILDDIFF is capable of detecting more fine-grained changes while keeping its high performance.

As stated above, BUILDDIFF did not perform perfectly on all commits of our evaluation sample. Investigating the number of build changes of a commit and the respective metrics of the evaluation, we state the hypothesis that the more build changes there are in a commit, the more error-prone is our approach. We substantiate this hypothesis by comparing the number of build changes in commits with a perfect f1-score (*i.e.* $f = 1$) with the number of build changes in commits with less than a perfect f1-score (*i.e.* $f < 1$) with a Mann-Whitney U-Test and the Cliff's Delta as effect size measure. The results show that the distributions are significantly different $p < 2.2e - 16$. Furthermore, the effect size is 0.79 which is considered large (Grissom and Kim 2005). Based on these results, we state that with an increasing number of build changes the performance of BUILDDIFF decreases.

In addition to these aggregated values, we investigate the metrics per build change type. Table 4 shows the results of this investigation. We see that for 55 (65%) of the build change types BUILDDIFF extracts the change type with a perfect performance. This group contains changes, such as the three GENERAL_PROPERTY_* changes. For 15 (18%) of the change types, BUILDDIFF achieves an f1-score of more than 0.8, such as DEPENDENCY_-INSERT, and for 12 (14%) of the changes types, such as PROFILE_UPDATE, of at least

Table 4 Detailed evaluation of BUILDDIFF per build change type

Change Type	Prec	Rec	f1-score
CONTRIBUTOR_INSERT	1.00	1.00	1.00
DEPENDENCY_MINOR_VERSION_DECREASE	1.00	1.00	1.00
DEPENDENCY_MINOR_VERSION_INCREASE	1.00	1.00	1.00
DEPENDENCY_PATCH_VERSION_INCREASE	1.00	1.00	1.00
DEPENDENCY_POSTFIX_VERSION_UPDATE	1.00	1.00	1.00
DEPENDENCY_VERSION_DELETE	1.00	1.00	1.00
DEVELOPER_DELETE	1.00	1.00	1.00
DEVELOPER_INSERT	1.00	1.00	1.00
DIST_REPOSITORY_DELETE	1.00	1.00	1.00
DIST_SNAPSHOT_REPOSITORY_DELETE	1.00	1.00	1.00
GENERAL_PROPERTY_DELETE	1.00	1.00	1.00
GENERAL_PROPERTY_INSERT	1.00	1.00	1.00
GENERAL_PROPERTY_UPDATE	1.00	1.00	1.00
LICENSE_INSERT	1.00	1.00	1.00
MANAGED_DEPENDENCY_MAJOR_VERSION_INCREASE	1.00	1.00	1.00
MANAGED_DEPENDENCY_MINOR_VERSION_DECREASE	1.00	1.00	1.00
MANAGED_DEPENDENCY_MINOR_VERSION_INCREASE	1.00	1.00	1.00
MANAGED_DEPENDENCY_PATCH_VERSION_INCREASE	1.00	1.00	1.00
MANAGED_DEPENDENCY_POSTFIX_VERSION_UPDATE	1.00	1.00	1.00
MANAGED_DEPENDENCY_UNKNOWN_VERSION_UPDATE	1.00	1.00	1.00
MODULE_DELETE	1.00	1.00	1.00
MODULE_INSERT	1.00	1.00	1.00
PARENT_ARTIFACTID_UPDATE	1.00	1.00	1.00
PARENT_GROUPID_UPDATE	1.00	1.00	1.00
PARENT_MAJOR_VERSION_INCREASE	1.00	1.00	1.00
PARENT_MINOR_VERSION_DECREASE	1.00	1.00	1.00
PARENT_MINOR_VERSION_INCREASE	1.00	1.00	1.00
PARENT_PATCH_VERSION_DECREASE	1.00	1.00	1.00
PARENT_PATCH_VERSION_INCREASE	1.00	1.00	1.00
PARENT_POSTFIX_VERSION_UPDATE	1.00	1.00	1.00
PARENT_UNKNOWN_VERSION_UPDATE	1.00	1.00	1.00
PARENT_UPDATE	1.00	1.00	1.00
PLUGIN_CONFIGURATION_DELETE	1.00	1.00	1.00
PLUGIN_CONFIGURATION_INSERT	1.00	1.00	1.00
PLUGIN_DEPENDENCY_MINOR_VERSION_INCREASE	1.00	1.00	1.00
PLUGIN_MINOR_VERSION_DECREASE	1.00	1.00	1.00
PLUGIN_REPOSITORY_DELETE	1.00	1.00	1.00
PLUGIN_REPOSITORY_UPDATE	1.00	1.00	1.00
PLUGIN_VERSION_DELETE	1.00	1.00	1.00
PLUGIN_VERSION_INSERT	1.00	1.00	1.00
PROFILE_DELETE	1.00	1.00	1.00
PROFILE_INSERT	1.00	1.00	1.00
PROJECT_ARTIFACTID_UPDATE	1.00	1.00	1.00
PROJECT_MAJOR_VERSION_DECREASE	1.00	1.00	1.00

Table 4 (continued)

Change Type	Prec	Rec	f1-score
PROJECT_MAJOR_VERSION_INCREASE	1.00	1.00	1.00
PROJECT_MINOR_VERSION_DECREASE	1.00	1.00	1.00
PROJECT_MINOR_VERSION_INCREASE	1.00	1.00	1.00
PROJECT_NAME_DELETE	1.00	1.00	1.00
PROJECT_PATCH_VERSION_DECREASE	1.00	1.00	1.00
PROJECT_PATCH_VERSION_INCREASE	1.00	1.00	1.00
PROJECT_POSTFIX_VERSION_UPDATE	1.00	1.00	1.00
REPOSITORY_DELETE	1.00	1.00	1.00
REPOSITORY_INSERT	1.00	1.00	1.00
SCM_URL_INSERT	1.00	1.00	1.00
SCM_URL_UPDATE	1.00	1.00	1.00
MANAGED_DEPENDENCY_INSERT	1.00	0.98	0.99
PLUGIN_INSERT	1.00	0.93	0.96
DEPENDENCY_INSERT	0.93	0.97	0.95
SCM_CONNECTION_UPDATE	1.00	0.90	0.95
DEPENDENCY_DELETE	0.93	0.96	0.95
SCM_DEVCONNECTION_UPDATE	1.00	0.88	0.93
DEPENDENCY_UPDATE	0.89	0.93	0.91
PLUGIN_DELETE	0.81	1.00	0.90
PLUGIN_CONFIGURATION_UPDATE	0.91	0.88	0.90
RESOURCE_UPDATE	0.80	1.00	0.89
PROFILE_UPDATE	0.92	0.86	0.89
REPOSITORY_UPDATE	0.78	1.00	0.88
DEPENDENCY_MAJOR_VERSION_INCREASE	0.75	1.00	0.86
PLUGIN_MINOR_VERSION_INCREASE	0.75	1.00	0.86
SCM_CONNECTION_INSERT	0.75	1.00	0.86
DEPENDENCY_UNKNOWN_VERSION_UPDATE	1.00	0.67	0.80
MANAGED_DEPENDENCY_DELETE	1.00	0.67	0.80
PLUGIN_PATCH_VERSION_INCREASE	0.67	1.00	0.80
PLUGIN_POSTFIX_VERSION_UPDATE	0.67	1.00	0.80
PLUGIN_REPORT_SET_UPDATE	1.00	0.67	0.80
RESOURCE_INSERT	0.67	1.00	0.80
SCM_DEVCONNECTION_INSERT	0.67	1.00	0.80
PLUGIN_MAJOR_VERSION_INCREASE	0.50	1.00	0.67
PLUGIN_UNKNOWN_VERSION_UPDATE	0.50	1.00	0.67
PROJECT_NAME_UPDATE	0.50	1.00	0.67
PLUGIN_UPDATE	0.38	0.83	0.53
MANAGED_DEPENDENCY_UPDATE	0.33	1.00	0.50
SCM_CONNECTION_DELETE	0.00	1.00	0.00
SCM_DEVCONNECTION_DELETE	0.00	1.00	0.00

```
<dependency>
 <groupId>com.typesafe.akka</groupId>
 <artifactId>akka-testkit_${scala.binary.version}</artifactId>
 <scope>test</scope>
</dependency>
```

Listing 9 Example of an incorrect Classification - Old Version

```
<dependency>
 <groupId>com.data-artisans</groupId>
 <artifactId>flakka-testkit_${scala.binary.version}</artifactId>
 <scope>test</scope>
</dependency>
```

Listing 10 Example of an incorrect Classification - New Version

0.5. Finally, for 2 (2%) of the change types, the f1-score is lower than 0.5. A detailed investigation of those two build change types (*i.e.* SCM_CONNECTION_DELETE, SCM_-DEVCONNECTION_DELETE) revealed that they occur exactly once in the evaluation data set in the same single commit and were misclassified. In both cases, the tool could not match the respective nodes and hence, it extracted a delete and an insert action. However, the correct action was an update because a human would recognize the match and classify the change as an update action.

Besides the quantitative evaluation, we performed a qualitative evaluation to find out in which scenarios BUILDDIFF shows a good performance and in which it does not. We present an example where BUILDDIFF did not properly extract and classify the build changes compared to a human evaluation. We found that the most common error can be attributed to the similarity analysis. This means that most of the wrongly classified changes are due to the fact that our similarity measure indicated a too low similarity and hence, two nodes could not be matched, which ultimately leads to a wrong change extraction. The example is taken from the flink project[9] and shows such a case. In this example, a dependency definition was changed. Listing 9 shows the dependency in the old version and Listing 10 shows the updated version of the dependency. BUILDDIFF extracted two changes, DEPENDENCY_DELETE and DEPENDENCY_INSERT. In fact, this is an update of the same dependency where the `groupId` and the `artifactId` changed simultaneously. Hence, the correct classification would be a DEPENDENCY_UPDATE. Our approach could not detect this change correctly because we use a distance measure to match the nodes of two build files. In this case, the measure indicated that the two dependency definitions are not close enough to be considered the same dependency and consequently, BUILDDIFF extracted the two changes wrongly.

In conclusion, we observe that:

BUILDDIFF is capable of extracting changes from MAVEN build files with an average precision of 0.97, an average recall of 0.98, and an average f1-score of 0.97. Updates to the dependency versions and changes to general properties are among the change types that achieve the best performance.

[9]http://goo.gl/rWPFDy

6 Build change frequency (RQ1)

In our first experiment, we investigate the frequency of build changes. We aim at gaining knowledge of which change types are frequently performed in projects and how often they are performed. Furthermore, we identify patterns of build changes that occur frequently in software projects. This information can help to understand the evolution of build files similar to the study of Gall et al. (2009). With these experiments, we answer RQ1: *Which build changes occur the most frequently?* We split this research question into two sub research questions:

RQ1.1 *What are the most frequently occurring build change types?*
The aim of this research question is to get a deeper insight into the frequency of build specification changes and the types of changes that are performed during a project. This topic has been addressed in our previous work (Macho et al. 2017). In this paper, we extend it by including 204 additional projects in the study to increase the generalizability of the results and explore the differences to the original work. The results of this research question can support researchers in future directions for build system research, such as our recent work on automatically repairing broken builds (Macho et al. 2018).

RQ1.2 *What are the most frequently occurring build change patterns?*
In this research question, we enrich our prior work with insights into the characteristics of build change patterns. We identify frequently performed change patterns and give examples of their occurrence. With the results of this research question, we can support developers when changing build specification files *e.g.* by warning them that additionally to their current build changes, another corresponding build change might be missing.

6.1 Frequency of build changes (RQ1.1)

In this section, we investigate the frequency of build changes. We use the data set that we described in Section 3 and count the frequency of build changes in commits that changed the build specification. In the following, we describe the approach that we used to count the build changes. Then, we present the most frequent build change types and validate the quantitative results with a manual investigation. This section concludes with the discussion of our results.

6.1.1 Approach

Starting with the data stored in the ChangeDB, we iterated over the 144 projects and counted the occurring changes. We counted the number of each change type per project and aggregated the numbers also per change category. As depicted in Table 1, the selected projects differ in their size, and hence, contain a different amount of total build changes (column BC). Given this variance in the number of build changes, we normalized the change counts to align the impact of each project on the result. We divided each aggregated change count by the number of total build changes in the project. For example, project `spring-roo` (row 30 in Table 1) contains a total amount of 11,561 build changes and 242 instances of the change type MODULE_INSERT. We calculated the relative occurrence of this change type with 242/11561 = 2.09% and used this relative value instead of the absolute value for our experiment. We then analyzed the relative occurrence of each build change type and report the top-10 most frequently occurring change types sorted by their median relative occurrence. Furthermore, we analyzed the number of build changes per build change type category and report their relative frequencies.

6.1.2 Results

Figure 2 shows the boxplots of the top-10 most frequently occurring build change types sorted by median relative frequency.

We study the top-10 most frequent change types sorted by the median. Figure 2 shows the boxplots of the top-10 change types. We see that PARENT_POSTFIX_VERSION_-UPDATE is the most frequent change type with a median relative frequency of 0.056. Furthermore, we observe that the second most frequently occurring change type is PARENT_POSTFIX_VERSION_UPDATE with a median of 0.045. Ranked third and fourth, we see DEPENDENCY_INSERT and GENERAL_PROPERTY_UPDATE with medians of 0.031 and 0.028, respectively.

Analyzing the change types, we see that six changes are of general purpose, two change types that modify dependency declarations, and two change types that concern the build

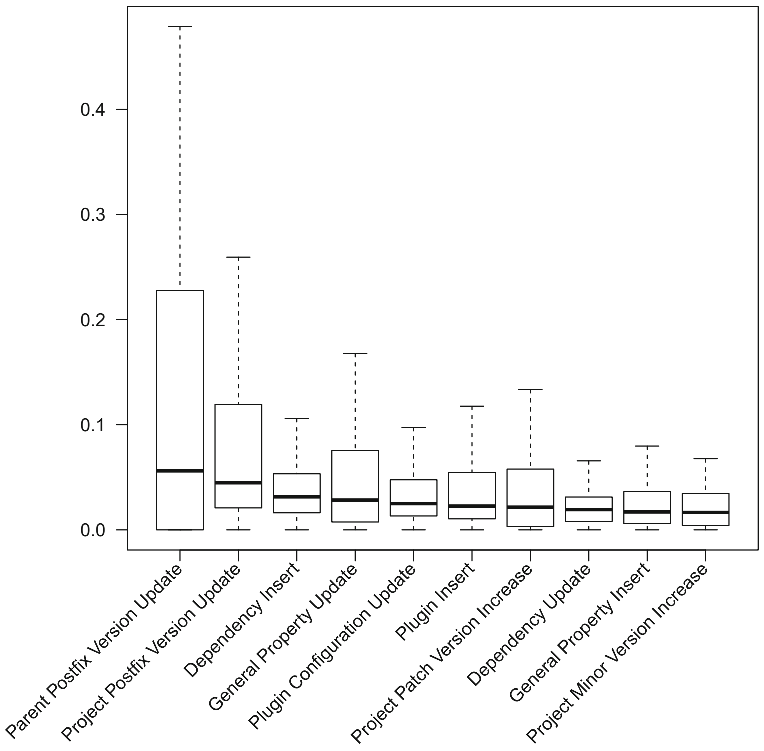


Fig. 2 Boxplots of the relative build change frequency for the top-10 most frequent change types (sorted by **median**)

parts of the build specification. We observe that the top-10 most frequently occurring build change types sorted by median account for 51% of all of the changes.

Given this analysis, we conclude that the most frequent changes concern updates of the version of the project itself and the respective parent project reference. Furthermore, we find that the numbers indicate that the dependency management system, which is a core part of the build system, is changed frequently and that plugins, which are another core part of the build system, are also changed frequently. In particular, new plugins are inserted, and the configuration of plugins are frequently changed. These findings substantiate the importance of these two parts of a build system. The observations above confirm the findings of the original work (Macho et al. 2017). Although we significantly split up the version updating changes, the top-10 changes still account for 51% of the total build changes.

The next step of the analysis deals with the frequencies of build changes per change category that we have defined in Section 4.1. Figure 3 shows the relative frequencies of the build changes per build category sorted by the median. We observe that the `General Changes` category accounts for 0.62 (62%) of all changes on average. We argue that this ratio is as expected because changes to the properties, parent changes, and changes to the project metadata, such as project version, are aggregated in this category. However, compared to the original study (64%) the expansion of the projects under study lowered the ratio of this change category.

Furthermore, we can see that `Dependency Changes` are the second most frequent change category (0.17). This is in line with the observations of the single change types. As mentioned above, the dependency management system is a core part of the build system and is frequently updated. The third most frequently occurring category contains the changes to the `Build Changes` category (0.12). Lastly, changes to the `Repository Changes` and to the `Team Changes` are rare (0.01 and 0.002, respectively).

Compared to the results of our original study, we observe that changes to the `Build Changes` category occur more frequently that in the original study (0.11)

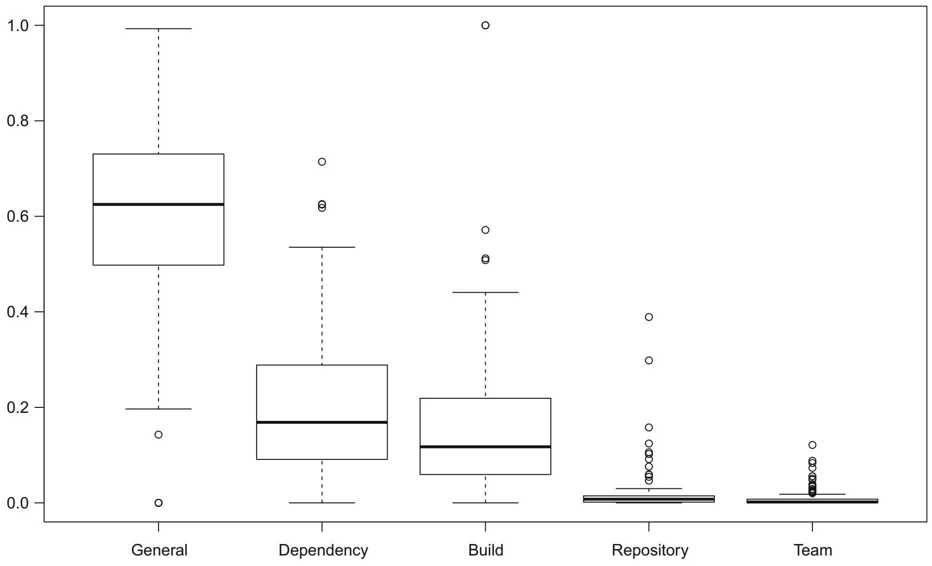


Fig. 3 Boxplots of the relative frequency of build changes per change type category (sorted by median)

and `Dependency Changes` occur less frequently (0.24). However, `Repository Changes` and `Team Changes` are still rare (originally 0.008 and 0.004, respectively).

To acknowledge that projects are different, for example in the size and vendor, we also analyzed the relative frequency of change types and change categories respecting these properties. We analyzed the projects with respect to the properties of the projects depicted in Table 1, namely the number of build changes (BC), the ratio of build changes per commit (BCPC), the number of commits (#Commits), number of releases (#R), and vendor (text before the '/' in column Name). For each aspect, we split the data by the quartiles and investigated whether the relative number of build changes differs between these four segments of the data. For brevity reasons, we only describe the general observations but the detailed results of this investigation can be found online.[10]

We made the following three observations:

– In the lower half of the distributions, the change types concerning parent versions are not in the top-10 most frequent build change types whereas they are at the top in the other half. This observation can be underlined with the fact that smaller projects often do not use the parent mechanism of Maven and hence, do not change the parent reference.
– Projects with a high amount of build changes, build changes per commit, high number of commits, and high number of releases, tend to have more changes of the `General Changes` category and fewer changes of the `Build Changes` category.
– Comparing vendor specific effects, we found that Google projects show a lower number of changes of the `General Changes` category but tend to have more changes of the categories of the categories `Build Changes` and `Dependency Changes`. Furthermore, Apache projects tend to have more changes of the `General Changes` category compared to the other vendors.

6.1.3 Discussion

In this research question, we use BuildDiff to study which build changes are frequently performed. While we aim at identifying build change types that generally occur frequently, we want to highlight that projects can have very different strategies to configure the build system and hence, the frequently performed changes can vary between different projects. However, we still found that particular changes are frequently performed in the majority of the projects that we studied. For example, we found that changes to the parent and project version are frequently performed. This implies that developers make heavy use of the versioning of modules offered by Maven. While these changes are relevant for the versioning of the projects, the changes usually do not have a large impact on the build result because they do not affect the critical parts of build specifications, such as the build procedure itself and the dependency management system. However, the changes to these parts were also ranked among the top-10. For example, we found that changes to the plugin configuration are the fifth most frequent change type. We consider this observation as critical because such changes have a direct impact on the build procedure by modifying how plugins work and hence, misconfigurations can easily break the build. In fact, 28.45% of the commits that changed the plugin configuration and 19.23% that changed the dependency declaration part describe a corrective aspect (Hattori and Lanza 2008) in their commit message. We consider this a large number and hence, we see them as critical change types and conclude

[10] https://zenodo.org/record/4153674#.X5rNXlNKhTY

that these changes need to be carefully applied as they are often performed to repair bugs. For example, the commit ed3ba07[11] of the Google Guava project shows a fix that broke the Java documentation due to a misconfigured plugin configuration. Correcting the configuration helped to avoid this issue. Concerning the dependency changes, the commits cf91012[12] of the jooby-project/jooby and 7e8d672[13] of saucelabs/sauce-java depict examples where the build failed due to missing dependencies. In these commits, the developers had to add another dependency to make the build successful again. These three examples underline that changes to the build and the dependency part of a build specification are crucial for the success of the build.

With these results, we can answer research question RQ1.1, *What are the most frequently occurring build change types?*:

The top-10 most frequently build change types account for 51% of the number of all build changes. Among this top-10, we find version changes and dependency changes frequently. PARENT_POSTFIX_VERSION_UPDATE, PROJECT_POSTFIX_VERSION_UPDATE, and DEPENDENCY_INSERT, are among the top-10 most frequently occurring build change types. The most frequently occurring build change category is `General Changes` **followed by** `Dependency Changes`, **and** `Build Changes`.

6.2 Build changes patterns (RQ1.2)

In this section, we investigate the build change frequencies in terms of build change patterns. We study the build changes and aim at finding frequently occurring build change patterns that occur. First, we describe the approach that we use to answer RQ1.2: *What are the most frequently occurring build change patterns?* Second, we present the patterns that we found in the data set and verify their validity in a qualitative study. Lastly, we discuss the build change patterns that we found.

6.2.1 Approach

The aim of this research question is to identify change patterns in build specifications, *i.e.* which build change types are frequently performed together. We use the terms *build change pattern* and *co-occurring build changes* as synonyms in this work, meaning two or more build change types that were performed in the same commit. For this research question, we use the data of build change type frequencies which are stored in the ChangeDB.

Identifying frequently occurring patterns is a well-known problem and the existing algorithms for such a problem have been used in many previous studies (Espinha et al. 2015; Livshits and Zimmermann 2005; Marsavina et al. 2014; Zimmermann et al. 2005). We use the well-known frequent item set mining and association rule mining techniques (Agrawal et al. 1993) to investigate which build change types occur frequently together with other build change types. In particular, we use the apriori algorithm (Agrawal et al. 1994) to detect frequent build change type patterns. The general idea of the apriori algorithm follows a market basket metaphor. A so called transaction represents a single market basket

[11] https://github.com/google/guava/commit/ed3ba0728dc18d58e2cb43c6308dc30b31cfc2f5

[12] https://github.com/jooby-project/jooby/commit/cf9101226c02fc8f0cea8a0c49f02337bb7d15e4

[13] https://github.com/saucelabs/sauce-java/commit/7e8d672b6b07b4a4515e4d9efbac8f30f5a347f8

and the contained items. All transactions are used to find sub sets of items that frequently occur together in these transactions (*i.e.* frequent item sets). In our case, a transaction refers to a single commit, the items refer to build changes performed in a commit, and frequent item sets refer to build change patterns. In general, a build change pattern is defined as an implication of the form $A \rightarrow B$ where A is a set of unique build change types also named left hand side (LHS) or antecedent, and B is a single build change type that does not appear in A also named right hand side (RHS) or consequent. For example, a build change pattern could be MANAGED_DEPENDENCY_INSERT → DEPENDENCY_INSERT and would be interpreted that if a MANAGED_DEPENDENCY_INSERT was performed, a DEPENDENCY_INSERT was also performed.

The quality of build change patterns is evaluated with metrics that are called interestingness measures. In this study, we use the well-known interestingness measures support and confidence. These interestingness measures have also been used in prior studies (Lubsen et al. 2009; Marsavina et al. 2014). According to Le and Lo (2015), support and confidence are not always the best choice as interestingness measures for association rules. They recommend to include other, less frequently used interestingness measures. We decided to follow this recommendation and also measure lift, odds-ratio, and leverage for our experiments. In the following, we explain each of the used interestingness measures in detail. Please, note that we give simple examples with only one item per set to increase the comprehensibility, but in general each set can consist of one or more items.

Support Support (Agrawal et al. 1993) denotes the relative frequency of the frequent item set with respect to the whole data set. In our case, the support interestingness measure denotes the number of commits in which a build change pattern could be found with respect to the number of total commits. It ranges between 0 and 1, with 0 indicating that this pattern is never found in the data set and 1 indicating that the build change pattern is found in each of the commits. For example, assuming a build change pattern {MODULE_INSERT => CONTRIBUTOR_INSERT} occurring in 35 commits and a total number of commits of 100, the support for this pattern is calculated with $35/100 = 0.35$.

Confidence Confidence (Agrawal et al. 1993) denotes the relative frequency of a consequent in baskets that contain the antecedent. It ranges between 0 and 1, with 1 indicating that the pattern has always been found with the respective consequent. For example, again assuming a build change pattern {MODULE_INSERT => CONTRIBUTOR_INSERT} with the support of 0.35 and a support (relative frequency) of 0.70 for the MODULE_INSERT build change, the confidence is $0.35/0.70 = 0.50$. This means that in half of the cases where MODULE_INSERT could be found the build change pattern could be found as well.

Lift Lift (Brin et al. 1997) describes a ratio of support assuming that the two items are independent. It ranges between 0 and infinity. A lift of 1 indicates that the two items are independently occurring in a transaction. In our case, the lift indicates whether two build changes are independently occurring in a commit. The higher the lift, the more dependent are the two build changes of a frequent pattern. For example, again assuming a build change pattern {MODULE_INSERT => CONTRIBUTOR_INSERT} with the support of 0.35, a support (relative frequency) of 0.70 for the MODULE_INSERT build change, and a support of 0.2 for the CONTRIBUTOR_INSERT build change, the lift is $0.5/(0.7 \times 0.2) = 3.57$.

Odds-ratio The odds-ratio by Tan et al. (2004) denotes the odds of having an item X in transactions that also contain item Y divided by the odds of having an item X in transactions that do not contain item Y. It ranges between 0 and infinity and 1 indicates that Y is not associated with X). In our case, for example, this means that the odds-ratio expresses the ratio between the odds of finding a DEVELOPER_UPDATE change in the commits where also a CONTRIBUTOR_UPDATE change was performed divided by the odds of finding a DEVELOPER_UPDATE change in the commits where no CONTRIBUTOR_UPDATE change performed. An odds-ratio of 1 indicates that build change Y is not associated with build change X.

Leverage Leverage or the Piatetsky-Shapiro measure by Piatetsky-Shapiro (1991) measures "the difference of X and Y appearing together in the data set and what would be expected if X and Y where statistically dependent". It ranges between -1 and 1 and Leverage values close to 0 indicate independent occurrence of items and therefore, uninteresting patterns.

The ChangeDB contains all the change frequencies of all of the build changing commits of the studied projects. However, each frequency is given with integer numbers. Similar to other studies (Marsavina et al. 2014), we map the frequencies to categories because we are not interested in how often a change type occurred in a commit but in the occurrence itself. Hence, we use a binary categorization indicating whether or not a build change type occurs in a commit. To create this data, we iterate over all commits that change the build specification (*i.e.*, contain at least one build change) of the studied projects and create a transaction for each of the commits. Table 5 shows two examples of these transactions. The set of transactions is input into an association rule mining framework. We use the well-known association rule mining framework `arules` by Hahsler et al. (2005) to compute the build change patterns. In particular, we use the apriori algorithm of this framework.

We conduct two experiments that aim to identify frequently occurring build change patterns. First, we aim at identifying ***global change patterns*** that can be found over all projects. We input all transactions of all of the projects into the apriori algorithm and identify change patterns that show a high performance according to our interestingness measures. Second, we investigate ***local change patterns*** that are only found within a project. We input each set of transactions of each of the studied projects separately and extract the change patterns of each project. For both experiments, we set thresholds for the apriori algorithm to avoid the calculation of less frequently occurring change patterns. We set the minimum support to 0.01 and the minimum confidence to 0.2 to improve the speed of the algorithm. Build changing commits are usually small and contain only a few build changes (Macho et al. 2017). We can confirm this finding also for our extended data set: The median number of build changes in build changing commits is 2, and the values for the 25% and 75% quartiles are 1 and 5, respectively. Hence, we only study build change patterns that are small *i.e.* having exactly a single build change on the left hand side (LHS) of a change pattern and one build change on the right hand side (RHS), for example $A \rightarrow B$ which means if A occurs

Table 5 Transaction examples

TransID	Representation
1	PLUGIN_INSERT, DEPENDENCY_DELETE
2	PLUGIN_CONFIGURATION_UPDATE, PROJECT_NAME_UPDATE

then also B occurs. Furthermore, larger change patterns are likely to contain smaller change patterns and hence, do not add to the knowledge. For example if a change pattern $A \rightarrow B$ has been found with high support and confidence, it is also likely to find a change pattern such as $A, X \rightarrow B$.

6.2.2 Results

First, we present the ***global change patterns*** that we could identify. Using all of the projects at once, we could find 46 change patterns. Table 6 shows the top-20 ***global change patterns*** sorted by confidence. As mentioned above, we also use other interestingness measures than support and confidence. However, we observed similar rankings for each of the recommended interestingness measures.

With respect to the confidence, change patterns 1-4, 8, and 9 concern the modification of the source code management (SCM) data. Whenever one of the three parts is modified, the other two need to be modified too. This is an expected finding because the SCM data contains URLs which all need to be modified in the same way to stay consistent. These change patterns also show a very high lift between 27 and 29. Furthermore, we see that change patterns 5-7 and 11-13 concern version changes of the projects' version and the parent projects' version. The change patterns indicate that the two are usually modified together which is backed by the high confidence of the change patterns. Furthermore, we can see that the modified part of the version declaration, *i.e.*, major, minor, and postfix, is the same within a change pattern, meaning that if the LHS modifies the major part of the version number, so does the RHS. This suggests that the version declarations of the project and its parent are usually modified at the same place *e.g.* if the projects major version increases then also the major version of the parent project declaration increases. We also observe this behavior for the patch part but with lower confidence. Change pattern 10 concerns insertions of dependency management declarations and indicates that they co-occur with dependency insert. This is the case, if a project adds a new dependency to a third party library for

Table 6 Global build change patterns with confidence > 0.5 sorted by confidence

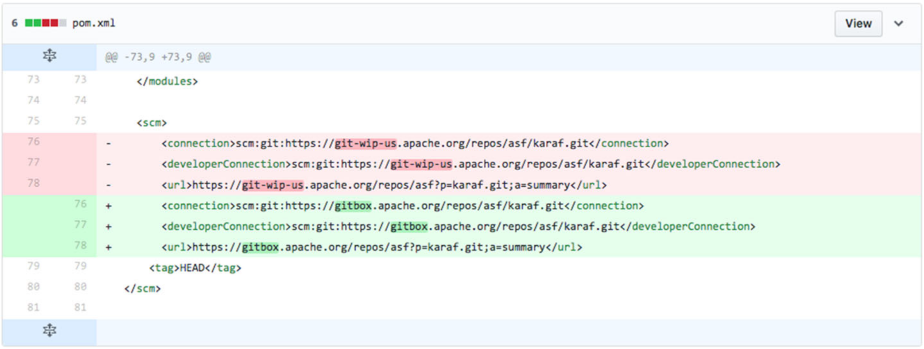
ID	lhs		rhs	sup	conf	lift
1	scm_devconnection_update	=>	scm_connection_update	0.03	0.98	28.50
2	scm_connection_update	=>	scm_url_update	0.03	0.95	27.71
3	scm_url_update	=>	scm_connection_update	0.03	0.95	27.71
4	scm_devconnection_update	=>	scm_url_update	0.03	0.95	27.61
5	parent_postfix_version_update	=>	project_postfix_version_update	0.07	0.91	7.81
6	parent_patch_version_increase	=>	project_patch_version_increase	0.03	0.88	15.99
7	parent_minor_version_increase	=>	project_minor_version_increase	0.02	0.81	19.52
8	scm_connection_update	=>	scm_devconnection_update	0.03	0.75	28.50
9	scm_url_update	=>	scm_devconnection_update	0.03	0.73	27.61
10	managed_dependency_insert	=>	dependency_insert	0.03	0.69	4.48
11	project_patch_version_increase	=>	parent_patch_version_increase	0.03	0.63	15.99
12	project_postfix_version_update	=>	parent_postfix_version_update	0.07	0.61	7.81
13	project_minor_version_increase	=>	parent_minor_version_increase	0.02	0.59	19.52

example, and the project uses the dependency management mechanism of Apache Maven. This mechanism ensures that all modules of a project will use the third party library as it was declared and configured in the dependency management part. Adding a new dependency in a module then requires a corresponding declaration in the dependency management part.

As stated above, we also investigate the change patterns for each project separately to identify ***local change patterns***. The results of this experiment are depicted in Table 7 which shows the top-20 ***local change patterns*** sorted by the number of projects in which they were observed. We mostly find similar change patterns compared to the global analysis. For example the global change patterns concerning the modification of the SCM part are also observed in many projects (local change patterns 13, 14, 16, 17, and 20) . The updating of the versions in projects' and the respective parents projects' declaration can be seen in change patterns 1-2, and 4-7 . Finally, the required insertion of a managed dependency when inserting a new dependency is also observed in change pattern 15 and 19 . However, we could also identify other frequently occurring build change patterns. For example, change patterns 11 and 18 show the relation of inserting a new dependency and inserting a new property. In our recent work (Macho et al. 2018), we found that version declarations of dependencies are often encapsulated in properties. Hence, we argue that this rule makes sense because if developers insert a new dependency, they also need to insert the new property which holds the version declaration of the dependency. For change patterns 3,8,15,16, and especially 12, we could not find an interpretation. However, we investigated such cases and found that the commits which contain such change patterns usually change a lot in the

Table 7 Top-20 local build change patterns sorted by their occurrence count

ID	lhs		rhs	sup	conf	lift	count
1	parent_postfix_version_update	=>	project_postfix_version_update	0.12	0.96	10.35	78 (54.17%)
2	project_postfix_version_update	=>	parent_postfix_version_update	0.13	0.85	10.90	73 (50.69%)
3	dependency_update	=>	dependency_insert	0.04	0.52	4.84	66 (45.83%)
4	parent_patch_version_increase	=>	project_patch_version_increase	0.08	0.94	17.80	65 (45.14%)
5	parent_minor_version_increase	=>	project_minor_version_increase	0.05	0.89	24.54	63 (43.75%)
6	project_minor_version_increase	=>	parent_minor_version_increase	0.05	0.82	24.54	63 (43.75%)
7	project_patch_version_increase	=>	parent_patch_version_increase	0.09	0.86	18.27	63 (43.75%)
8	dependency_insert	=>	dependency_update	0.04	0.33	5.31	57 (39.58%)
9	dependency_delete	=>	dependency_update	0.03	0.39	5.04	50 (34.72%)
10	dependency_update	=>	dependency_delete	0.03	0.30	5.04	50 (34.72%)
11	general_property_insert	=>	dependency_insert	0.04	0.56	6.59	49 (34.03%)
12	dependency_delete	=>	dependency_insert	0.04	0.54	4.90	45 (31.25%)
13	scm_connection_update	=>	scm_url_update	0.09	0.93	34.98	45 (31.25%)
14	scm_url_update	=>	scm_connection_update	0.09	0.95	34.98	45 (31.25%)
15	managed_dependency_insert	=>	dependency_insert	0.06	0.73	5.25	44 (30.56%)
16	scm_connection_update	=>	scm_devconnection_update	0.07	0.97	39.08	44 (30.56%)
17	scm_devconnection_update	=>	scm_connection_update	0.07	0.99	39.08	44 (30.56%)
18	dependency_insert	=>	general_property_insert	0.05	0.46	7.49	41 (28.47%)
19	dependency_insert	=>	managed_dependency_insert	0.06	0.39	5.45	41 (28.47%)
20	scm_devconnection_update	=>	scm_url_update	0.08	0.93	33.23	40 (27.78%)



```
@@ -73,9 +73,9 @@
      </modules>

      <scm>
-         <connection>scm:git:https://git-wip-us.apache.org/repos/asf/karaf.git</connection>
-         <developerConnection>scm:git:https://git-wip-us.apache.org/repos/asf/karaf.git</developerConnection>
-         <url>https://git-wip-us.apache.org/repos/asf?p=karaf.git;a=summary</url>
+         <connection>scm:git:https://gitbox.apache.org/repos/asf/karaf.git</connection>
+         <developerConnection>scm:git:https://gitbox.apache.org/repos/asf/karaf.git</developerConnection>
+         <url>https://gitbox.apache.org/repos/asf?p=karaf.git;a=summary</url>
        <tag>HEAD</tag>
    </scm>

```

Fig. 4 An example of the SCM change pattern in the Apache Karaf project

build specification.[14] Hence, we argue that these patterns are only appear because of the statistical impact of such large commits.

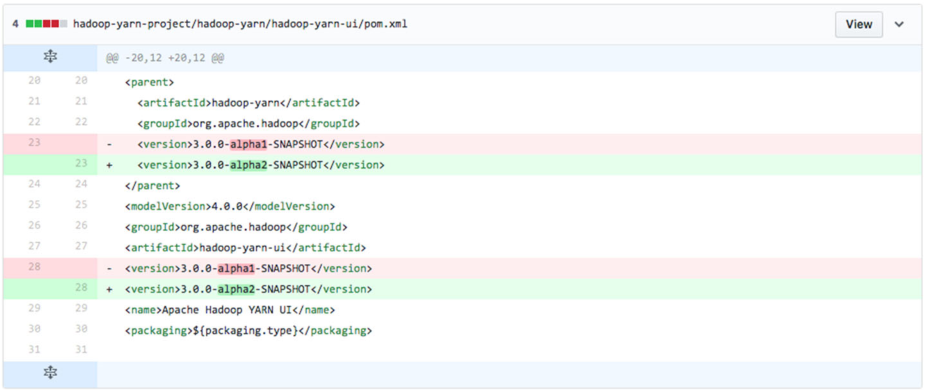
In the following, we give some examples of the change patterns that we identified above to substantiate our quantitative findings.

In our experiments, we discovered that the three parts of the SCM system declaration are usually changed together. An example of these change patterns is shown in Figure 4. It shows the changes of a commit[15] of the Apache Karaf project. In this commit, the author updates all three parts of the SCM part in the MAVEN build specification. We concluded that missing one of the three changes can lead to errors, for example, a release can fail if the SCM information is incomplete or inconsistent.

Other change patterns that we identifed concern updates to the version of the project and the parent project declaration. We observed that these version numbers are often changed together. Moreover, the co-changes usually concern the same part of the version. For example, if the major part of the project's version number is increased, the major part of the parent project's version is also increased and vice versa. The same behaviour can be observed for the other parts of a version number, such as the patch part and the minor version part. This indicates that the project's version and the respective parent project's version are frequently changed together. Furthermore, the part of the version that is changed is usually the same. Figure 5 depicts the co-occurring changes to the parent and project version in a commit[16] of the Apache Hadoop project. Moreover, we see that the same parts of the version are changed.

Another observation is that the insertion managed dependencies implies an insertion of a corresponding dependency declarations. In Fig. 6, two changes are recorded in a commit of the project Alluxio.[17] We see that a dependency declaration is added to the dependency management part, and the same dependency is added to the dependency part as well.

[14]Commits that contain such a pattern show statistically significantly more build changes than the whole population. Wilcoxon test shows $p < 0.01$ and Cliff's Delta yields small.
[15]https://github.com/apache/karaf/commit/007857025478f2e808aad5cf4ce3b3e4bbe7d10d
[16]https://github.com/apache/hadoop/commit/a53e981adfaf47b64b135978c7ad9eb7ed6b4f8e
[17]https://github.com/Alluxio/alluxio/commit/e45975a5b0bedc0e8b410b231a687c069a06956c

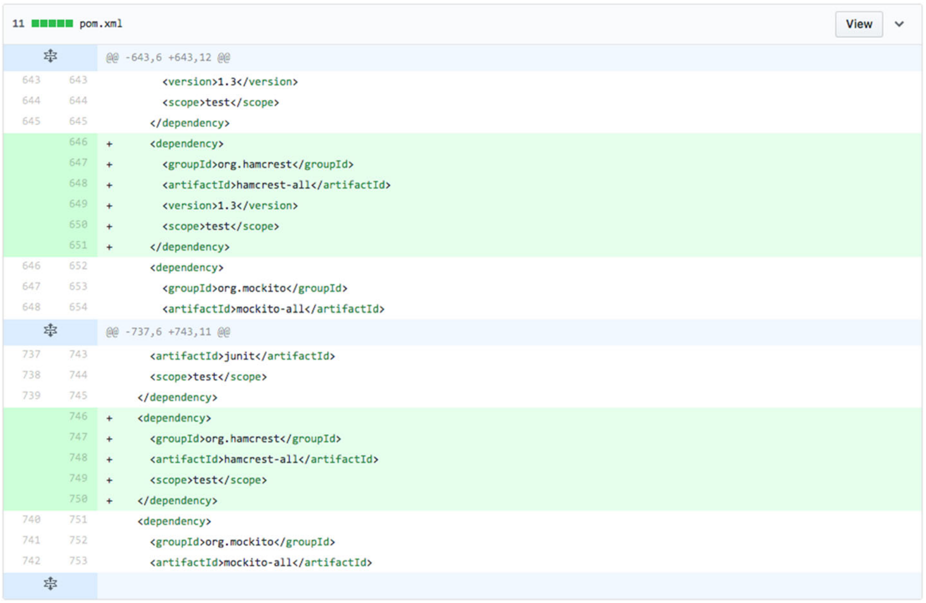


```
4 hadoop-yarn-project/hadoop-yarn/hadoop-yarn-ui/pom.xml                    View
@@ -20,12 +20,12 @@
    <parent>
      <artifactId>hadoop-yarn</artifactId>
      <groupId>org.apache.hadoop</groupId>
-     <version>3.0.0-alpha1-SNAPSHOT</version>
+     <version>3.0.0-alpha2-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-yarn-ui</artifactId>
-   <version>3.0.0-alpha1-SNAPSHOT</version>
+   <version>3.0.0-alpha2-SNAPSHOT</version>
    <name>Apache Hadoop YARN UI</name>
    <packaging>${packaging.type}</packaging>
```

Fig. 5 An example of the version change pattern in the Apache Hadoop project

6.2.3 Discussion

Patterns are an often studied problem (Espinha et al. 2015; Livshits and Zimmermann 2005; Marsavina et al. 2014; Zimmermann et al. 2005). In this study, we focused on patterns that occur in single commits. We are aware that we only cover on particular type of change patterns, however, these patterns are the most studied in recent work and also interesting from a industry perspective. Hence, for this work, we do not study patterns that occur over time,



```
11 pom.xml                                                                  View
@@ -643,6 +643,12 @@
          <version>1.3</version>
          <scope>test</scope>
        </dependency>
+       <dependency>
+         <groupId>org.hamcrest</groupId>
+         <artifactId>hamcrest-all</artifactId>
+         <version>1.3</version>
+         <scope>test</scope>
+       </dependency>
        <dependency>
          <groupId>org.mockito</groupId>
          <artifactId>mockito-all</artifactId>
@@ -737,6 +743,11 @@
        <artifactId>junit</artifactId>
        <scope>test</scope>
      </dependency>
+     <dependency>
+       <groupId>org.hamcrest</groupId>
+       <artifactId>hamcrest-all</artifactId>
+       <scope>test</scope>
+     </dependency>
      <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-all</artifactId>
```

Fig. 6 An example of the dependency change pattern in the Alluxio project

such as sequence patterns (Zaki 2001) but focus on patterns that occur in a single development step. For example, we found that also changes to the build specification files yield patterns of such changes. One pattern concerns the simultaneous update of the project version number and the project's parent version number. This often happens when developers are using a tool to release the software. Analyzing this effect, we found that 60.60% of the studied commits that show this pattern were performed by a tool. While this seems to be a large number, 39.40% of the commits were still performed by a developer. We argue that the pattern is important and can help developers when releasing their projects without a tool. Analyzing the messages of the commits that were performed manually by developers, we found that nearly all of them were performed while releasing the project. Hence, missing one of the two changes might lead to inconsistent versioning, which is a huge problem especially when the project is used by third parties, or can even lead to failing the build if the respective version is incompatible with the new one. To that extent, we highlight that although this pattern might be obvious for experienced Maven users it is important to check and validate its correct application to avoid problems. Another pattern that we found states that dependency changes often co-occur with corresponding changes to the dependency management part. While there is an increasing interest in research about dependencies and their management (Benelallam et al. 2019; Kula et al. 2018; Soto-Valero et al. 2019; Soto-Valero et al. 2020), to the best of our knowledge, there is no dedicated tooling that helps in organizing dependencies. Our results contribute towards a preliminary tooling that can check and validate best practices for using Maven. Especially in the early phases of a project in which the developers might not use a multi-module project because of its additional complexity, it is important to clearly manage the dependencies of the project. We believe that, for example our empirically confirmed change pattern about dependency changes, will encourage developers to consistently change dependencies starting from the first commit.

With these results, we can answer RQ1.2, *What are the most frequently occurring build change patterns?*, as follows:

The three parts of the SCM part of a MAVEN build specification are frequently changed together. Furthermore, version numbers of the parent project and the project itself usually change together and the modified part of the version number is usually the same. Modifying a dependency in the dependency management part often co-occurs with a corresponding modification of the dependency declaration.

7 When are the changes recorded (RQ2)

The second research question deals with the occurrence of build changes over the projects' life time. We aim at identifying special time periods, such as the time around releases, that contain a significantly high amount of build changes and build change categories. The hypothesis that we confirmed in our prior work (Macho et al. 2017) states that build changes are not equally distributed over the projects timeline, but clustered around release days.

Knowing when build changes usually happen can have two possible benefits to developers. First, using this knowledge to indicate that changes that are usually applied to the build specification are still missing can warn developers that they might miss an important change. These hints can help in avoiding build breakage due to missing build changes. Approaches, such as our prior work on missing build changes (Macho et al. 2016) can incorporate this information to improve the predictive power. Second, some changes to the build specification are considered critical. For instance, if a configuration of a plugin, such as the compiler plugin, is changed directly before a planned release, it can impact the release procedure and possibly lead to build breakage. For example, if the used Java version is changed before a release, it is obvious that this change can have a huge impact on the buildability of the project. Based on this motivation, we set out to answer RQ2: *When are build changes recorded?* We split this research question into two sub research questions.

RQ2.1 *How are build changes distributed over time?*
This research question has already been addressed in our previous work (Macho et al. 2017). In this paper, we extend it by including more projects in the study to increase the generalizability of the results and explore the differences to the original work. The aim of this research question is to confirm that build changes are usually clustered around releases. This knowledge can improve developers' awareness of when and how to modify their build specification.

RQ2.2 *Which build changes happen before, during, and after releases?*
In this research question, we investigate frequently occurring co-change patterns within build changes. We explore which build changes frequently occur with other build changes and find patterns in this co-occurrence. We aim at identifying frequent patterns to give developer hints for possibly missing co-changes.

We first present the replication and extension of the prior study in RQ2.1 and show that the results hold and can be generalized to more projects. Then, we refine the investigation and study the build change categories and build change types that occur around releases. We aim at gaining knowledge on which changes are frequently performed before, during, and after releases in RQ2.2 which refines RQ1.1. In this paper, we use the terms "release" and "release day" synonymously and refer to the day on which the release was made. For both research questions, we first present the approach that we use in the experiment and then we show the respective results.

7.1 When build changes happen (RQ2.1)

In this section, we investigate when the build changes occur. We suppose that build changes are not equally distributed over the project, but have phases in which they occur significantly more frequently than in other phases. Hence, we used the build change data that we extracted using BUILDDIFF to check whether our hypothesis holds and answer RQ2.1: *How are build changes distributed over time?*.

7.1.1 Approach

We started with the aggregated change data that we created in Section 6. This data contains for each commit of a project the number of changes per change type that have been performed in the commit. For this research question, we added the date on which the commit was performed and summed up all build changes to a single value per day, *i.e.* one row of our data set contains the ID of the commit, the number of build changes that were performed in that commit, and the date of the commit. For example, if exactly two commits were made on 23th June 2016 with 10 and 15 build changes, respectively, we created a single data point with 25 build changes. Based on this information, we investigated the data in two ways, as a single day value and with a sliding window approach.

The first investigation treats each day as a single data point, and hence, adds the number of build changes of commits that were made on the same day. The second investigation uses this data and applies a sliding window approach, similar to the approach of Maarek et al.

(1991). We summed the number of build changes of k days to increase the context of the build changes.

As our hypothesis for this research question states, we suppose that build changes are not equally distributed over the project, but occur more frequently in some time periods of the project. We further suppose that one special period in the project that shows a significantly higher amount of build changes, is the time around releases. Thus, we extracted the release data of each of the studied projects provided by the GitHub API. In particular, we extracted the commit ID of the release, the day of the release, and its name. Column #R of Table 1 shows the number of releases per project that we could extract. We could not extract release information for all of the projects. Hence, we excluded 11 projects, such as `CoreNLP`, for which no release data was available and performed the experiment with the remaining 223 projects.

To substantiate our claim, we will show that days that are close to a release contain statistically significantly more build changes than days that are not close to a release. To that extent, we consider a day to be close to a release if it is in between t days around the release. Compared to the original work (Macho et al. 2017), we improved the calculation of t. Previously, t was considered a fixed value for each of the projects and releases. However, a fixed value suffers from two major problems. First, releases are sometimes performed in different intervals. Hence, using a fixed t can include too many or too few days for the release. We improved the calculation by considering the distance between consecutive releases. The value for t is selected individually for each release by using a percentage p of the smaller of the distances to the previous and the next release. For p, we use two values *i.e.* 10% and 25%.

Furthermore, the analysis uses two approaches, namely a single day approach that counts the number of build changes per day, and a sliding window approach that extends the context of the build changes. For the analysis on a daily basis, we consider $k = 1$ and for the sliding window approach, we consider $k \in \{3, 5, 7\}$ days, with k being the number of days aggregated. We choose different values for k to investigate the influence of the size of the window on the results. Similar to the original work, we performed the experiment with all of the k values and the results were similar. Thus, we only present results for $k = 1$ and $k = 7$ in the paper.

Next, we checked if the two distributions (*i.e.* number of build changes on days that are near a release day and not near a release day) are significantly different. We checked this with a Mann-Whitney-Wilcoxon test ($\alpha < 0.01$) and calculated the effect size d using Cliff's Delta (Cliff 1993). We used Mann-Whitney-Wilcoxon and Cliff's Delta since the number of build changes is non-normal distributed. The effect size is considered negligible for $d < 0.147$, small for $0.147 \leq d < 0.33$, medium for $0.33 \leq d < 0.47$, and large for $d \geq 0.47$ (Grissom and Kim 2005).

7.1.2 Results

Figures 7 and 8 show the distribution of build changes of the project `spring-roo` with different sliding windows of sizes $k = 1$ and $k = 7$, respectively. The black line depicts the number of build changes that were performed on the respective day. Each vertical red line indicates a release. We can see that most of the peaks of the black lines (number of build changes) appear close to a vertical red line (release). This suggests that our hypothesis is correct.

Furthermore, looking at the distribution of the number of build changes in days near release days and comparing it with the distribution of build changes in days that are far from

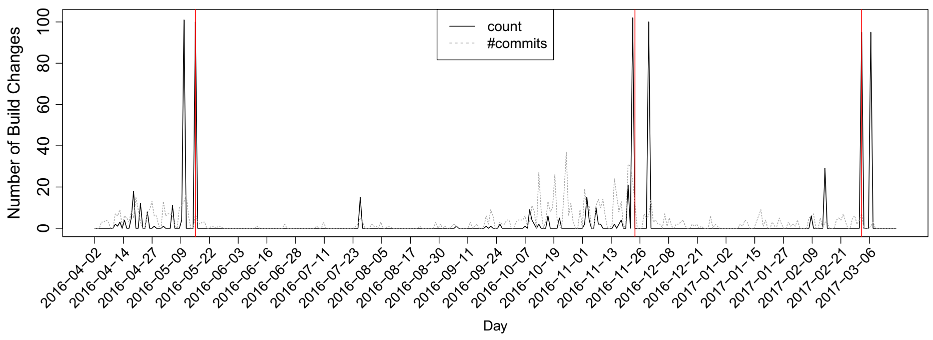


Fig. 7 Excerpt of the number of build changes over time in the `spring-roo` project using a single day approach ($k = 1$). Releases are depicted as vertical red lines

release days, we can see that the distributions appear to be different. Figure 9 shows the boxplots for both approaches (*i.e.*, single day and sliding window), both distributions (*i.e.*, number of build changes near release days and not near release days), and both values of the release-near threshold (*i.e.*, t calculated with $p \in \{10\%, 25\%\}$). We see that in all four subfigures of Fig. 9, the boxplot of the number of build changes that are near releases (left) seems to contain significantly higher numbers than the boxplot of days far from releases (right). Furthermore, we see that the threshold value t has no visible impact on the distribution whereas the window size k increases the build change counts but does not visibly change the structure of the boxplot.

To substantiate these results, we perform a Mann-Whitney-Wilcoxon test on the distributions and measure the p-value and Cliff's Delta to show that there is a statistically significant difference between release-near and not release-near number of build changes for each of the studied projects. Table 8 presents the p-value (p) of the Mann-Whitney-Wilcoxon test and Cliff's Delta d per each approach in detail for the projects of the original study (Macho et al. 2017). The project `h2o-2` that we studied in our previous work (Macho et al. 2017) has no commits in 2019 and was therefore not considered in this study. The full table can be found online.[18] For 5 projects, we did not calculate these values because each project

[18] https://zenodo.org/record/4153674#.X5rNXlNKhTY

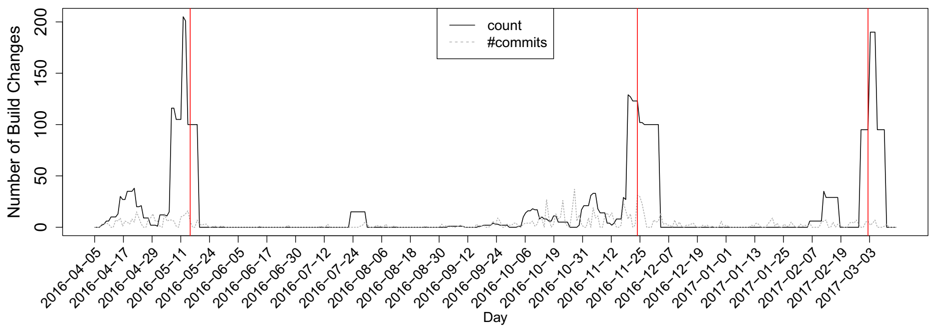


Fig. 8 Excerpt of the number of build changes over time in the `spring-roo` project using a sliding window ($k = 7$). Releases are depicted as vertical red lines

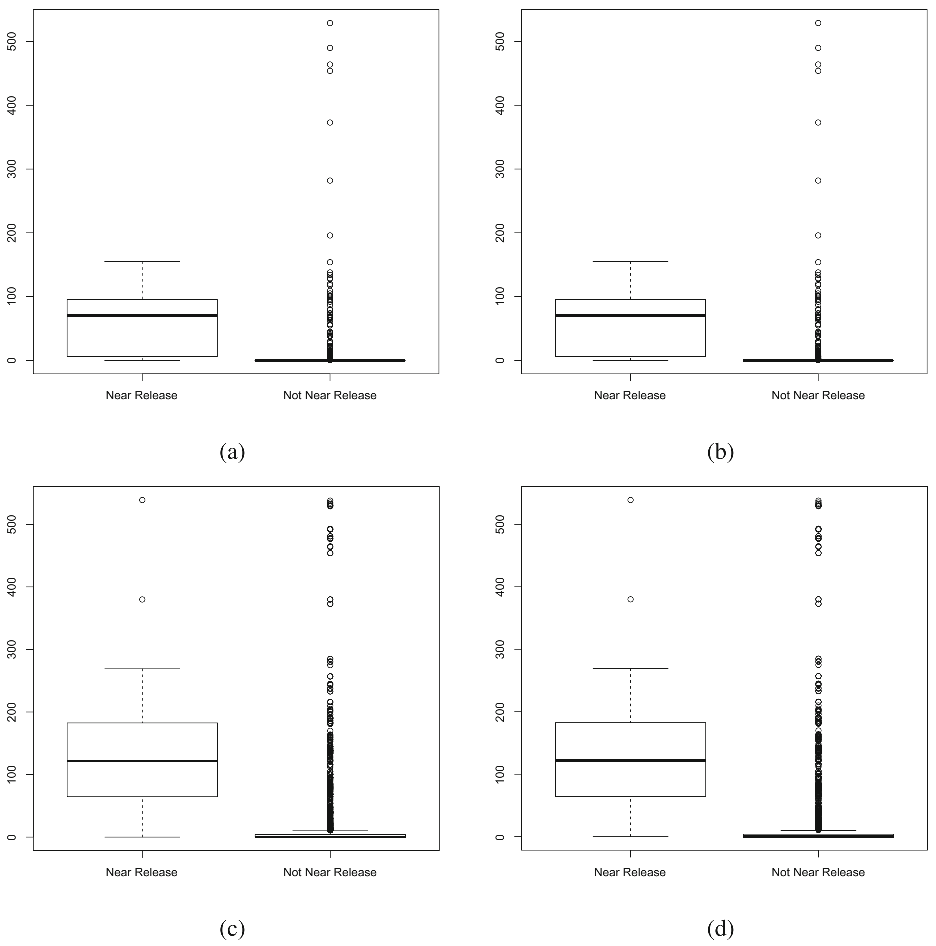


Fig. 9 Boxplots showing the distributions of the counts of near release and non-near release build changes of `spring-roo` as computed with the single day approach and sliding window approach using $k = 1$ and $k = 7$, and different values for the near-release threshold, *i.e.* $t = 10$ and $t = 25$

contained less than ten build changes or did not have any public releases. The p-values show that the frequency of build changes near and not near a release differ significantly (all $p < 0.01$) except for the project `CoreNLP`.

The effect size can be considered large in all projects except for 5 projects: `hadoop` and `hazelcast` having medium effect size, `hbase` and `CoreNLP` having low effect size, and `jenkins` having negligible effect size. We find that these lower effect sizes are caused by the release information. For example, `hadoop` shows a dense release plan in the beginning of the data and this can influence the sliding window approach.

Although this study yields promising insights into the distribution of build changes, we also need to take into account that the number of build-changing commits per day is not normally distributed over time. This means that on some days more build changing commits are performed and, moreover, we know that especially shortly before releases the number of commits increases. To address this, we conducted an additional experiment and normalized

Table 8 Results of the Mann-Whitney-Wilcoxon test (p: p-value) and Cliffs Delta d of the distributions of the number of build changes near and non-near releases using the single day approach and sliding window approach with $k = 7$

Project	Threshold ($t = 10$)				Threshold ($t = 25$)			
	Single Day ($k = 1$)		Sliding ($k = 7$)		Single Day ($k = 1$)		Sliding ($k = 7$)	
	p	d	p	d	p	d	p	d
Alluxio/alluxio	$2.98e^{-37}$	0.77	$9.50e^{-17}$	0.67	$2.98e^{-37}$	0.77	$9.50e^{-17}$	0.67
apache/activemq	$2.69e^{-49}$	0.83	$1.89e^{-25}$	0.82	$2.69e^{-49}$	0.83	$1.89e^{-25}$	0.82
apache/camel	$5.86e^{-63}$	0.86	$7.01e^{-53}$	0.80	$5.86e^{-63}$	0.86	$7.01e^{-53}$	0.80
apache/flink	$3.23e^{-59}$	0.92	$1.07e^{-34}$	0.84	$3.23e^{-59}$	0.92	$1.07e^{-34}$	0.84
apache/hadoop	$4.73e^{-23}$	0.35	$6.45e^{-22}$	0.40	$4.73e^{-23}$	0.35	$6.45e^{-22}$	0.40
apache/hbase	$8.47e^{-33}$	0.33	$4.30e^{-12}$	0.23	$8.47e^{-33}$	0.33	$4.30e^{-12}$	0.23
apache/karaf	$6.62e^{-53}$	0.88	$1.27e^{-38}$	0.85	$6.62e^{-53}$	0.88	$1.27e^{-38}$	0.85
apache/storm	$6.83e^{-25}$	0.71	$1.70e^{-11}$	0.64	$6.83e^{-25}$	0.71	$1.70e^{-11}$	0.64
apache/wicket	$3.19e^{-111}$	0.72	$5.51e^{-31}$	0.57	$3.19e^{-111}$	0.72	$5.51e^{-31}$	0.57
deeplearning4j/deeplearning4j	$1.67e^{-33}$	0.94	$2.71e^{-23}$	0.86	$1.67e^{-33}$	0.94	$2.71e^{-23}$	0.86
druid-io/druid	$2.38e^{-129}$	0.73	$3.12e^{-82}$	0.72	$2.38e^{-129}$	0.73	$3.12e^{-82}$	0.72
eclipse/jetty.project	$1.15e^{-173}$	0.87	$3.44e^{-82}$	0.71	$1.15e^{-173}$	0.87	$3.44e^{-82}$	0.71
google/closure-compiler	$7.59e^{-143}$	0.50	$1.40e^{-60}$	0.71	$7.59e^{-143}$	0.50	$1.40e^{-60}$	0.71
google/guava	$2.79e^{-160}$	0.80	$5.14e^{-35}$	0.72	$2.79e^{-160}$	0.80	$5.14e^{-35}$	0.72
Graylog2/graylog2-server	$1.09e^{-169}$	0.88	$2.18e^{-36}$	0.56	$1.09e^{-169}$	0.88	$2.18e^{-36}$	0.56
hazelcast/hazelcast	$2.72e^{-60}$	0.50	$1.88e^{-24}$	0.46	$2.72e^{-60}$	0.50	$1.88e^{-24}$	0.46
hibernate/hibernate-search	$6.61e^{-149}$	0.86	$4.23e^{-53}$	0.76	$6.61e^{-149}$	0.86	$4.23e^{-53}$	0.76
jenkinsci/jenkins	$3.02e^{-231}$	0.73	$1.19e^{-08}$	0.14	$3.02e^{-231}$	0.73	$1.19e^{-08}$	0.14
languagetool-org/languagetool	$2.14e^{-59}$	0.95	$3.27e^{-17}$	0.86	$2.14e^{-59}$	0.95	$3.27e^{-17}$	0.86
naver/pinpoint	$7.12e^{-43}$	0.98	$3.62e^{-18}$	0.90	$7.12e^{-43}$	0.98	$3.62e^{-18}$	0.90
neo4j/neo4j	$3.11e^{-59}$	0.51	$1.16e^{-37}$	0.47	$3.11e^{-59}$	0.51	$1.16e^{-37}$	0.47
netty/netty	$1.42e^{-202}$	0.91	$1.28e^{-53}$	0.69	$1.42e^{-202}$	0.91	$1.28e^{-53}$	0.69
orientechnologies/orientdb	$4.96e^{-141}$	0.90	$3.63e^{-45}$	0.69	$4.96e^{-141}$	0.90	$3.63e^{-45}$	0.69
prestodb/presto	$2.95e^{-98}$	0.70	$2.23e^{-39}$	0.52	$2.95e^{-98}$	0.70	$2.23e^{-39}$	0.52
SonarSource/sonarqube	$9.58e^{-103}$	0.80	$3.57e^{-33}$	0.56	$9.58e^{-103}$	0.80	$3.57e^{-33}$	0.56
spring-projects/spring-boot	$1.50e^{-51}$	0.82	$4.44e^{-38}$	0.74	$1.50e^{-51}$	0.82	$4.44e^{-38}$	0.74
spring-projects/spring-roo	$8.17e^{-54}$	0.76	$8.37e^{-26}$	0.85	$8.17e^{-54}$	0.76	$8.37e^{-26}$	0.85
stanfordnlp/CoreNLP	$7.58e^{-01}$	-0.05	$3.40e^{-01}$	0.29	$7.58e^{-01}$	-0.05	$3.40e^{-01}$	0.29
wildfly/wildfly	$1.91e^{-50}$	0.90	$6.82e^{-39}$	0.83	$1.91e^{-50}$	0.90	$6.82e^{-39}$	0.83

the number of build changes on a day with the ratio of build-changing commits on that day. For example, assume a day with ten total commits out of which five were build-changing, and that BUILDDIFF extracted 20 build changes from the five build-changing commits. We normalize the 20 build changes by multiplying with the ratio of build-changing commits ($5/10 = 0.5$) which gives us a normalized value of $20 * 0.5 = 10$ build changes for that day.

We see that some peaks are now flattened because on these days the ratio of build-changing commits is low. However, we still see that most of the high peaks happen around releases. We also investigated the distributions of release near build changes and build

Table 9 Results of the Mann-Whitney-Wilcoxon test (p: p-value) and Cliffs Delta *d* of the *normalized* distributions of the number of build changes near and non-near releases using the single day approach and sliding window approach with $k = 7$

Project	Threshold ($t = 10$)				Threshold ($t = 25$)			
	Single Day ($k = 1$)		Sliding ($k = 7$)		Single Day ($k = 1$)		Sliding ($k = 7$)	
	p	d	p	d	p	d	p	d
Alluxio/alluxio	$7.59e^{-44}$	0.78	$6.73e^{-15}$	0.63	$7.59e^{-44}$	0.78	$6.73e^{-15}$	0.63
apache/activemq	$1.65e^{-50}$	0.83	$9.19e^{-24}$	0.79	$1.65e^{-50}$	0.83	$9.19e^{-24}$	0.79
apache/camel	$6.16e^{-63}$	0.86	$5.51e^{-47}$	0.75	$6.16e^{-63}$	0.86	$5.51e^{-47}$	0.75
apache/flink	$9.70e^{-58}$	0.90	$3.18e^{-30}$	0.78	$9.70e^{-58}$	0.90	$3.18e^{-30}$	0.78
apache/hadoop	$1.76e^{-24}$	0.35	$8.20e^{-15}$	0.32	$1.76e^{-24}$	0.35	$8.20e^{-15}$	0.32
apache/hbase	$1.52e^{-38}$	0.35	$4.30e^{-11}$	0.22	$1.52e^{-38}$	0.35	$4.30e^{-11}$	0.22
apache/karaf	$8.72e^{-52}$	0.87	$8.83e^{-35}$	0.81	$8.72e^{-52}$	0.87	$8.83e^{-35}$	0.81
apache/storm	$1.83e^{-26}$	0.72	$2.25e^{-10}$	0.60	$1.83e^{-26}$	0.72	$2.25e^{-10}$	0.60
apache/wicket	$1.56e^{-113}$	0.71	$7.75e^{-26}$	0.51	$1.56e^{-113}$	0.71	$7.75e^{-26}$	0.51
deeplearning4j/deeplearning4j	$1.67e^{-35}$	0.95	$5.81e^{-23}$	0.85	$1.67e^{-35}$	0.95	$5.81e^{-23}$	0.85
druid-io/druid	$5.52e^{-155}$	0.76	$8.24e^{-70}$	0.66	$5.52e^{-155}$	0.76	$8.24e^{-70}$	0.66
eclipse/jetty.project	$6.66e^{-186}$	0.88	$1.36e^{-76}$	0.69	$6.66e^{-186}$	0.88	$1.36e^{-76}$	0.69
google/closure-compiler	$2.33e^{-175}$	0.50	$1.37e^{-49}$	0.58	$2.33e^{-175}$	0.50	$1.37e^{-49}$	0.58
google/guava	$4.51e^{-164}$	0.80	$8.42e^{-32}$	0.68	$4.51e^{-164}$	0.80	$8.42e^{-32}$	0.68
Graylog2/graylog2-server	$5.04e^{-167}$	0.86	$2.46e^{-31}$	0.51	$5.04e^{-167}$	0.86	$2.46e^{-31}$	0.51
hazelcast/hazelcast	$3.42e^{-58}$	0.49	$5.00e^{-17}$	0.38	$3.42e^{-58}$	0.49	$5.00e^{-17}$	0.38
hibernate/hibernate-search	$1.33e^{-151}$	0.85	$4.38e^{-42}$	0.68	$1.33e^{-151}$	0.85	$4.38e^{-42}$	0.68
jenkinsci/jenkins	$1.42e^{-276}$	0.78	$5.49e^{-12}$	0.17	$1.42e^{-276}$	0.78	$5.49e^{-12}$	0.17
languagetool-org/languagetool	$1.05e^{-61}$	0.95	$3.41e^{-17}$	0.86	$1.05e^{-61}$	0.95	$3.41e^{-17}$	0.86
naver/pinpoint	$8.73e^{-44}$	0.98	$5.43e^{-17}$	0.87	$8.73e^{-44}$	0.98	$5.43e^{-17}$	0.87
neo4j/neo4j	$2.01e^{-57}$	0.50	$4.98e^{-28}$	0.40	$2.01e^{-57}$	0.50	$4.98e^{-28}$	0.40
netty/netty	$1.68e^{-209}$	0.91	$1.26e^{-48}$	0.66	$1.68e^{-209}$	0.91	$1.26e^{-48}$	0.66
orientechnologies/orientdb	$5.85e^{-145}$	0.89	$1.44e^{-39}$	0.64	$5.85e^{-145}$	0.89	$1.44e^{-39}$	0.64
prestodb/presto	$2.04e^{-125}$	0.74	$5.28e^{-32}$	0.47	$2.04e^{-125}$	0.74	$5.28e^{-32}$	0.47
SonarSource/sonarqube	$7.51e^{-100}$	0.78	$2.20e^{-27}$	0.51	$7.51e^{-100}$	0.78	$2.20e^{-27}$	0.51
spring-projects/spring-boot	$4.97e^{-53}$	0.83	$1.66e^{-35}$	0.71	$4.97e^{-53}$	0.83	$1.66e^{-35}$	0.71
spring-projects/spring-roo	$1.85e^{-59}$	0.76	$9.81e^{-24}$	0.80	$1.85e^{-59}$	0.76	$9.81e^{-24}$	0.80
stanfordnlp/CoreNLP	$7.80e^{-01}$	-0.04	$2.50e^{-01}$	0.33	$7.80e^{-01}$	-0.04	$2.50e^{-01}$	0.33
wildfly/wildfly	$9.37e^{-52}$	0.90	$2.24e^{-36}$	0.80	$9.37e^{-52}$	0.90	$2.24e^{-36}$	0.80

changes that are far from the release again with the normalized data and recomputed the Mann-Whitney-Wilcoxon test. The results of the test of the normalized data can be found in Table 9. Investigating the plots, we see a flattening effect similar to the time line plots which indicates that the effect size might be lower as before but still seem to be statistically significant. This is confirmed by the results of the test in which we see similar significance and effect size values. To keep the paper concise, we refrained from including all the plots and tables of the study with the normalized data in the paper and refer the reader to the supplementary material for further information.

7.1.3 Discussion

This research question, we investigated the time when build changes are performed. We studied the evolution of the projects under investigation and found that build changes are not equally distributed over the time but often cluster around special points of time. To avoid observing the effect of simply having more commits near releases, we also studied the number of changes normalized by the number of commits on that particular point in time and found that this effect can also be observed in the normalized data. Investigating the special points in time, we found that many changes to the build specification are performed around releases. While this can be natural behavior motivated by bug fixing before a release or preparing a release, there are many changes that might not necessarily be performed shortly before the release. We argue that developers should aim at applying changes to the build specification as early as possible to avoid late effects of merging such changes. However, the plots depicting the number of build changes per day indicate that in the projects that we studied, most of the build changes are performed around releases.

With these results, we can answer research question RQ2.1, *How are build changes distributed over time?*, as follows:

Build changes are not equally distributed over the projects' timeline. Some points in time show a significantly higher number of build changes frequencies than others. We found that especially around release days, a high build change frequency is observed.

7.2 Distribution of build change types (RQ2.2)

In RQ1.1, we observed that build changes are usually clustered around releases. However, as the number of build changes is an aggregation of various types of modifications to the build specification, we examine whether specific build changes or build change categories also cluster around releases or show a different distribution. Furthermore, the clustering around release days is imprecise. We address this issue and study whether particular changes are performed before, on, or after the release day. This gives us a deeper insight into when exactly the changes are performed around release days and knowing when particular changes are usually performed can help in avoiding build breakage as described in the introduction of this section. We first describe the approach that we used to study the change types over time and then present the results of the study. The last part of this section discusses our findings.

7.2.1 Approach

For this research question, we used the data of RQ2.1 with one minor exception. Instead of summing up all the build change counts, we keep the counts per change type. Moreover, we add five columns representing the build change categories described in Section 4 and calculated their value by summing up all the values of the build change types that fall under the respective build change category. For example, the column `Team Changes` contains the sum of all values of, for example, DEVELOPER_INSERT or CONTRIBUTOR_DELETE. We also reuse the release data from RQ2.1 in this research question and perform the study with the mentioned parameters for the release-near threshold t (t calculated with $p \in \{10\%, 25\%\}$) and the window size $k \in \{1, 3, 5, 7\}$.

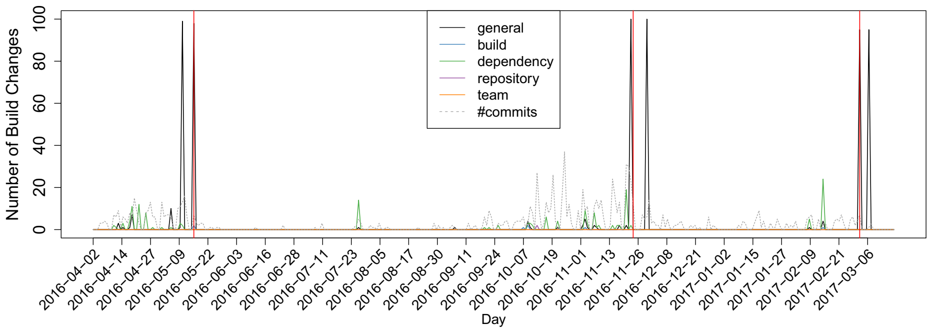


Fig. 10 Excerpt of the number of build changes per build change category over time in the `spring-roo` project using a single day approach ($k = 1$). Releases are depicted as vertical red lines

7.2.2 Results

We first present the results of the investigation of the build change categories and then we show the results of the detailed investigation per build change type.

Concerning build change categories, Figs. 10 and 11 show the changes of each category that have been performed over time. Similar as before, we can see that the two charts are similar. In both figures, changes of the category `General Changes` and `Dependency Changes` occur with high peaks in the plot. However, it seems that the peaks are occurring at different places in the time line. `General Changes` seem to happen on the release day whereas `Dependency Changes` also happen frequently before and after releases. To substantiate this visual indication, we investigate the indication that changes of certain build change categories frequently happen at particular times in the time line in more detail. We differ between changes that happen *before*, *on*, and *after* a release day. For each of these three special periods of time around release days, we compared the data points that are counted in the special period of time with the other two (*e.g.* we compared the data points for before with the data points of on and after). Figure 12 shows the p-values and Cliff's delta values of the frequency of each of the build change categories concerning their occurrence before, on, and after releases. In the upper plot, the two dashed red lines indicate p-values

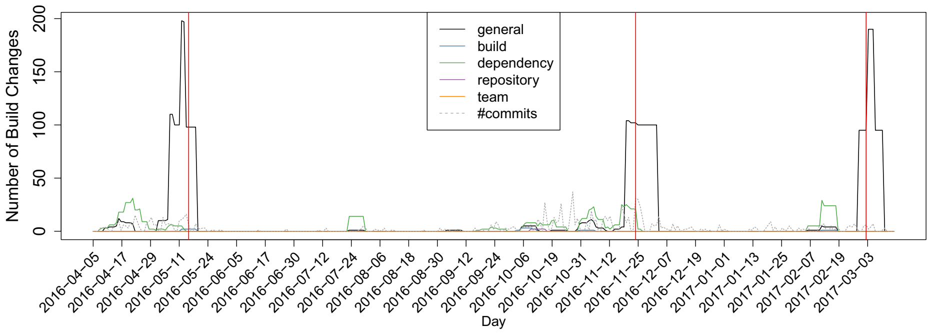


Fig. 11 Excerpt of the number of build changes per build change category over time in the `spring-roo` project using a sliding window ($k = 7$). Releases are depicted as vertical red lines

of 0.05 and 0.01. The three dashed lines in the lower plot represent the threshold values for the effect size categories negligible, small, medium, and large.

We can see that build changes of type `General Changes` happen at all three positions showing median values (p-value/Cliff's delta) of $4.80e^{-19}$/0.21, $3.10e^{-61}$/0.89, and $2.40e^{-9}$/0.13 for before, on, and after the release day. The Cliff's delta for the changes on the release day can be considered large, before the release day as small, and after the release day as negligible. Concerning changes to the category `Build Changes`, we see that they usually occur before an release (0.006/0.04) or directly on the release day ($7.60e^{-6}$/0.15). As depicted in the plot, they normally do not occur frequently after releases (0.12/0.05). `Dependency Changes` show a similar behavior before the release day (0.002/0.15) and on the release day ($1.24e^{-5}$/0.15). For the categories `Repository Changes` and `Team Changes`, we could not find statistical evidence for a clustering before, on, or after the release day.

We also investigated the occurrence of the fine-grained build change types before, on, and after releases. Figure 13 shows the p-values and Cliff's delta effect size measure for

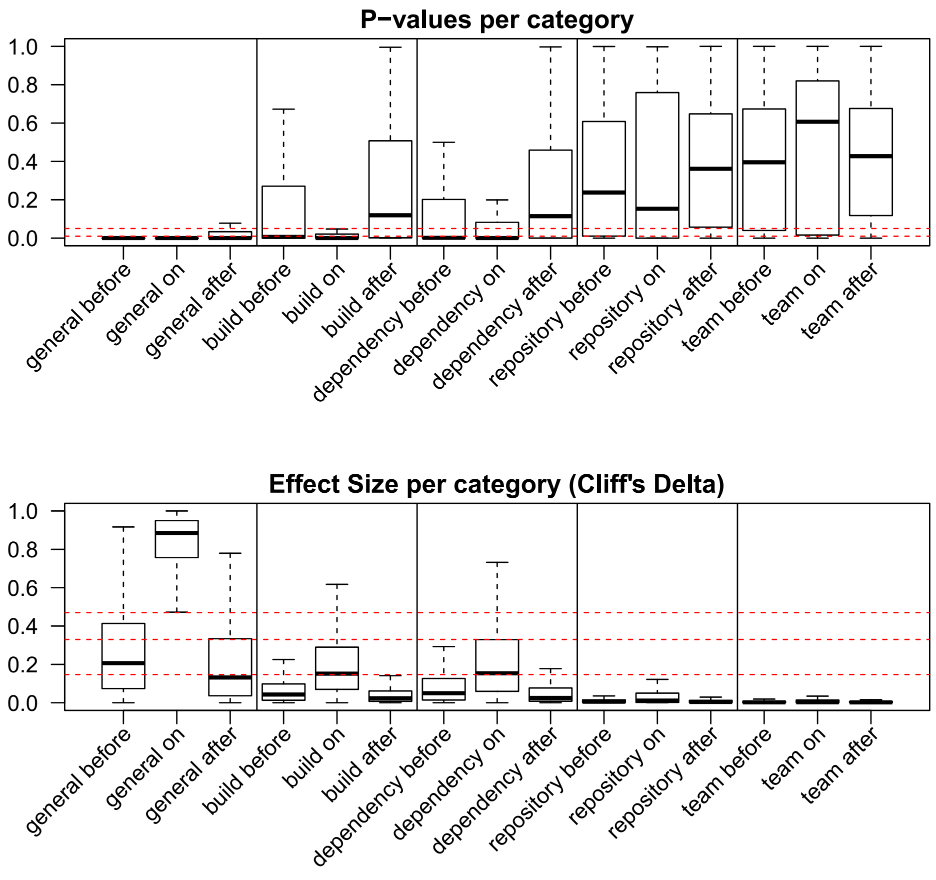


Fig. 12 Boxplots of the p-values and Cliff's delta values of all of the projects, windowSizes, and threshold values per change type category

the change types with $p < 0.05$. We see that several change types are statistically significantly more often observed nearby release days. In general, we observe four categories of changes. First, we observe that changes to the parent and the project versions are usually applied around release days. We argue that this is an expected observation because during a release, the version number is usually updated. Second, we observe that properties are often changed before and on the release day. As found by our prior work (Macho et al. 2018), version numbers are usually covered within properties which can imply that the property updates often refer to version updates (*e.g.* for the project/parent version or a dependency version). The case of updating the dependency version is also observed with the explicit change type DEPENDENCY_POSTFIX_VERSION_UPDATE on the release day. Lastly, we observed that also critical changes, such as PLUGIN_CONFIGURATION_UPDATE and DEPENDENCY_INSERT, are often changed on the release day.

As depicted above, we recomputed these values for the normalized data as well. We again do not show all figures in the paper to keep it concise. Once more, we see that overall the changes are very similar compared to the non-normalized version. We still see the different types of version changes for the project and the parent project happening around releases. Furthermore, we still find updates to the plugin configuration and inserting plugins in this figure, although the effect size is negligible.

These observations let us conclude that still many changes are performed on the release day itself. In particular, the changes to the plugin configuration and the dependency system are considered critical because such changes can easily break the build. Hence, in an ideal process they should not be performed on the release day. While some of such changes might be necessary to be performed on the release day (*e.g.* in form of late merges of pull requests of features that need to be released or fixes to repair the build), we argue that these changes should be performed during the development in between two consecutive releases.

As mentioned in Section 7.1.2, we only performed our experiments using the absolute number of build changes that were performed on a daily basis. We also applied the normalization using the ratio of the number of build-changing commits to the number of total commits per day to respect the fact that the number of commits increases before releases.

Similar to the observations in the previous section, we see that the peaks of build change categories remain very similar although some peaks appear to be damped. For example, the two large peaks at the end of the depicted time frame appear lower with normalization. However, the big picture remains the same and suggests that the categories `General Changes`, `Build Changes` and `Dependency Changes` seem to happen especially around releases.

Again, we see that the overall picture of the normalized boxplots is very similar compared to Fig. 12. The median values (p-value/Cliff's Delta) for the category `General Changes` are $1.68e^{-16}/0.18$, $2.63e^{-59}/0.82$, and $1.94e^{-09}/0.13$ for before, on, and after the release day. For `Build Changes`, we observe values of 0.007/0.04 and $6.31e^{-06}/0.15$ for before and on the release day, respectively. Changes of this category usually do not happen directly after release days which is indicated by the values of 0.10/0.02. Similar as before, the changes of the `Dependency Changes` category show the same pattern. They are usually performed before (0.002/0.05) and on ($6.80e^{-06}/0.16$) the release day, but not after (0.11/0.02). The categories `Repository Changes` and `Team Changes` still show p-values larger than 0.05 for all three time periods which indicates that they are not particularly performed before, on, and after releases. Summarizing the results of the normalized analysis, we see that the p-values and the effect sizes are very similar compared to the results of the experiment without normalizing the data.

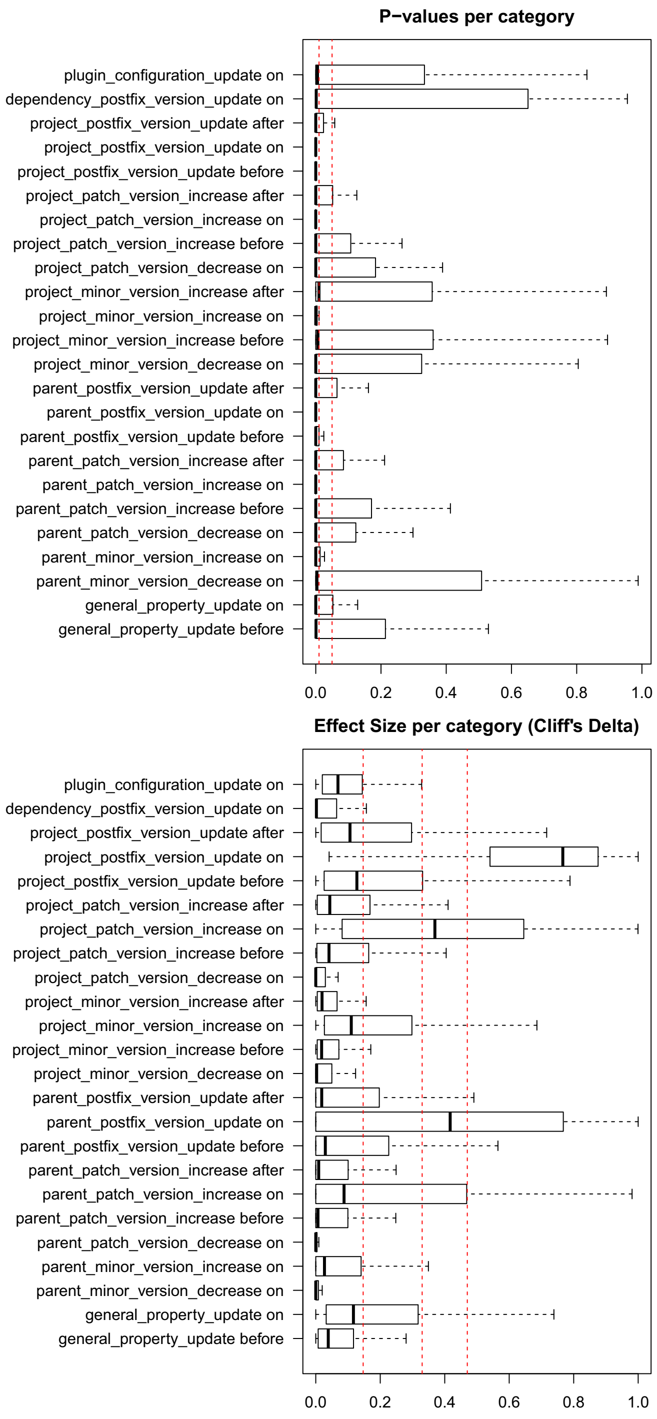


Fig. 13 Boxplots of the p-values and Cliff's delta values of all of the projects, window sizes, and threshold values per change type. ($p < 0.05$)

7.2.3 Discussion

Based on the observations of RQ2.1, we investigated the effect of build change clustering around releases in this study because intuitively one would assume that not all types of changes may show this effect. In fact, we studied the changes in more detail. We separately investigated the number build changes over time by their build change category and their build change type and again found that particular build changes types are performed around releases while we could not observe this effect for other build change types. This can increase the danger of introducing changes that might break the build just before a planned release. As discussed in Section 6.1.3, we consider changes to the build part and the dependency management system as potentially critical, especially when this changes happen around releases. However, further research is necessary to show and measure this potential risk. From the point of view of this work, we conclude that developers need to be especially careful when introducing such changes shortly before a release. Research needs to address this potential risk and provide approaches and tools to detect such late, potentially critical changes and to avoid the bad impact that these changes may cause (*i.e.* breaking the build). Looking at the plots of the build changes over time, we see that such changes are actually happening before releases.

With these results, we can answer research question RQ2.2 *Which build changes happen before, during, and after releases?*:

We found that, in particular, the build change categories `General Changes` and `Dependency Changes` occur more frequently around releases than other categories. Furthermore, we found that critical changes, such as plugin configuration updates, are performed shortly before or on the release day.

8 Discussion

In this section, we first discuss implications of our results on recent and ongoing research of build systems and their configuration. Furthermore, we discuss implications for developers who use Maven as build system. Finally, we discuss the threats to the validity of our results.

8.1 Implications of the results

The major implication that follows from the results our study is that tooling that supports developers when changing their build configurations is needed. For example, considering late changes right before an upcoming release can be dangerous and further work is needed to estimate this danger to help developers to avoid introducing bugs with these changes. Furthermore, our results about frequent build change patterns also help to identify possible problems when updating the build configuration, *e.g.* by indicating a missing co-change, and also requires tooling to directly help developers. In the following, we describe more implications and potential applications of our results in more detail.

On research Compared to the state-of-the-art, our fine-grained build changes enable a more detailed analysis of changes to the build system specification. This can help research in several ways.

First, approaches, such as our previous work on repairing dependency-related build breakage (Macho et al. 2018), can benefit from this work. This approach studies changes to the build specification and identifies patterns to automate the repair of broken builds. The process of studying build changes that the developers made to repair a broken build can be refined by using the extended version of BUILDDIFF. We believe that additional change patterns that repair broken builds can be found and that the additional information on the type of a version change can improve the existing repair strategies.

Second, we believe that studies on effort estimation can benefit from our detailed insights into build changes. As maintenance, and build maintenance in particular, are also tasks that need to be planned, we argue that with a deeper knowledge of when and which type of build changes are performed a better planning of such activities is possible. For example, managers can recognize that over the last time less then usual time on maintaining the build specification was spent, and as a consequence, more time for maintenance might be needed soon. However, this line of research needs to be investigated in more detail and studies, such as from Corazza et al. (2010,2013), Ferrucci et al. (2014), and Kocaguneli et al. (2012) and Kocaguneli et al. (2010), can benefit from our approach by incorporating our fine-grained information about build changes.

Third, refactoring approaches, such as MAKAO (Adams et al. 2007b) and Formiga (Hardt and Munson 2015) can be transfered to MAVEN build specifications. Enriched with our detailed change information, these approaches can lead to refactoring tools that improve the quality of MAVEN build specifications. For example, our work on build change patterns revealed best practices for changing build specifications based on empirical findings. Researchers can further study build change patterns and derive useful refactorings, such as managing dependencies in the management part, that can be automated to improve the quality of build specifications.

Fourth, studies of build complexity (McIntosh et al. 2012) can also benefit from our detailed analysis of build changes by including dynamical information, such as our detailed build change information, to the calculation of the metrics. Researchers can use this information for various studies, including studying the evolution of build systems in more detail or estimating complexity to refactor build specifications.

Finally, our build changes can be used to improve prediction models. Prior studies often used churn or similar measures to approximate the amount of changes. However, this might not fully represent the actual changes that were performed. Similar to improving models to predict bug-prone build files (Giger et al. 2011) or suggest potentially missing changes to build configurations (Macho et al. 2016) for source code files, using BUILDDIFF may help to refine prediction models.

On development We observed that build changes occur more frequently near release days. This observation can help developers to avoid build breakage by increasing the awareness that changes to the build configuration directly before or on the release day is critical and can possible break the build. More specifically, we found that changes to the dependency declarations, especially inserting new dependencies, and configuration updates of plugins are often performed close before or even on the day of a release. Performing such essential changes to the build specification close to a release might increase the risk of build breakage. As a counteraction to avoid such a risk increase, we recommend to consult build experts prior to merging such changes to the release branch. This consultation, for example, can be performed by including dedicated experts from the build or release engineering department to review the changes in the corresponding pull request. Another counteraction that developers may consider is to early change the build configuration, especially by updating the versions of dependencies earlier and by changing plugin configurations earlier in the development iteration. However, as already mentioned above, more tooling is needed to support developers to avoid introducing such changes first hand, which we also aim to provide in future work.

Furthermore, project managers can use this finding to consider the peak of build changes near release days in their planning of releases and work load. For example, if only a few developers are experts in maintaining the build configuration, project managers can plan ahead and reserve time buffers before releases for these developers.

We also give insight into the type of build changes that are frequently made. This can be used by developers, for instance, to identify and refactor plugins that often change their configuration. We believe that frequently changed plugin configurations are also more prone to cause build breakage, which is especially critical if such changes are performed shortly before releases. Our analysis of build change patterns can lead to tools that developers can use to detect missing changes in their change sets, for example, if they change the source code management information. In this context, we also want to highlight that further research is needed to achieve these goals, in particular, we plan to relate frequently performed changes and change patterns with the build result to investigate the impact of such changes on the build result.

Besides using the results that we found in our studies, developers can also use BUILDDIFF and integrate it with a code visualization tool. Providing precise and concise change visualization will help the developers and code reviewers to faster review the code and it can even help to improve the quality of the reviews, *e.g.* because the attention of the code reviewer can be drawn to the actual changes. Compared to the commonly used line difference visualization, our tool helps to highlight only such actual changes, excluding moving elements or changing white spaces, and enrich this visualization with the type of the change.

Lastly, the detailed change information can lead to tools that can support lead development teams in deciding who should review the code and how critical the code review is. For example, we believe that modifying critical parts of the build system specification needs more experienced reviewers than modifying uncritical parts of the build system specification. Highlighting the changes can also help developers during such code reviews to detect and categorize the changes that were performed.

Summary Summarizing, we give the following actionable results to researchers and developers:

- Researchers can use BUILDDIFF to refine existing approaches and to study different aspects of build maintenance, such as the impact of build changes. With this work, we contribute towards a more holistic view of changes happening to software projects.
- Our work especially contributes to refine prediction models dealing with changes by giving more detailed information on changes to build specification files.
- Developers can use our best practice findings and should check whether violations to the build change patterns that we found were unintentionally made.
- Commits and pull requests that contain potentially critical changes to the build specification need a more careful review, such as by developers who are build experts.

8.2 Threats to validity

Regarding the validity of our results, we identified the following threats to construct, internal, and external validity.

Construct validity One threat is that our taxonomy may not cover all possible change types or change categories that could be theoretically made to a build specification. We mitigated this threat in two ways. First, we compared the taxonomy with the XML schema of MAVEN build files to cover all important changes. Second, we asked two experienced MAVEN users to verify the taxonomy and create the categories, including a discussion if necessary.

Furthermore, we retrieved the release data of the 144 open source Java projects from GitHub as the only resource. To that extent, we could miss possible releases if they are not covered by the GitHub data. However, we mitigated this threat by manually checking if the data is compliant with the data provided by the source code management system.

In RQ2.1 and RQ2.2, we study build changes on a daily basis. This may not fully reflect the actual order of commits in the repository. However, we quantified this threat and found that 51% of the releases include no other commit, 77% include 1 or less, and 85% 2 or less. The remaining 15% include a larger amount. Hence, we believe that the bias that we might introduce with our heuristic is sufficiently small.

Internal validity A threat to internal validity concerns whether BUILDDIFF can extract the changes to a build configuration file accurately. We mitigated this threat by covering all changes of the taxonomy with JUnit tests and a manual evaluation comparing against the opinions of two experienced MAVEN users. Furthermore, this is the second independent evaluation of our tool because the basic changes have already been evaluated in the original work (Macho et al. 2017) and the selected data for the evaluation in this paper was selected independently from the original data set. Concerning the evaluation of BUILDDIFF, a threat is that the randomly selected commits do not include all change types. We mitigated this threat by calculating the proportion of actually missed changes due to the selection. We observed that we only miss 1.0% of the changes and hence, we can safely assume that the majority of the changes are covered by BUILDDIFF. For our study of build change patterns, we used the well-known metrics support and confidence. However, as also pointed out by prior work (Le and Lo 2015) these metrics might not be sufficient to precisely express the interestingness of the change patterns in the data. We mitigated this threat by incorporating other metrics, such as lift, odds-ratio, and leverage, to measure the quality of the observed change patterns.

Concerning the build change patterns, we identified another threat to validity. We constructed the item sets per commit. However, this way we might miss patterns that are scattered across multiple commits (*e.g.* a dependency management insert was performed in commit A but the corresponding insert to the dependency declarations was performed later in commit B). We mitigated this threat by analyzing 500 randomly selected build-changing commits. For each of these commits, two authors manually inspected the five preceding and the five succeeding commits. We validated whether one of these ten surrounding commits contained other build changes, and, if yes, whether these build changes might be coupled to the build changes in the selected commit. If there was at least a little indication that the changes might be coupled, we counted them as possible coupled build changes. In this investigation, we found that 86.40% of the commits have at least one build change in the ten surrounding commits. However, in only 8.20% of the cases, we found an indication that these build changes might belong together. Hence, we believe that the error that might be introduced due to the way we construct our item sets is small enough to draw valid conclusions.

Furthermore, we study all the commits that changed the build file. However, some commits are performed by non-humans, such as tools which we also included to calculate our build change patterns. We mitigated this threat by manually investigating 500 randomly selected build changing commits. In these 500 commits, the only used tool that we could identify was the `maven-release-plugin`. Although 11,037 (60.60%) commits out of 18,215 in which this pattern appears are performed by the tool, we believe the other 39.40% are still enough evidence that the pattern is also performed by humans, and hence, important to ensure the quality of build specifications as described in Section 6.2.3.

Another threat to internal validity concerns the way we normalize the data. As described in Section 7.1, we normalize the number of build changes with the ratio of build-changing commits per day. Various other ways, such as normalizing with the total number of changes, might be more suitable. However, in this work, we do not have a holistic knowledge about other change types, such as source code changes, but we are convinced that our approach of normalizing is a sufficient approximation.

We aggregate the build changes on a daily basis which may include changes of commits that actually happened after the releasing commit. However, we investigated this threat and found that the median number of commits on the same day but after the releasing commit is 0 and the 3^{rd} quartile is 1. Hence, for the majority of the releases, we only cover the commits that have actually been released with this commit.

External validity The main threat to external validity stems from the selection of projects that we used in our study. We mitigated this threat by selecting 144 open source Java projects of different vendors, sizes, and purposes. This increases the number of investigated projects from the original work (30) by a factor of 4.8. However, additional experiments with projects using other build systems and in particular from industrial settings are needed to further generalize our results. Another threat to external validity is that our taxonomy is tailored to MAVEN build configurations. While we designed the taxonomy to be usable for other build tools as well, the taxonomy may not generalize to all other build systems.

9 Conclusions

Build systems are an essential part in the engineering process of modern software systems. In this paper, we proposed BUILDDIFF, an approach for extracting fine-grained build changes from MAVEN build files. In a manual evaluation, we showed that BUILDDIFF is capable of extracting build changes with an average precision, recall, and f1-score of 0.97, 0.98, and 0.97, respectively. With the build changes extracted from 144 open source Java projects, we investigated two main aspects of build changes. First, we investigated their frequencies by analyzing the single change types, change type categories, and change patterns of build changes. Second, we studied the times when build changes and build changes categories are performed. These two main research questions led to the following results.

Concerning RQ1 *Which build changes occur the most frequently?*:

- **(RQ1.1)** The most frequently occurring build change types are PROJECT_POSTFIX_VERSION_UPDATE, PARENT_POSTFIX_VERSION_UPDATE, and DEPENDENCY_INSERT. The most frequent change category is `General Changes` followed by `Dependency Changes`, and `Build Changes`. The top-10 change types account for 51% of all changes.
- **(RQ1.2)** The three URL declarations of the source code management system are usually changed together. Furthermore, the version numbers of the parent project and the project itself are frequently changed together. Within this change pattern, we found that also the changed part of both is usually the same (*e.g.* a major version increase). Lastly,

we found that the declaration of dependencies often co-occurs with a corresponding declaration in the dependency management part of the build specification.

Concerning RQ2 *When are build changes recorded?*:

- **(RQ2.1)** Build changes are not equally distributed over the projects' timeline. We observed that especially near release days build changes occur more frequently than in days that are not near a release day.
- **(RQ2.2)** The build change categories `General Changes` and `Dependency Changes` occur more frequently close to release days than other build change categories. Critical changes, such as plugin configuration updates, are performed shortly before or on the release day.

Research can benefit from our results because studies on build system specifications can be refined using detailed information of build changes. Developers in industry can use our findings to increase awareness of when (not) to modify the build specification and which changes to include in their change sets for releases.

Future work We plan to use the extended version of BUILDDIFF to improve the repair strategies for automated repair of broken builds. Moreover, we aim at extending our approach to support other build systems, such as Gradle,[19] and compare the evolution of Gradle build specifications with MAVEN build specifications. By linking our fine-grained build change data with build results, we want to study the impact of build changes on the build outcome. Connecting our finding that many build changes are clustered around releases, we plan to investigate whether this has a negative impact on the build outcome. Finally, we plan to perform a more detailed analysis of the co-evolution between build changes and source code changes.

Funding Open Access funding provided by University of Klagenfurt.

References

Adams B, Schutter KD, Tromp H, Meuter WD (2007a) The Evolution of the Linux Build System, Electron Commun Eur Assoc Softw Sci Technol 8. https://doi.org/10.14279/tuj.eceasst.8.115

Adams B, Tromp H, Schutter KD, Meuter WD (2007b) Design recovery and maintenance of build systems. In: Proceedings of the international conference on software maintenance, pp 114–123. IEEE

Agrawal R, Imieliński T, Swami A (1993) Mining association rules between sets of items in large databases. In: Proceedings of the 1993 ACM SIGMOD international conference on management of data, SIGMOD '93, pp 207–216, New York, NY, USA. ACM

[19] https://gradle.org

Agrawal R, Srikant R et al (1994) Fast algorithms for mining association rules. In: Proc. 20th int. conf. very large data bases, VLDB, vol 1215, pp 487–499

Benelallam A, Harrand N, Soto-Valero C, Baudry B, Barais O (2019) The maven dependency graph: a temporal graph-based representation of maven central. In: 2019 IEEE/ACM 16th international conference on mining software repositories (MSR), pp 344–348. IEEE

Bezemer C-P, McIntosh S, Adams B, German DM, Hassan AE (2017) An empirical study of unspecified dependencies in make-based build systems. Empir Softw Eng 22(6):3117–3148

Brin S, Motwani R, Ullman JD, Tsur S (1997) Dynamic itemset counting and implication rules for market basket data. Acm Sigmod Record 26(2):255–264

Cliff N (1993) Dominance statistics: Ordinal analyses to answer ordinal questions. Psychol Bull 114(3):494

Cohen J (1968) Weighted kappa: nominal scale agreement provision for scaled disagreement or partial credit. Psychological bulletin 70(4):213

Corazza A, DiMartino S, Ferrucci F, Gravino C, Sarro F, Mendes E (2010) How effective is tabu search to configure support vector regression for effort estimation? In: Proceedings of the 6th international conference on predictive models in software engineering, p 4. ACM

Corazza A, DiMartino S, Ferrucci F, Gravino C, Sarro F, Mendes E (2013) Using tabu search to configure support vector regression for effort estimation. Empir Softw Eng 18(3):506–546

Désarmeaux C, Pecatikov A, McIntosh S (2016) The Dispersion of Build Maintenance Activity across Maven Lifecycle Phases. In: International conference on mining software repositories, pp 492–495. ACM

Dintzner N, Van Deursen A, Pinzger M (2014) Extracting Feature Model Changes from the Linux Kernel Using FMDiff. In: International workshop on variability modelling of software-intensive systems, pp 22:1–22:8. ACM

Dotzler G, Philippsen M (2016) Move-optimized source code tree differencing. In: Proceedings of the 31st IEEE/ACM international conference on automated software engineering, pp 660–671. ACM

Espinha T, Zaidman A, Gross H-G (2015) Web api growing pains: Loosely coupled yet strongly tied. J Syst Softw 100:27–43

Falleri J-R, Morandat F, Blanc X, Martinez M, Monperrus M (2014) Fine-grained and accurate source code differencing. In: Proceedings of the international conference on automated software engineering, pp 313–324. ACM

Ferrucci F, Harman M, Sarro F (2014) Search-based software project management. In: Software project management in a changing world, pp 373–399. Springer

Fluri B, Gall HC (2006) Classifying change types for qualifying change couplings. IEEE International Conference on Program Comprehension 2006:35–45

Fluri B, Würsch M, Giger E, Gall HC (2009) Analyzing the co-evolution of comments and source code. Softw Qual J 17(4):367–394

Fluri B, Würsch M, Pinzger M, Gall HC (2007) Change distilling: Tree differencing for fine-grained source code change extraction. Transactions on Software Engineering 33(11):725–743

Gall HC, Fluri B, Pinzger M (2009) Change analysis with evolizer and changedistiller. IEEE Software 26(1):26–33

Giger E, Pinzger M, Gall HC (2012) Can we predict types of code changes? an empirical analysis. In: Working conference of mining software repositories, pp 217–226. IEEE

Giger E, D'Ambros M, Pinzger M, Gall HC (2012) Method-level bug prediction. In: International symposium on empirical software engineering and measurement, pp 171–180. ACM

Giger E, Pinzger M, Gall HC (2011) Comparing Fine-grained Source Code Changes and Code Churn for Bug Prediction. In: International working conference on mining software repositories, pp 83–92. ACM

Grissom RJ, Kim JJ (2005) Effect Sizes for Research: A Broad Practical Approach. Lawrence Erlbaum Associates, New Jersey

Hahsler M, Gruen B, Hornik K (2005) arules – A computational environment for mining association rules and frequent item sets. J Stat Softw 14(15):1–25

Hardt R, Munson EV (2013) Ant build maintenance with formiga. In: Proceedings of the international workshop on release engineering, pp 13–16. IEEE

Hardt R, Munson EV (2015) An empirical evaluation of ant build maintenance using formiga. In: Proceedings of the international conference on software maintenance and evolution, pp 201–210

Hashimoto M, Mori A (2008) Diff/TS: a tool for fine-grained structural change analysis. In: 2008 15th Working conference on reverse engineering, pp 279–288. IEEE

Hattori L, Lanza M (2008) On the nature of commits. In: Proceedings of the international conference on automated software engineering, pp 63–71. ACM/IEEE

Kerzazi N, Khomh F, Adams B (2014) Why do automated builds break? an empirical study. In: International conference on software maintenance and evolution, pp 41–50. IEEE

Kocaguneli E, Menzies T, Keung JW (2012) On the value of ensemble effort estimation. IEEE Trans Softw Eng 38(6):1403–1416

Kocaguneli E, Tosun A, Bener A (2010) Ai-based models for software effort estimation. In: Software engineering and advanced applications (SEAA), 2010 36th EUROMICRO Conference on, pp 323–326. IEEE

Kula RG, German DM, Ouni A, Ishio T, Inoue K (2018) Do developers update their library dependencies? Empir Softw Eng 23(1):384–417

Le T-DB, Lo D (2015) Beyond support and confidence: Exploring interestingness measures for rule-based specification mining. In: Software analysis, evolution and reengineering (SANER), 2015 IEEE 22nd International Conference on, pp 331–340. IEEE

Lebeuf C, Voyloshnikova E, Herzig K, Storey M-A (2018) Understanding, debugging, and optimizing distributed software builds: A design study. In: 2018 IEEE International conference on software maintenance and evolution (ICSME), pp 496–507. IEEE

Livshits B, Zimmermann T (2005) Dynamine: finding common error patterns by mining software revision histories. In: ACM SIGSOFT Software Engineering Notes, vol 30, pp 296–305. ACM

Lubsen Z, Zaidman A, Pinzger M (2009) Using association rules to study the co-evolution of production & test code. In: Mining software repositories, 2009. MSR'09. 6th IEEE International working conference on, pp 151–154. IEEE

Maarek YS, Berry DM, Kaiser GE (1991) An information retrieval approach for automatically constructing software libraries. Transactions on Software Engineering 17(8):800–813

Macho C, McIntosh S, Pinzger M (2016) Predicting Build Co-Changes with Source Code Change and Commit Categories. In: Proceedings of the international conference on software analysis, evolution, and reengineering, pp 541–551. IEEE

Macho C, McIntosh S, Pinzger M (2017) Extracting Build Changes with BuildDiff. In: Proc. of the international conference on mining software repositories (MSR), pp 368–378

Macho C, McIntosh S, Pinzger M (2018) Automatically Repairing Dependency-Related Build Breakage. In: Proc. of the International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp 106–117

Marsavina C, Romano D, Zaidman A (2014) Studying fine-grained co-evolution patterns of production and test code. In: Source code analysis and manipulation (SCAM), 2014 IEEE 14th International working conference on, pp 195–204. IEEE

McIntosh S, Adams B, Hassan AE (2010) The evolution of ANT build systems. In: Proceedings of the international working conference on mining software repositories, pp 42–51. IEEE

McIntosh S, Adams B, Hassan AE (2012) The evolution of java build systems. Empir Softw Eng 17(4-5):578–608

McIntosh S, Adams B, Nagappan M, Hassan AE (2014) Mining co-change information to understand when build changes are necessary. In: Proceedings of the international conference on software maintenance and evolution, pp 241–250. IEEE

McIntosh S, Adams B, Nguyen ThanhHD, Kamei Y, Hassan AE (2011) An empirical study of build maintenance effort. In: 2011 33rd International Conference on Software Engineering (ICSE), pp 141–150. IEEE

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

Miller W, Myers EW (1985) A file comparison program. Software: Practice and Experience 15(11):1025–1040

Munaiah N, Kroh S, Cabrey C, Nagappan M (2017) Curating github for engineered software projects. Empir Softw Eng 22(6):3219–3253

Myers EW (1986) Ano (nd) difference algorithm and its variations. Algorithmica 1(1-4):251–266

Nielsen J (1995) Card sorting to discover the users' model of the information space

Piatetsky-Shapiro G (1991) Discovery, Analysis, and Presentation of Strong Rules. In: Piatetsky-Shapiro G, Frawley WJ (eds) Knowledge Discovery in Databases. AAAI/MIT Press, pp 229–248

Raemaekers S, Deursen AV, Visser J (2014) Semantic Versioning versus Breaking Changes : A Study of the Maven Repository. In: Proceedings of international working conference on source code analysis and manipulation, pp 215–224

Ren Z, Jiang H, Xuan J, Yang Z (2018) Automated localization for unreproducible builds. arXiv:1803.06766

Romano D, Pinzger M (2011) Using source code metrics to predict change-prone Java interfaces. In: Proceedings of the international conference on software maintenance, pp 303–312. IEEE

Seo H, Sadowski C, Elbaum S, Aftandilian E, Bowdidge R (2014) Programmers' build errors: a case study (at google). In: International conference on software engineering, pp 724–734. ACM

Shridhar M, Adams B, Khomh F (2014) A qualitative analysis of software build system changes and build ownership styles. In: Proceedings of the 8th ACM/IEEE international symposium on empirical software engineering and measurement, pp 1–10

Soto-Valero C, Benelallam A, Harrand N, Barais O, Baudry B (2019) The emergence of software diversity in maven central. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp 333–343. IEEE

Soto-Valero C, Harrand N, Monperrus M, Baudry B (2020) A comprehensive study of bloated dependencies in the maven ecosystem. arXiv:2001.07808

Tamrawi A, Nguyen HA, Nguyen HV, Nguyen TN (2012) Build code analysis with symbolic evaluation. In: Proceedings of the international conference on software engineering, pp 650–660. IEEE

Tan P-N, Kumar V, Srivastava J (2004) Selecting the right objective measure for association analysis. Inf Syst 29(4):293–313

Vassallo C, Proksch S, Zemp T, Gall HC (2018) Un-break my build: Assisting developers with build repair hints

Wang Y, Wen M, Liu Z, Wu R, Wang R, Yang B, Yu H, Zhu Z, Cheung S-C (2018) Do the dependency conflicts in my project matter? In: Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp 319–330. ACM

Wang Y, Wen M, Wu R, Liu Z, Tan SH, Zhu Z, Yu H, Cheung S-C (2019) Could i have a stack trace to examine the dependency conflict issue? In: 2019 IEEE/ACM 41st international conference on software engineering (ICSE), pp 572–583. IEEE

Wen R, Gilbert D, Roche MG, McIntosh S (2018) Blimp tracer: Integrating build impact analysis with code review. In: 2018 IEEE International conference on software maintenance and evolution (ICSME), pp 685–694. IEEE

Xia X, Lo D, McIntosh S, Shihab E, Hassan AE (2015) Cross-project build co-change prediction. In: Proceedings of the international conference on software analysis, evolution, and reengineering, pp 311–320. IEEE

Xia X, Lo D, Wang X, Zhou B (2014) Build system analysis with link prediction. In: Symposium on applied computing, pp 1184–1186. ACM

Zaki MJ (2001) Spade: An efficient algorithm for mining frequent sequences. Machine learning 42(1-2):31–60

Zimmermann T, Zeller A, Weissgerber P, Diehl S (2005) Mining version histories to guide software changes. IEEE Trans Softw Eng 31(6):429–445

Affiliations

Christian Macho[1] · **Stefanie Beyer**[1] · **Shane McIntosh**[2] · **Martin Pinzger**[1]

Stefanie Beyer
stefanie.beyer@aau.at

Shane McIntosh
shane.mcintosh@uwaterloo.ca

Martin Pinzger
martin.pinzger@aau.at

[1] University of Klagenfurt, Klagenfurt, Austria

[2] David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada