On the Relationship Between Architectural Changes and Continuous Integration Build Outcome

Anonymous Author(s)

Abstract

Continuous Integration (CI) is becoming an essential component in the software development process. However, the time cost of a CI build may often be high, if the build is broken. Therefore, identifying reasons for build failure and warning developers about these likely failures can save valuable time and cost for developers. In this paper, we investigate if there is a relationship between architectural changes and CI build outcomes. We form two hypotheses about the relationship: (1) architectural changes lead to higher CI build failures and (2) higher CI build failures lead to architectural changes. To investigate these hypotheses, we design a fully automated extensible framework that analyzes the architectural changes between two consecutive versions of a software system. We use three well-established techniques to reconstruct the projects' architecture from the source code and measure change with eight metrics. In addition, we analyze the build logs to investigate the point of failure during the build. We mine almost 50,000 builds from 159 open-source Java repositories, but find no significant correlation between architectural change and build outcome. To enable other researchers to replicate our negative results and to investigate additional architectural metrics, we publish our framework which can easily be extended for new research questions and metrics.

ACM Reference Format:
Anonymous Author(s). 2018. On the Relationship Between Architectural Changes and Continuous Integration Build Outcome . In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Continuous integration (CI) is nowadays common practice in software development [3]. It helps finding faults earlier and therefore reduces the costs to fix them [13]. On the other hand, build failures decrease the pace of the development [5]. Hence, by decreasing the number broken builds, the speed of development is increased whereby costs are decreased.

There are numerous research directions for predicting CI build failures. Within the context of the MSR 2017 challenge [4], different approaches emerged, using for example cascaded tree classifiers [23] or based upon the person who was submitting the commit [29]. In general, tree classifiers are a useful machine learning (ML) technique to predict build outcome using meta data about the current and previous commits [14]. These meta-data based ML approaches alone do not have high enough accuracy, because it has been shown that source code metrics impact the build outcome [17]. This paper wants to investigate this route further.

Paixão et al. have found correlation between CI build outcomes and non-functional requirements [25]. Since non-functional requirements are often addressed by software architects [2], it is interesting to investigate if the software architecture influences the build results or is itself influenced by continuous integration. This could happen if, for example, multiple builds fail and, thus, the architecture gets changed in the problematic area.

To investigate the relationship between CI and software architecture, it is necessary to measure the architectural change (AC) between two builds. Architectural change has been studied by different researchers in different contexts [6, 19, 22, 24] , e.g. finding the causes [8], checking consistency with the designed architecture [7, 11], investigating the impact on the business model [30] or the non-changed modules of the architecture [1].

Paixao et al. [26] show that most developers are not aware that their change in the code actually affects their architecture. If made aware, the developers tend to be more careful and the overall architecture gets improved deliberately by them. If we can prove to the developers that their actions that change the architecture influence the outcome of their builds, something they can see on a daily basis, this should increase their awareness.

The contribution of this work is the combination of architectural change with the prediction of continuous integration build outcomes. For that, the TravisTorrent dataset is exploited and the 159 Java Maven software systems are analyzed. Every build of those systems is reviewed for architectural change with respect to its predecessor using 8 different metrics. These are then compared against the build outcomes and later used prediction. We hypothesize that architectural change is either (1) preceded or (2) followed by failing builds. The change is preceded by failing builds

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

because if there is a bigger problem in the system, see-able through failing builds, it is necessary to change the architecture. On the other hand, it may be possible that change in the architecture results in unexpected consequences and lead, therefore, to build failures (2).

To investigate this problem, we develop an easy extensible framework. It allows to run a toolchain which automatically downloads snapshots of a software system, reconstructs the architecture and computes change metrics. With only a few lines of code, new extractors, reconstructors and metric calculators can be added. Doing so, we reuse tools for all mentioned steps from previous researches, namely HUSACCT [28], ARCADE [20] and Martin's metrics [21].

After calculating the change metrics for almost 50,000 commits using two different architecture reconstruction techniques, almost no correlation was found between change and build outcome. In total, 48 correlations are calculated with 32 different configurations. The hypothesis that architectural change impacts the build outcome must therefore be rejected under the tested circumstances.

2 Background

The TravisTorrent dataset [4] consists of over 3.5 million build samples done in TravisCI from over 1,300 projects, written in Java, JavaScript and Ruby. TravisCI is a continuous integration tool which is tightly integrated into GitHub. CI is used to automatically compile, build and test a software system, whenever new code is committed to the repository. This helps the developer to find bugs and should increase the development speed. TravisTorrent saves over fifty features for every build, including the project name, the commit ID of the build and the outcome.

Software architecture is usually represented as a graph of the high-level components, called modules, of a software system which are connected through links, called dependencies. It can be modeled with architecture description languages (ADLs), like the component diagram of the UML, though there is no standardized way [31]. As there is no single way for describing an architecture, there is no best way for reverse engineering the architecture. The most basic way is using the package structure. This assumes that the developers have thought about the architecture and translated it into packages. In that case, high level packages are modules, and module A is dependent on module B if an element of package A is using an element which is contained in package B. Other approaches for reconstructing architecture are ACDC or ARC as introduced in the following section.

3 Related Work

This section is split up into research about continuous integration builds and architecture change and consistency. This work combines both parts and is new in this regard.

3.1 Continuous Integration

Islam and Zibran [17] have studied the relationship between build outcome, and project and build metrics. They found, that there is no correlation between size or the number of contributors in the project and build outcomes. They also discovered that which tool is used for building the software (e.g. Maven or Ant) impacts the result of the build as well as the number of changed lines and files. Test code as well as the development branch did not impact the result. Since the number of changed lines and files affects the build result, we can conclude that source code has a relationship to the the build outcome, and architecture, which is directly affected by the source code, is a valid idea for investigation.

In 2006, a study achieved a 69% specificity rate (detecting failures) with decision trees using meta data about the developer and the project itself [14]. The research was continued for the MSR '17 challenge by using cascading tree classifiers. The authors improved the prediction accuracy using information about previous builds, e.g. the outcome of the last build [23]. In that research, the prediction was only done per project because it was shown that certain subsystems or developers are more erroneous. While this is certainly useful in practice, it does not help to mitigate the reasons of failing builds on a more general (i.e. cross-project) level.

3.2 Architecture

There are numerous approaches how architecture is researched in software engineering. Generally, there are two topics. First, the reconstruction (1) of the implemented architecture in the code and, second, the comparison (2) of this architecture against (a) the designed architecture or (b) the architecture of different versions of the system, i. e. architecture evolution.

The ARAMIS workbench [24] takes the step from static reconstruction of the architecture based on method calls and inheritance to extracting the data flow during runtime to check it against the predefined structure. For model-driven and generative software development, the task of extracting architecture is more complex than in traditional projects, because it passes through different layers of abstraction and possibly multiple DSLs. Thus, the way of checking consistency must account for different layers and is abstracted with architecture description languages (ADL) [6, 11]. Also using ADLs, Haitzer et al. [12] study architectural drift and erosion. In their system, the architect can simulate multiple implementation scenarios. With the help of the ADL, the program then calculates the consequences of this change, and the architect can make an educated decision which change has the best impact on the system and its architecture.

Caracciolo et al. [7] found that the automated checking of the implemented architecture with the intended one, leads to fewer architecture violations. They extracted the architecture using classic static analysis and a dependency graph.

Their system checks then for violations of the MVC pattern and reports it directly to the developer. Over the course of time, fewer violations were found compared to a control group, which shows that the developers learned how to avoid architecture violations.

Nakamura and Basili [22] introduced the measuring of architectural distance using kernels. For that, both architectures need to be represented as a graph structure (here OO class structure) and are then compared for similar substructures. This solves the problems with renaming and is applicable to every graph. Because it works on the complete class graph and not an abstracted version, i.e. the architecture, this technique is not viable for consistency checking, as the designed architecture is not represented down to the class level. Since the graph is compared to kernels, it is necessary to define a distance measurement for this. A similar approach is used by Garcia [9] with the cluster coverage metric.

Tonu et al. [33] have researched architecture stability, primarily in C and C++ projects, but also introduced support for Java.[1] Using four different types of metrics, growth, change, cohesion and coupling, they try to analyze when the architecture of a software system stabilizes and then predict stability for future versions. It does so by extracting facts from the source code (or in case of Java byte code) and reconstructing the architecture. A fact can be anything, from LoC to function calls and classes. In both studied projects, the architecture stabilizes relatively quickly and has only small changes afterwards. Unfortunately, we cannot use this tool in our research, because we want to study the source files and not the byte code of Java projects. But we do analyze change and coupling metrics.

HUSAACT [27, 28] is a highly customizable architecture conformance framework for Java and C#. Dependencies between architecture modules can be defined in multiple ways and different kinds of dependencies are possible. They argue, that a "call" dependency cannot be compared directly to a "inherits" dependency and, hence, must be treated differently. The framework extracts several types of dependencies between source code elements (e.g. classes, interfaces, packages). HUSAACT needs to have a module specification, i.e. which root packages or classes are part of which architecture module. If given this, it maps the defined architecture vs the real implementation and reports violations. If no such specification is available, it is not possible to compare two versions of software on a higher level than the implementation graph. Hillemacher [15] has compared four different tools for architecture extraction and conformance checking and elected HUSACCT to be the best extraction tool, which is what we need here.

In his PhD thesis [9], Garcia developed the ARCADE framework. It is used to detect architectural decay, also called thrift and erosion, as well as architecture smells. During this, various recovery techniques were compared [10], whereat ACDC [34] and ARC were the most successful. ACDC clusters modules based on patterns. Patterns are, for example, if two elements are in the same package, are used together, or if they depend on the same resources. For Java, ACDC recovers the architecture from the class files. ARC is introduced by Garcia himself, clusters entities based on semantic similarity using a statistical language model based on comments and identifiers in the source code. The ARCADE framework uses these different recovery techniques to compute similarity metrics, namely a2a (architecture to architecture) and cvg (cluster coverage) [20]. The first one defines similarity based on the minimum number of steps to get from one architecture graph to the other. For example, removing one dependency or adding one node is each one step. While a2a looks at the architecture from a top-level perspective, cvg looks into the modules, or clusters, and computes if they are still the same module. For example, if all classes in a module are changed but the module name itself is untouched, it cannot still be considered to be the same module. Hence, it compares the entities inside the modules and only considers two modules equal, if at least $x\%$ of the entities are equal.

One reason for failing builds are syntax errors. Because builds with syntax errors are uncompilable, we do not want to use recovery techniques which rely on the compiled class files. Thus, we do not use the SWAG Kit or ARCADE directly. However, the metrics and the recovery technique are promising in our context. Hence, we combine the strengths of HUSAACT in extracting the structure from the source files with the recovery technique ACDC and the metrics from ARCADE. In previous studies, other researchers [10, 15] have given us confidence that the choices HUSACCT and ACDC are indeed some of the best tools, which we can use. The cvg metric is comparable to kernel based similarity whereas a2a considers change. Coupling in the architecture will be addressed with Martin's metrics. We reuse the modules from HUSACCT and ARCADE directly, while we need to implement the Martin's metrics ourself.

4 Metrics

To measure the similarity or change between two architectures, or even between designed and implemented architecture, there are several approaches, as described in Section 3. For OO software systems, as studied here, the so-called Martin's metrics [21] were introduced, i.e. afferent (Aff), efferent (Eff) coupling and instability (I). Those metrics describe the independence of modules. Coupling describes how many incoming or outgoing transitions a module has. For example, if class `Car` of module `Vehicles` implements the interface `Engine` of the module `Parts`, then `Vehicles` has an outgoing transition to `Parts`, which itself has the corresponding incoming transition. In other words, `Vehicles` depends on

[1] http://www.swag.uwaterloo.ca/javex/

`Parts`. Instability is the ratio between efferent coupling and the complete coupling, which represents if a module is highly dependent on other modules (I close to 1) or if many modules depend on this one (I close to 0).

$$I = Eff/(Eff + Aff) \tag{1}$$

In the context of architectural change, this metric can be used in the following way: If the instability changes, this module has an increase or decrease in the number of dependencies. This would not be the case, if the module has the same change in afferent and efferent coupling, but this is unlikely. Still, this metric remains a heuristic.

More sophisticated change metrics are a2a and cvg. Architecture to architecture measures the minimum amount of steps to get from the first to the second architecture divided by the sum of the steps needed to build both architectures themselves (Equation 2). A step is either the adding, removing or moving of an entity inside a module or the creation or deletion of a module.

$$a2a(A_i, A_j) = 1 - \frac{mto(A_i, A_j)}{aco(A_i) + aco(A_j)} \tag{2}$$

Where:

$$mto(A_i, A_j) = \text{steps from } A_i \text{ to } A_j$$
$$aco(A_i) = \text{steps to create } A_i$$

The cluster coverage (cvg) is a metric that measures how the inside of modules is changing. For that, the cluster to cluster (c2c) metric is calculated, which defines how many elements of the cluster overlap divided by the number of elements in the bigger cluster (Equation 3). As an example, if a module has 20 elements, and from one version to another two elements get removed and three are added, then $c2c = \frac{18}{21}$. If this ratio is above a predefined threshold, the clusters are called equal (Equation 4). The cvg metric then calculates the ratio between equal modules and the amount of modules in the architecture (Equation 5). In ARCADE the cvg is calculated with a threshold of 75% from the earlier version to the latter one (cvg_{src}) and with 66% the other way around (cvg_{tar}). These are the standard settings in ARCADE, but it is not explained why these numbers are chosen. One explanation might be that adding functionality to a module does not change its behavior as much as removing elements. Therefore, the higher threshold for cvg_{src} is accounting for this.

$$c2c(c_i, c_j) = \frac{|\text{entities}(c_i) \cap \text{entities}(c_j)|}{\max(|\text{entities}(c_i)|, |\text{entities}(c_j)|)} \tag{3}$$

$$simC(A_i, A_j) = \{c_i | c_i \in A_i, \exists c_j \in A_j) \\ (c2c(c_i, c_j) > th_{cvg})\} \tag{4}$$

Table 1. Ten Largest Projects under Study

project name	commits	analyzed	range	project type
sonarqube	11590	4573	1.25y	Code Analyzer
graylog2-server	8490	842	4.25y	Log Analyzer
okhttp	5367	1770	3.5y	HTTP Client
cloudify	4984	821	2.5y	Dev. Framework
structr	3809	1253	2.5y	Development App
owlapi	3238	2178	3.5y	OWL API
jOOQ	3196	451	3y	SQL API
checkstyle	3029	325	2y	Code Analyzer
vectorz	3025	46	3.5y	Math Library
owner	2671	2099	3.5y	Property Files API

$$cvg(A_i, A_j) = \frac{|simC(A_i, A_j)|}{|allC(A_i)|} \tag{5}$$

The problem with a2a and cvg is, that they do not explicitly take the dependencies between modules into account. This is only done while reconstruction the architecture itself, which we do with the pattern based ACDC from the ARCADE framework. To get a view of the change of the dependencies explicitly, we combine a2a and cvg with the widely accepted Martin's metrics [16], which especially account for the change in between the modules.

5 Methodology

In this study, we have analyzed 159 projects from the TravisTorrent dataset, which use Java and Maven. In the dataset are 241 projects which use Java as their main programming language and are built with Maven. We excluded projects, which do not have pushed anything to the master branch or only have one commit, because we cannot compare architectures if there is only one. Then there are some projects where builds or logs could not be retrieved anymore from GitHub and TravisCI or the toolchain was unable to reconstruct the architecture. We are left with 49, 531 commits. An exemplary list of the ten biggest analyzed projects, in terms of unrestricted commits, can be found in table 1.

As described, we see failures at or before compilation as a source of build failures. Vassallo et al. [35] classified the different fail types based on Maven goals. Because it is inefficient for us to rerun every build we analyzed for multiple goals, we will use a simpler approach. Furthermore, because some builds are already older and have dependencies which are not anymore available or need to be installed manually, the approach to rerun everything may even be impossible. Still, we base our heuristic on the Maven goals. Hence, we analyze the log files from TravisCI for all builds, to classify the build outcome further, than in success and failure. We used a heuristic to filter first lines that start with the "`[Error]`" keyword, which is used by the Maven log to indicate defects. Afterwards, we look for "Compilation failure", "dependencies" or "test failures" to categorize the failures. We map this with the actual build result to mitigate errors in the heuristic,

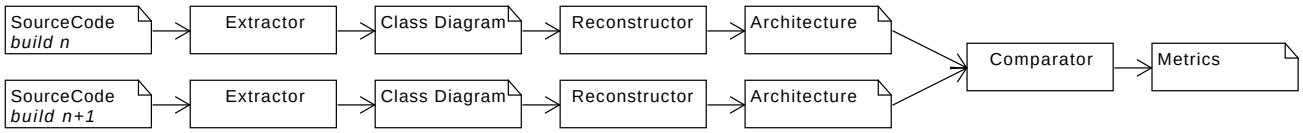


Figure 1. Overview of the process

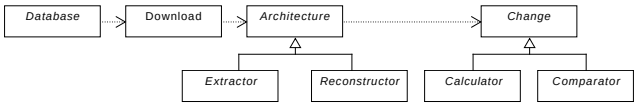


Figure 2. Structure of the Framework

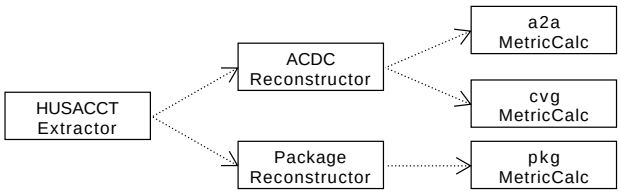


Figure 3. Implemented tool chain

so that only if the actual build failed, we use one the fail categories. We end up with five possible build results: (0) no error, (1) error during dependency resolution, (2) compilation failure, (3) test failure or (4) unknown error.

Because the build tool impacts the outcome of the build but not the architecture itself [17], we will limit the study to only one build tool. Although this introduces bias, it ensures consistent results in return. HUSACCT needs the path to the source files, so we can exclude test files and resources. Therefore, we choose the build tool maven with its convention over configuration paradigm. In this way we can be sure where everything is stored in the file structure. This eases the implementation of the class structure extraction. Because there are possibly multiple builds for the same commit, we only use the build that builds a certain version of the software first. The following builds are most likely only configuration changes of the CI server and do not introduce new change. Thus, we can ensure that changing configuration does not bias the results. Instead, we compare the changing source code based on the older configuration, i.e. the configuration which is equal to the one of the older build.

In general, to get an architectural change, we have to follow three steps. First, we need to extract the class structure out of the source code, then we reconstruct the architecture and eventually we take two architectures and compare them with each other to get a change metric (Fig. 1). The complete source code and results can be found in our GitHub repository.[2]

5.1 Framework

We implemented the extraction of the architectural change metrics between two commits using a framework (Fig. 2). First, the unique commit ID is extracted out of the TravisTorrent database, with which the snapshot can be downloaded from GitHub. The "Architecture" module is split into two parts, the extraction of the class structure and the reconstruction of the architecture. Intermediate results are saved in a file using the Rigi Standard Format (rsf) [18], because the ARCADE tools rely on this filetype. This filetype saves simple relation triples in the form: type, source, target (see listing 1). This relational system is easy convertible into a graph structure or a database and stores the modules implicitly in the dependencies. The downside of this format is that the modules cannot be extracted separately but only implicitly while analyzing the dependencies. Fortunately, this is irrelevant for our work.

Listing 1. Rigi Standard Format example

```
subPkg    checkstyle  checkstyle.grammars
subPkg    checkstyle  checkstyle.gui
contains  checkstyle  checkstyle.Main
```

Then, for every two subsequent commits, we compare the architectures and save the results in a JSON file. With the methodology structured into this framework, the research can easily be expanded with more reconstructors or metric calculators. Therefore, one needs only to implement a new subclass for the changing module, store its result in the Rigi Standard Format and it can be integrated with all existing tools. Doing so, we have combined HUSACCT and ACDC and integrated it with the ARCADE metrics. Similarly, we have implemented the package reconstruction and metrics, so that both ways of computing change can be run independent or simultaneous (Fig. 3).

5.1.1 Extraction

To extract the class structure from the source files, the extractor of the HUSACCT framework is used. For simplicity, and because it is not used in the further analyses, the dependency types are restricted to "subPkg", "contains" and "references",

[2] https://github.com/jodokae/cmput663-architecture

Table 2. Metrics

Name	Full Name	Description	Builds With Change
#V	Number of Vertexes	Number of changing modules	1.6%
#E	Number of Edges	Number of changing dependencies	3.6%
A. Inst	Absolute Instability	Change of instability for weighted dependencies	16%
R. Inst	Relative Instability	Change of instability for unweighted dependencies	3.4%
deg	Average node degree	Sum of Edges (unweighted) per module	3.1%
a2a	architecture to architecture	number of steps to get from one architecture to another	67%
cvg_{src}	cluster coverage (source)	% of changed modules based on the earlier arch	39%
cvg_{tar}	cluster coverage (target)	% of changed modules based on the latter arch	40%

where "contains" means that that a package contains a Java type and "references" is any dependency between two Java types. The results are then fed into the reconstruction.

5.1.2 Reconstruction

Two different reconstruction techniques were implemented. We chose ACDC, because it was evaluated to have one of the best reconstruction accuracy [10] in comparison with other common techniques. ACDC, in its implementation given in the ARCADE framework, is working directly on the Java byte code. Given that a prominent reason for failing builds (here: 10% of the builds) is compilation error this is problematic. Therefore, we feed the class structure from the HUSACCT extractor into ACDC and deactivate its own first level extraction.

The other technique is the package reconstruction which is often used as ground truth analysis. This technique looks for the root packages in the system. If there are only a few root packages, it is assumed that the implemented architecture lies a level deeper in the packages. For example, if there is only the "org.sonar" as root, this is just the base container and not an architectural element. Therefore, all direct children of this package are considered as modules until there are at least an acceptable number of modules. In this work, 10 was considered the minimum amount of modules needed. This number was chosen with manual sampling over different projects, so that the achieved results were the most consistent with the believed intended architecture. As dependencies all "references" edges were considered and added to their respective parent module. This is a valid approach according to Song et al [32].

5.1.3 Change Metrics

As explained in Section 4, the ARCADE metrics a2a and cvg as well as Martin's metrics are used. We calculate the ARCADE metrics on the ACDC architecture. Cvg is calculated twice, one based on the modules of the first version (called source) and one based on the second version (called target). The coupling metrics are calculated on the package architecture to have some ground truth metrics for comparison. We calculate five package metrics; thus, we have 8 metrics in total (cf. table 2). For every module, the absolute number of incoming and outgoing edges are calculated, the degree of the node (sum over ingoing and outgoing edges), as well as the absolute and relative instability. The absolute instability is based on the absolute number of connections between the modules, i.e. if Module A has only three ingoing connections from Module B, then the afferent coupling is considered three. For the relative instability, the coupling is instead considered one, so it is not measured how tightly connected two modules are, just if they are connected at all.

Since we need the change for the complete architecture, and not for each module, we calculate the mean instabilities and node degree as well as the absolute number of modules (vertexes) and dependencies (edges).

The metrics are then compared pairwise. To compare proportional metrics, like instability, the difference between the values is taken, e. g. if architecture A has an average instability of 30%, and architecture B has an average instability of 25%, then they are 5% different to each other (Equation 6). For metrics with absolute numbers, like node degree, the similarity is the proportion of the two values, e.g. A has an average node degree of 8 and B has 10, then they are 80% similar, i.e. 20% change (Equation 7).

$$c_{rel}(m_1, m_2) = \min\{m1, m2\} - \max\{m1, m2\} \tag{6}$$

$$c_{abs}(m_1, m_2) = 1 - \frac{\min\{m1, m2\}}{\max\{m1, m2\}} \tag{7}$$

5.2 Evaluation

The calculated metrics are tested for correlation with the build result. This is done in multiple ways. First, the metric is tested against the direct build outcome, then the previous or following b outcomes for $b \in \{2, 3, 5, 10\}$. This means we look for correlation between change in build x and the build result in the builds $\{x, x \pm 1, \ldots, x \pm b\}$. From this, we can see if architectural change correlates to build failures in recent history or near future. Because we have different types of failures, for testing in the near past and future, the fail types get categorized into non-failure and failure. In this way, we

Table 3. Correlation between Metrics

	#V	#E	A. Inst	R. Inst	deg	a2a	cvg_{src}	cvg_{tar}
#V	1	0.92	0.68	0.66	0.68	0.16	0.33	0.37
#E	0.92	1	0.69	0.69	0.86	0.16	0.34	0.38
A. Inst	0.68	0.69	1	0.91	0.58	0.01	0.23	0.27
R. Inst	0.66	0.69	0.91	1	0.59	0.01	0.21	0.25
deg	0.68	0.86	0.58	0.59	1	0.11	0.24	0.29
a2a	0.16	0.16	0.01	0.01	0.11	1	0.17	0.17
cvg_{src}	0.33	0.34	0.23	0.21	0.24	0.17	1	0.71
cvg_{tar}	0.37	0.38	0.27	0.25	0.29	0.17	0.71	1

get a single integer how many builds failed, instead of having multiple variables. Then, every metric is converted to a Boolean, which is true if there is a change and false if there is none. This is calculated with a threshold t, where everything below t is considered to be no change, and anything above as a change for $t \in \{0.01, 0.02, 0.03, 0.04, 0.05, 0.1, 0.2, 0.5\}$. These values were chosen because most found changes in the dataset were small (Fig. 4) We take these thresholds to find out if small changes affect the amount correlation present in the data. Every comparison which includes no Boolean variables is run with Pearson's correlation test, the others with Spearman's test.

6 Results

This section presents the findings with the proposed methodology. First, the projects under study are introduced, then calculated metrics are evaluated and, eventually, the correlation between the metrics and the build outcomes is shown.

6.1 Studied Projects

We have studied 49, 531 commits from 159 projects which use Java Maven. The pass rate is 84.9%. Around 10% of the builds failed because of errors before or during the compilation of the software. On third of the errors could be mapped to test errors. The remaining part of the fails were not further separated, but are for example configuration errors or wrong used maven plugins. The active time of the projects varies. For the ten biggest projects (in terms of number of commits) it is between one and four years of development time, some start at a well-established state (the TravisTorrent dataset starts, for example, at build number 500 for SonarQube), others at build 1 (e.g. Checkstyle). The projects vary in their application field. In the ten biggest projects, there are two code analyzers, three APIs, a web client and an IDE, some for databases, linear algebra or ontologies. Table 1 gives a detailed list.

6.2 Metric Evaluation

We have extracted 8 architecture change metrics. Most builds show no change (table 2). This is to be expected, since most changes in the system should not be architectural changes. Only a2a has more changing builds than non-changing builds. But then, most of these changes are really small, as shown in Fig. 4e. In other metrics, most changes are quite small as well, with only few big architectural changes (Fig. 4f and 4b). This seems usual, because we assume that the architecture does not have big changes that often. Only the number of nodes (Fig. 4a) has more widespread changes, which lies in the nature of the reconstruction process. Because a minimum number of ten nodes per architecture is hard coded into the package-architecture reconstruction, but is done level wise, a change from ten to nine nodes cannot appear, but will result in a change from 10 to 18 nodes. The effects of this red line are not as bad as they sound, because the node degree and number of edges metrics are in the expected norm, so that it is indicated that the few outliers do not affect the results badly.

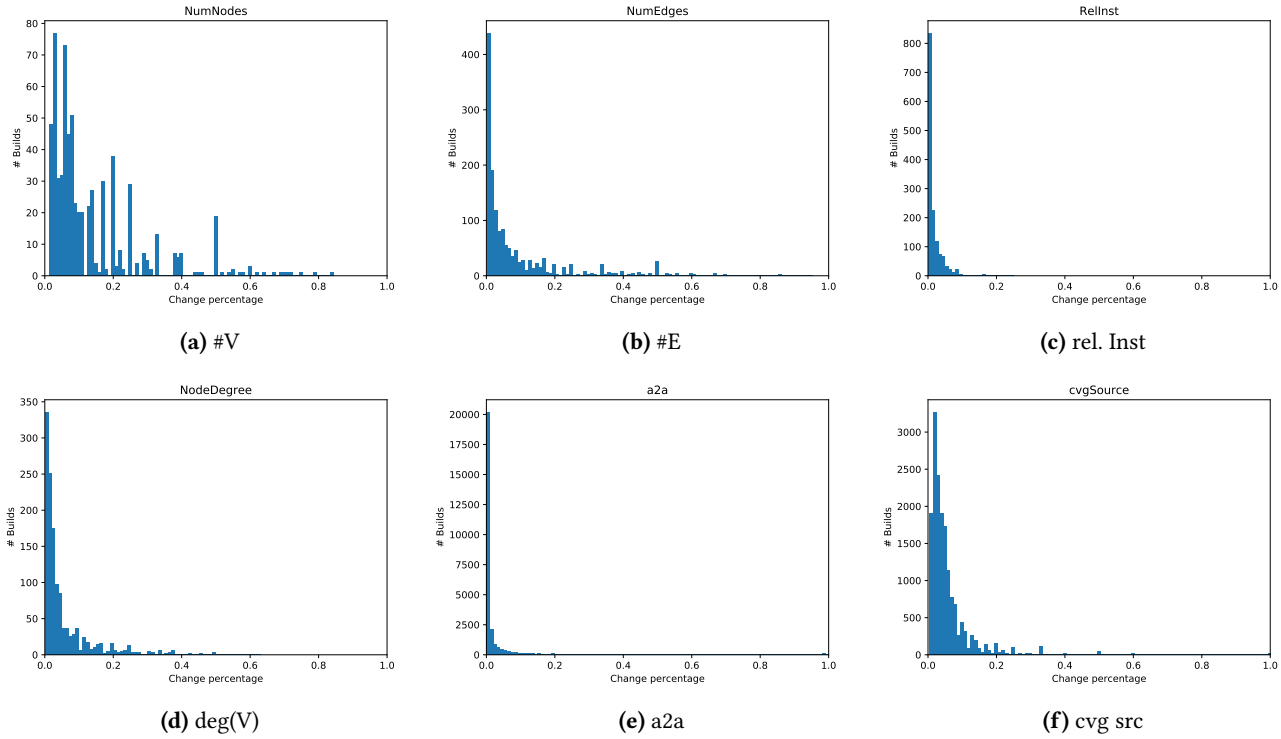
In table 3 the correlation between the metrics is shown. All p values were equal or close to 0. the highest p value cross-metric wise is 3.7^{-102}. The package metrics are highly correlated, all with c values above 0.6 up to 0.92. a2a does not correlate with the other metrics strongly, only between $0.10 - 0.17$, while cvg shows low to medium correlation to the package metrics. Still, this gives the indication that all metrics are truly an indicator for architectural change if we assume that at least one of them is, since the p-values do not leave any doubt, that the metrics are dependent. As expected, the metrics look at disjunct types of architectural change, a2a and cvg highlight the change at the module level, whereat the Martin's metrics take the focus at the relationship between the modules. This strengthens our confidence in the chosen tool chain.

6.3 Relationship

Around 16% of all builds that had architectural change failed ($P(f|c)$). This we found while extracting data for all metrics for all tested change thresholds, at the median. The values ranged from $8-18\%$. If we check for the probability of change under the condition that the build failed ($P(c|f)$), we get a range from 0% up to 60%. For the package metrics, the probability was almost always low, but the the ARCADE metrics, the results are more interesting. First, the probability drastically decreases while the change threshold increases. If we look at the probability of architectural change according to the a2a metric, and we consider every change, then we end up with 61%.. Of course, this is natural since we do not change the number of failing builds but the number of considered changes when we change the threshold of what we consider a change. Constant values around $p(f|c) = 16\%$ and peak values for $p(c|f)$ with 61% for a2a and $35 - 36\%$ for cvg, leads us to the possibility that there is also some correlation between the data.

6.4 Correlation

When we first analyzed only the ten biggest projects, we found lot of correlation values between 0.1% and 3%, with



(a) #V **(b)** #E **(c)** rel. Inst

(d) deg(V) **(e)** a2a **(f)** cvg src

Figure 4. Histograms metric over all projects

even some values as high as 11%. Now, when analyzing all projects, the feeling, that there is no correlation gets confirmed. When calculating correlation between architectural change and the different types of build outcome, some of the metrics show no correlation at all ($c < 0.4\%$ and $p > 0.25$), while the top metrics, a2a and cvg, peak at 4.3% ($p = 6.8 \cdot 10^{-22}$) and 3.0% ($p = 1.2 \cdot 10^{-11}$). The threshold that controls how many previous or following builds were considered, had no large impact. Only when going really high, so it considers more than 40 previous builds, it was possible to see a significant rise in the correlation (Fig. 5). But first, a rise from 3% to 6%, although it is a doubling, is not that interesting, and second to consider the previous 80 builds, where the peak occurred, is more by chance than by actual correlation. In other words, it is improbable that code that was committed 80 builds before the current one, influences the architecture. Nevertheless, we computed the correlation values over the next 80 builds for different change thresholds, and confirmed this educated feeling that the correlation is even lower than the one for the next 10 builds after the initial peak. The highest correlation found for a2a was the one with a change threshold of 0.0, i.e. every little change was considered as change. But even the best correlation, in a realistic scenario, was only 4% over the next ten builds (Fig. 6).

Some of the metrics have low p-values which shows that there is a statistical significant relationship between build outcome and those metrics. But this correlation is so small that we cannot conclude this as a meaningful impact. Because taking the smallest changes into consideration increases the correlation coefficient, and the threshold for previous and following builds had no big significant change on that coefficient, it must be assumed that the stronger correlations could be noise. Because the strongest correlation was found with the a2a metric, which works on the modules and not the relationship between them, the possible correlation is at the modules and within them. Furthermore, over all correlations, the tendency was towards a positive value, with 38% negative correlation values. This shows, if there truly is a correlation, then change is occurring more often around non failing builds, whereas a static architecture shows in the near of failing builds.

7 Discussion

While checking 159 Java projects, no significant correlation between architectural change and build results was found. Apart from there actually being no correlation, there could be various factors influencing this result.

First of all, only Java projects which are build with Maven were evaluated. Hence, we cannot conclude our results globally to Java or other programming languages. Still, we believe that our results are not dependent to Maven, and can be obtained with other build tools, as well.

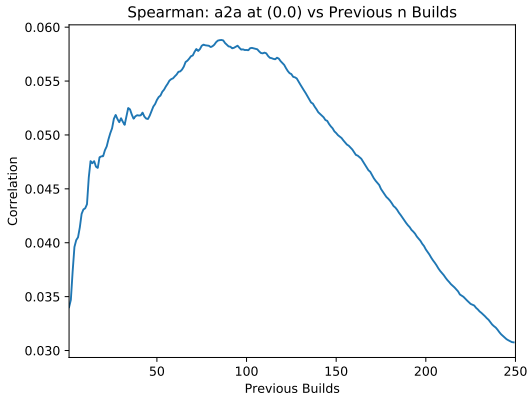


Figure 5. a2a vs previous n builds over 0.0 threshold

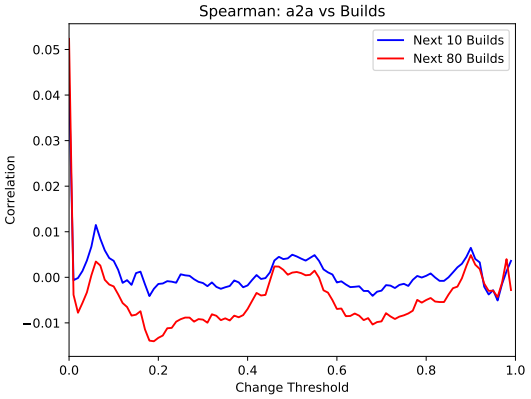


Figure 6. a2a vs next ten and eighty builds over different change thresholds

Despite having projects which start at build one, the number of failing builds was relatively small. It is possible that the missing connection between architecture and build results lies in the possibility that the big projects which have a stronger impact on the results have a well thought-through architecture and do not change it in a problematic way. This can be backed up by the fact that most found changes are below a threshold of 10% change. It would be interesting to see if projects with a more unstable architecture show more failures. We have not found interesting cross-project differences.

It is also possible, that the problems which are produced by architectural change are not transferred into build results but to a different level, e.g. (1) code coverage or (2) number failed tests. Unfortunately, the dataset does not give the opportunity to check for those, because they are (1) not reported and (b) have missing values in around 20% of the builds. The fact that software architecture is important is common knowledge. We showed that it changes over the lifetime of a project. However, it remains unknown how architecture change affects the progress of the development process.

As Garcia et al. [10] show even the best recovery techniques have problems detecting the true architecture. Therefore, it is possible that HUSACCT and ARCADE are not the right tools for the research question. This problem was addressed since we consider multiple recovery techniques and rely on studies which evaluate ARCADE as being one of the most accurate recovery tools. The produced metrics highly correlate, which strengthens the claim that the change detection is working as intended. Still, it is possible that the chosen metrics are not the best set. Because they correlate so strongly, they are a weak feature set if one would try to use machine learning techniques to predict the build outcome. For this, a broader set of metrics is necessary.

In the future, more metrics and extractors should be added to the framework. Then, it is necessary to broaden the research to find better quantitative data which show problems in the software that could be caused by bad or changing architecture. Because this work introduces an easy expandable framework, this should be possible with little to no effort.

8 Conclusion

While analyzing 49, 531 builds from 159 different Java Maven projects from various domains, we could not find any significant impact of architectural change on build outcome. We approached this result through an easily expandable framework to recover the architecture from the builds using one extractor, two reconstructors, and three metric sets based on previous researches. In total, 8 metrics were tested for correlation in 6 different scenarios including the previous and following builds.

The hypothesis that architectural change impacts CI build outcomes at or near the change was rejected. The highest correlation found was around 4% and would imply that change leads to fewer failures. This could just be noise, due to the well-established nature of the studied projects, or that is no correlation between architectural change and build outcome.

We conclude that change does not impact the build outcome and, thus, the research questions have to be changed. Since it is known that the architecture impacts the development, other quantitative features must be found to measure the impact. We propose to extend the measured metrics, because we believe that one metric cannot explain the impact alone, and search for correlation in more nonfunctional metrics, like code coverage or development pace. The built framework can be efficiently adapted for the new questions through extracting more facts out of the code using various tools.

References

[1] Adeel Ahmad, Henri Basson, Laurent Deruelle, and Mourad Bouneffa. 2009. A knowledge-based framework for software evolution control.

In *INFORSID*, Vol. 9. 286–291.

[2] D. Ameller, C. Ayala, J. Cabot, and X. Franch. 2012. How do software architects consider non-functional requirements: An exploratory study. In *2012 20th IEEE International Requirements Engineering Conference (RE)*. 41–50. https://doi.org/10.1109/RE.2012.6345838

[3] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2016. Oops, my tests broke the build: An analysis of Travis CI builds with GitHub. *PeerJ Preprints* 4 (April 2016), e1984v1.

[4] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *Proceedings of the 14th working conference on mining software repositories*.

[5] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2013. Early Detection of Collaboration Conflicts and Risks. *IEEE Transactions on Software Engineering* 39, 10 (2013), 1358–1375.

[6] Arvid Butting, Timo Herbert Greifenberg, Bernhard Rumpe, and Andreas Wortmann. 2017. Taming the Complexity of Model-Driven Systems Engineering Projects. In *Grand Challenges in Modeling 2017 : STAF 2017 - Software Technologies: Applications and Foundations* (2017-07-17). 2.

[7] A. Caracciolo, M. Lungu, O. Truffer, K. Levitin, and O. Nierstrasz. 2016. Evaluating an Architecture Conformance Monitoring Solution. In *2016 7th International Workshop on Empirical Software Engineering in Practice (IWESEP)*. 41–44. https://doi.org/10.1109/IWESEP.2016.12

[8] Wei Ding, Peng Liang, Antony Tang, and Hans Van Vliet. 2015. Understanding the Causes of Architecture Changes Using OSS Mailing Lists. *International Journal of Software Engineering and Knowledge Engineering* 25, 09n10 (2015), 1633–1651.

[9] Joshua Garcia. 2014. *A unified framework for studying architectural decay of software systems*. Ph.D. Dissertation.

[10] J. Garcia, I. Ivkovic, and N. Medvidovic. 2013. A comparative analysis of software architecture recovery techniques. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 486–496. https://doi.org/10.1109/ASE.2013.6693106

[11] Timo Greifenberg, Klaus Müller, and Bernhard Rumpe. 2015. Architectural Consistency Checking in Plugin-Based Software Systems. In *Proceedings of the 2015 European Conference on Software Architecture Workshops (ECSAW '15)*. Article 58, 7 pages. https://doi.org/10.1145/2797433.2797493

[12] Thomas Haitzer, Elena Navarro, and Uwe Zdun. 2015. Architecting for Decision Making About Code Evolution. In *Proceedings of the 2015 European Conference on Software Architecture Workshops (ECSAW '15)*. Article 52, 7 pages. https://doi.org/10.1145/2797433.2797487

[13] Tilmann Hampp and Markus Knauß. 2008. Eine Untersuchung über Korrekturkosten von Software-Fehlern [english: An Investigation on the Correcture Costs of Software Faults]. *Softwaretechnik-Trends* 28 (2008).

[14] A. E. Hassan and K. Zhang. 2006. Using Decision Trees to Predict the Certification Result of a Build. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. 189–198. https://doi.org/10.1109/ASE.2006.72

[15] Steffen Hillemacher. 2016. *Automated architecture checking for MontiCore-based software development projects*. Master's thesis. RWTH Aachen University.

[16] Sami Hyrynsalmi and Ville LeppÃđnen. 2009. A Validation of Martin's Metric. In *Proceedings of 11th Symposium of Programming Languages and Software Tools and 7th Nordic Workshop on Model Driven Software Engineering*. 87–101.

[17] M. R. Islam and M. F. Zibran. 2017. Insights into Continuous Integration Build Failures. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 467–470. https://doi.org/10.1109/MSR.2017.30

[18] Holger M. Kienle and Hausi A. Müller. 2010. Rigi - An Environment for Software Reverse Engineering, Exploration, Visualization, and Redocumentation. *Sci. Comput. Program.* 75, 4 (April 2010), 247–263. https://doi.org/10.1016/j.scico.2009.10.007

[19] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic. 2015. An Empirical Study of Architectural Change in Open-Source Software Systems. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. 235–245. https://doi.org/10.1109/MSR.2015.29

[20] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic. 2015. An Empirical Study of Architectural Change in Open-Source Software Systems. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. 235–245. https://doi.org/10.1109/MSR.2015.29

[21] Robert Martin. 1994. OO design quality metrics. *An analysis of dependencies* 12 (1994), 151–170.

[22] T. Nakamura and V. R. Basili. 2005. Metrics of Software Architecture Changes Based on Structural Distance. In *11th IEEE International Software Metrics Symposium (METRICS'05)*. 24–24. https://doi.org/10.1109/METRICS.2005.35

[23] A. Ni and M. Li. 2017. Cost-Effective Build Outcome Prediction Using Cascaded Classifiers. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 455–458.

[24] Ana Nicolaescu, Horst Lichter, Artjom Göringer, Peter Alexander, and Dung Le. 2015. The ARAMIS Workbench for Monitoring, Analysis and Visualization of Architectures Based on Run-time Interactions. In *Proceedings of the 2015 European Conference on Software Architecture Workshops (ECSAW '15)*. Article 57, 7 pages. https://doi.org/10.1145/2797433.2797492

[25] K. V. R. Paixão, C. Z. Felício, F. M. Delfim, and M. D. A. Maia. 2017. On the Interplay between Non-Functional Requirements and Builds on Continuous Integration. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 479–482. https://doi.org/10.1109/MSR.2017.33

[26] Matheus Paixao, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman. 2017. Are Developers Aware of the Architectural Impact of Their Changes?. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 95–105. http://dl.acm.org/citation.cfm?id=3155562.3155578

[27] Leo Pruijt and Jan Martijn E. M. van der Werf. 2015. Dependency Types and Subtypes in the Context of Architecture Reconstruction and Compliance Checking. In *Proceedings of the 2015 European Conference on Software Architecture Workshops (ECSAW '15)*. Article 56, 7 pages. https://doi.org/10.1145/2797433.2797491

[28] Leo J. Pruijt, Christian Köppe, Jan Martijn van der Werf, and Sjaak Brinkkemper. 2014. HUSACCT: Architecture Compliance Checking with Rich Sets of Module and Rule Types. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. 851–854. https://doi.org/10.1145/2642937.2648624

[29] M. RebouÃğas, R. O. Santos, G. Pinto, and F. Castor. 2017. How Does Contributors' Involvement Influence the Build Status of an Open-Source Software Project?. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 475–478. https://doi.org/10.1109/MSR.2017.32

[30] K. Rostami, R. Heinrich, A. Busch, and R. Reussner. 2017. Architecture-Based Change Impact Analysis in Information Systems and Business Processes. In *2017 IEEE International Conference on Software Architecture (ICSA)*. 179–188. https://doi.org/10.1109/ICSA.2017.17

[31] B. Rumpe, M. Schoenmakers, A. Radermacher, and A. Schurr. 1999. UML+ROOM as a standard ADL?. In *Engineering of Complex Computer Systems, 1999. ICECCS '99. Fifth IEEE International Conference on*. 43–53. https://doi.org/10.1109/ICECCS.1999.802849

[32] X. Song, S. Wang, M. Zhao, and Y. Wang. 2010. Complex network characters in object-oriented software desigh. In *2010 2nd IEEE InternationalConference on Network Infrastructure and Digital Content.*

953–957. https://doi.org/10.1109/ICNIDC.2010.5657937

[33] S. A. Tonu, A. Ashkan, and L. Tahvildari. 2006. Evaluating architectural stability using a metric-based approach. In *Conference on Software Maintenance and Reengineering (CSMR'06)*. 10 pp.–270. https://doi.org/10.1109/CSMR.2006.26

[34] V. Tzerpos and R. C. Holt. 2000. ACDC: an algorithm for comprehension-driven clustering. In *Proceedings Seventh Working Conference on Reverse Engineering*. 258–267. https://doi.org/10.1109/WCRE.2000.891477

[35] C. Vassallo, G. Schermann, F. Zampetti, D. Romano, P. Leitner, A. Zaidman, M. D. Penta, and S. Panichella. 2017. A Tale of CI Build Failures: An Open Source and a Financial Organization Perspective. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 183–193. https://doi.org/10.1109/ICSME.2017.67