An Empirical Study of Architectural Changes in Code Commits

Di Cui
School of Computer Science and Technology,
Xi'an Jiaotong University, Xi'an 710049, China
cuidi@stu.xjtu.edu.cn

Jiaqi Guo
School of Automation Science and Engineering,
Xi'an Jiaotong University, Xi'an 710049, China
jasperguo2013@stu.xjtu.edu.cn

Ting Liu
School of Cyber Science and Engineering,
Xi'an Jiaotong University, Xi'an 710049, China
tingliu@mail.xjtu.edu.cn

Qinghua Zheng
School of Computer Science and Technology,
Xi'an Jiaotong University, Xi'an 710049, China
qhzheng@mail.xjtu.edu.cn

ABSTRACT

The maintenance of software architecture is challenged by fast-delivery code changes since developers are rarely aware of the architectural impacts of their code changes. To ease the burdens of architects, in this work, we proposed a light-weight framework to identify changes in architectures from code commits automatically. The framework identifies architectural changes without heavy architecture recovery techniques. Instead, it only takes a code commit as input. The framework, on the one hand, can be integrated into prevalent continuous integration systems to monitor architectural changes. On the other hand, it can be plugged into code review systems to help developers realize the architectural changes they introduce. Based on the framework, we further conducted a large-scale empirical study on 368,847 commits of 16 Apache open projects to study architectural changes. Our study reveals several new findings regarding the frequency of architectural change commits, the common and risky intents under which developers introduce architectural changes, and the correlations of architectural changes with lines of code and number of modified source files in commits. Our findings provide practical implications for software contributors and shed light on potential research directions on architecture maintenance.

CCS CONCEPTS

• **Software and its engineering → Software Architecture**.

KEYWORDS

Software Quality, Software Architecture, Empirical Study

ACM Reference Format:
Di Cui, Jiaqi Guo, Ting Liu, and Qinghua Zheng. 2020. An Empirical Study of Architectural Changes in Code Commits. In *12th Asia-Pacific Symposium on Internetware (Internetware'20), May 12–14, 2021, Singapore, Singapore.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3457913.3457924

Internetware'20, May 12–14, 2021, Singapore, Singapore

ACM ISBN 978-1-4503-8819-1/20/11…$15.00
https://doi.org/10.1145/3457913.3457924

1 INTRODUCTION

Modern software systems undergo code changes on a day-to-day basis. The fast delivery and large volume characteristics of code changes pose critical challenges for the maintenance of software: code understanding, continuous integration, architecture decay, and so on. Of particular interest to us is the architecture decay, which is a class of problems that can lead to increased maintenance effort. It is usually caused by the introduction of architectural changes that have not been carefully considered by the systems' developers [28].

The importance of architectural design decisions and the downsides of architecture decay have long been recognized in the research community [28, 40]. But as pointed out in recent studies, developers, in most cases, are not aware of the architectural impact of their code changes, and they also rarely discuss architectural impact during code reviews [40, 53]. As a result, a system's architecture is likely to degenerate as the system evolves.

Under the circumstances, architects need to identify a system's architectural changes from code changes, analyze its impact, and refactor the architecture when it is necessary. However, considering the fast delivery characteristic of code changes, architects have to do these tedious and time-consuming jobs much more frequently. Besides, a lack of thorough understanding of architectural changes on a day-to-day basis, e.g., how often do developers introduce architectural changes, also makes architects struggle in practice. To ease the burden of architects, numerous techniques have been proposed in the community, e.g., architecture recovery, architectural smells detection, architecture quality measurement, and so on. However, to the best of our knowledge, no existing work addresses the identification of architectural changes from developers' code changes and locates them in code.

In this paper, we proposed a light-weight architectural change analysis framework to identify architectural changes from code commits automatically. Overall, rather than recovering and comparing architectures of a system before and after a code commit, our framework identifies changes in structure dependencies of code components from code difference of a commit. Recent studies have revealed that structural dependency is one of the best proxies for developers' perception of cohesion and coupling [9, 12]. Hence, changes in structural dependencies will lead to the change of cohesion and coupling, ultimately causing architectural changes. Specifically, taken a code commit as input, our framework operates in two phases to identify architectural changes. In the first phase, it

constructs a *change graph*, a multi-grained code differences representation that we propose, from the input commit. Then, based on the change graph, we adopt *Design Structure Matrix* [50], a state-of-the-art technique for software structural analysis, to identify changes in structure dependencies. In what follows, we use the term *structural changes* and *architectural changes* interchangeably to denote changes in structure dependencies. This light-weight analysis framework, on the one hand, can be easily integrated into prevalent continuous integration systems (CI) to monitor architectural changes on a day-to-day basis. On the other hand, it can be deployed on a code review system to facilitate the code review process, e.g., highlighting the places where architectural changes are introduced to attract developers' attention during the discussion.

Based on this framework, we further conducted a large-scale empirical study on architectural changes. The study results can largely extend the body of our empirical knowledge regarding architectural changes in software systems and shed light on potential research directions in architecture maintenance. To be more specific, we collected 368,847 code commits from 2,571 releases of 16 Apache open source projects. We then applied our architectural change analysis framework on these commits to identify architectural changes. Finally, we conducted a thorough qualitative and quantitative analysis on architectural changes. Our study contributes to the following findings:

(1) On average, there are 53.4% of commits (196,918 out of 368,847) introducing architectural changes. Developers contribute 5.6 code commits per day, among which 3.0 commits introduce architectural changes. This finding reveals that architectural changes are frequently introduced on a day-to-day basis.
(2) Bug fixing is the most common intent when developers introduce architectural changes for having the largest number of architectural changes. However, when developers implement and improve the features of a system, they have the highest probabilities to introduce architectural changes.
(3) Lines of code and number of source files in code commits do not exhibit a strongly positive correlation with architectural changes. This finding deviates from our empirical knowledge that commits that involve a large number of source files or lines of code prone to cause architectural changes.

Our findings can provide implications for software contributors and the research community. Since architectural changes are intensively introduced on a day-to-day basis, it is imperative for architects to keep monitoring architectural changes and measure their impact regularly. In addition, it is necessary for developers to pay more attention on architectural changes in daily activities, especially when they intend to fix a bug or improve a feature. In terms of the research community, considering a large number of architectural changes in a system, it is impractical to warn architects every other commit that changes architecture and to require architects to evaluate its impact. To this end, two natural questions raise: 1) When should architects evaluate the compounding effect of continuous architectural changes? 2) Are existing architectural change impact measures ready to be used in practice to alleviate the burden of architects? The answers to these questions are highly valuable for the research community.

2 ARCHITECTURAL CHANGE ANALYSIS FRAMEWORK

In this section, we present our two-phase architectural change analysis framework. Figure 1 illustrates the workflow of it. Taken a code commit as input, the framework first constructs a multi-grained representation of code changes, named *Change Graph*, from the commit. With a change graph, we can easily analyze code changes in multiple levels (e.g., statement, variable, and method). Then, the framework leverages the Design Structure Matrix technique [50], or DSM for short, to locate changes in structural dependencies, on the change graph.

In what follows, we first declare a few important definitions that are used throughout the paper. Then, we elaborate on the details of the framework with a running example.

Definition 1 (**Code Commit**). *A code commit can be modeled as a tuple with its hash ID and a set of modified files FileSet:*

$$Commit = \langle ID, FileSet \rangle \tag{1}$$

$$FileSet = \left\{ \left(f_1^0, f_1^1 \right), \cdots, \left(f_N^0, f_N^1 \right) \right\} \tag{2}$$

where N denotes the number of modified files, and for a file f_j, $\left(f_j^0, f_j^1\right)$ denotes the file before and after the commit.

Definition 2 (**Change Graph**). *A change graph is a multi-grained representation of code changes in a commit. It is a directed graph, where vertexes are change units $\{CU_i\}$ in a commit and edges are change relations $\{CR_j\}$ defined over $\{CU_i\}$.*

Definition 3 (**Change Unit**). *A change unit CU represents a modification in a source file:*

$$CU = \langle file, operation, ctype, bline, eline, cname \rangle \tag{3}$$

$$operation \in \{Insert(t_1), Update(t_2), Delete(t_3)\}, \tag{4}$$

where file denotes the source file in which a modification is introduced, and operation denotes the type of operation applied in the modification. ctype represents the type of code that are modified. ctype ranges over declaration, statement and expression, among which expression has more than 90 subtypes [2]. bline and eline represent the start line number and end line number of the modification in the source file, respectively. cname represents the identifier of a change unit. It exists only if the ctype of this change unit is declaration.

Definition 4 (**Change Relation**). *A change relation CR is defined as the relation between two change units. We consider three types of relations that have been shown to be effective in modeling relations in code changes [23]:*

- ***Contain**, referred as **Con**, indicates that change unit b is contained in unit a.*
- ***Def-use**, referred as **DU**, indicates that change unit b invokes the definition of the unit a. In most cases, the ctype of a is the declaration of variable, field or method.*
- ***Replace**, referred as **Rep**, denotes that change unit b is committed to replace the unit a.*

Definition 5 (**Structural Change**). *A structural change is defined as a change in structural dependencies among code components.*

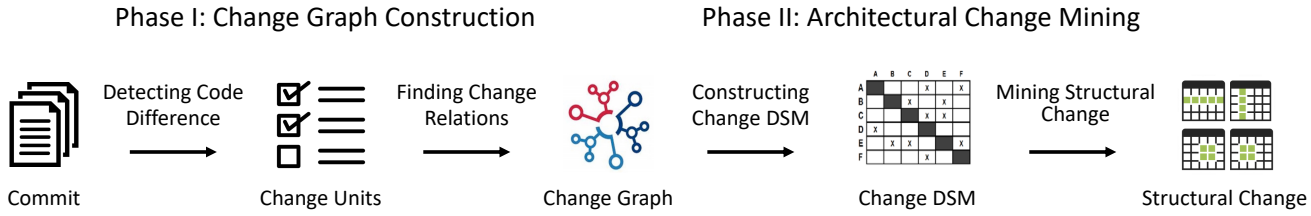


Figure 1: The Workflow of Architectural Change Analysis Framework.

A structural change is introduced if there exists change relations between a pair of change units that occur in different source files. These change relations will ultimately cause changes in the cohesion and coupling of the software system, and consequently, lead to changes in the architecture [9, 40].

2.1 Phase I: Change Graph Construction

Taken a code commit $Commit = \langle ID, FileSet \rangle$ as input, the goal of this phase is to construct a change graph. Specifically, we first detect all code changes from the commit to construct change units, and then we explore all relations between each pair of change units.

To detect code changes, we leverage *GumTree* [5], a state-of-the-art code difference detection tool. Concretely, for each file $\left(f_i^0, f_i^1\right)$ in $FileSet$, *GumTree* parses f_i^0 and f_i^1 into their corresponding abstract syntax trees AST_0 and AST_1, respectively. Next, it finds out the differences in the two ASTs. The output of *GumTree* is a set of edit actions. An edit action $EA_k^{(i,j)}$ is defined as follows:

$$EA_k^{(i,j)} = \{op, (n_0, p_0), (n_1, p_1)\}, \tag{5}$$

where op denotes the type of action on AST_0, including add, delete, update and move. (n_0, p_0) denotes the node in AST_0 that receives the action and its position in the tree. Accordingly, (n_1, p_1) denotes the node in AST_1 that is affected by the action and its position. If the action is add, (n_0, p_0) comes to be $(\emptyset, \emptyset)$. Similarly, if the action is delete, (n_1, p_1) becomes $(\emptyset, \emptyset)$.

Next, we construct change units from a set of edit actions with *Spoon* [41], a tool for source code analysis and transformation according to the work of Falleri et al. [17]. For a source file, *Spoon* can parse it into a set of concise program units. Thus, we iteratively examine each changed AST:(n_i, p_i) in edit actions and map it into parsed program units with *Spoon* based on its AST position: p_i. As a result, we obtain a set of change units with *Spoon*.

At last, we enumerate each pair of change units to find out relations between them. Specifically, for a pair of change units, we determine whether there exists a relation between them based on the code element API of Spoon (*CtElement*). Based on the reference API of Spoon (*CtReference*), we further classify the relation into types (*Contain*, *Def-use*, and *Replace*). To this end, we construct a change graph from the input commit.

2.2 Phase II: Architectural Change Mining

In this phase, we aim to systematically mine structural changes on a change graph. To do so, we leverage the design structure matrix (DSM) technique to analyze the relation between change units in a change graph. A design structure matrix (DSM) is a square matrix in which rows and columns are labeled by the same list of elements. In this context, elements are all change units in the change graph. A marked cell in row x and column y, $cell(x, y)$, means that the element in row x depends on the element in column y. The mark in cells can denote different types of relations between two elements. Thus, we can exhaustively examine all the relations between change units and detect a set of suspect change relations as structural changes:

$$SCR\,Set = \{\langle CU_a, CU_b \rangle \mid CU_a.file \neq CU_b.file, a \neq b\}, \tag{6}$$

where CU_a and CU_b denotes two individual change units. If they have a change relation and belong to different files, they are defined as a suspect change relation in structural change.

Given a commit, if it has a not empty suspect change relation set, it is diagnosed with structural changes. Consequently, we term this commit as a *structural change commit* or an *architectural change commit*. Notably, the framework does not need to recover the architectures of a system before and after code changes. This technique does require the ASTs before and after the commit of the modified program to compute the relations between change units using Spoon. Hence, it is light-weight enough to be adopted in practice to detect architectural changes.

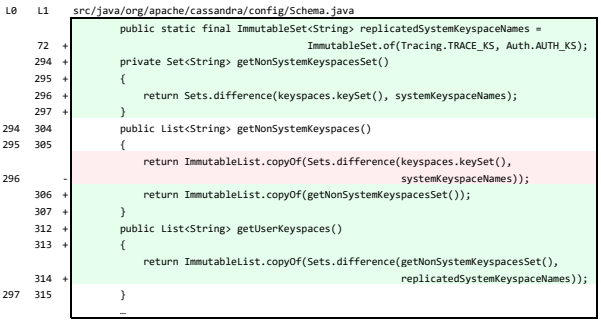


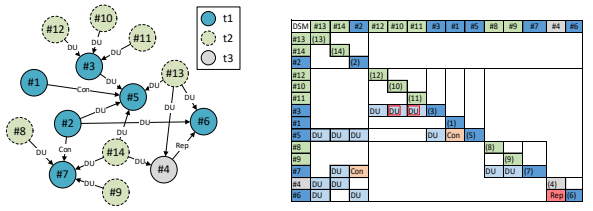
Figure 2: Code Changes in Commit f4b21f5 from Cassandra.

2.3 Running Example

Figure 2 visualizes changes of *Schema.java* in a code commit with ID f4b21f5 from *Cassandra* [1]. These changes involve that 9 lines of code are inserted and 1 lines of code are deleted.

Table 1: The Change Units in Schema.java. The column 'index' represents the identifier for a change unit and other columns represent attributes of change unit defined in Definition 3.

index	operation	ctype	file	bline	eline	cname
#1	t_1	Method	Schema.java	312	312	org.apache.cassandra.config.Schema.getUserKeyspaces
#2	t_1	Method	Schema.java	294	294	org.apache.cassandra.config.Schema.getNonSystemKeyspacesSet
#3	t_1	Field	Schema.java	72	72	org.apache.cassandra.config.Schema.replicatedSystemKeyspaceNames
#4	t_3	MethodInvocation	Schema.java	296	296	*
#5	t_1	Return	Schema.java	314	314	*
#6	t_1	VariableAccess	Schema.java	306	306	*
#7	t_1	Return	Schema.java	296	296	*
#8	t_2	Field	Schema.java	59	59	org.apache.cassandra.config.Schema.keyspaces
#9	t_2	Field	Schema.java	71	71	org.apache.cassandra.config.Schema.systemKeyspaceNames
#10	t_2	Field	Auth.java	56	56	org.apache.cassandra.auth.Auth.AUTH_KS
#11	t_2	Field	Trace.java	54	54	org.apache.cassandra.tracing.Tracing.TRACE_KS
#12	t_2	Method	*	*	*	com.google.common.collect.ImmutableSet.of
#13	t_2	Method	*	*	*	com.google.common.collect.ImmutableList.copyOf
#14	t_2	Method	*	*	*	com.google.common.collect.Sets.difference



(a) Change Graph **(b) Change DSM and Stuctural Change**

Figure 3: The Change Graph, Change DSM and Structural Change in *Schema.java*

Change Graph Construction. In this phase, the framework constructs a change graph for an input commit. Specifically, it first obtains all change units in the commit with tools GumTree and Spoon. For changes of *Schema.jave* in Figure 2, it successfully obtains 14 change units. The *cname* of #4 - #7 are marked with '*', because their *ctype* are not declaration. The $file$ of #12 - #14 are marked with '*' for these units are located in third party library and they are introduced as API usage. Then, our framework identifies change relations between each pair of change units. As a result, for the 14 change units in *Schema.java*, 16 change relations are identified, including 13 instances of *Def-use*, 2 instances of *Contain*, and 1 instance of *Replace*. Having identified change units and change relations, we are ready to construct a change graph for this commit. Figure 3(a) visualizes the change graph of changes in *Schema.java*. Vertexes (Change units) in different colors represent different types of operation. The text in each vertex refers to its index in Table 1, and the text on each edge refers to different change relations in Definition 4.

Architectural Change Mining. In this phase, the framework mines structural changes from the constructed change graph with the design structure matrix technique. The 14 change units in Figure 3(a) can also be analyzed into a change DSM, as is shown in Figure 3(b). Essentially, a change DSM is a square matrix in which rows and columns are labeled by the same list of change units in the change graph. The mark in cells can denote different types of relations between two elements. For example, the *cell*(#3, #10) in Figure 3(b), is marked with "DU", which means that change unit #10 has the Def-use relation with the unit #3. A cell with marked with rectangle means that there is a change relation between two change units and the two change units involve two different source files. As we have defined above, such interactions between change units induce structural changes. We can further two instances of change relation involve structure changes. To this end, the input commit is finally regarded as an architectural change commit.

3 EMPIRICAL STUDY

Based on the architectural change analysis framework, we conduct a large-scale empirical study on architectural change commits in popular software systems to understand their characteristics. To this end, we explore the following four research questions:

RQ1: How often do developers introduce architectural change commits? Intuitively, architectural changes should be carefully and seldom made by developers, as they have long-term impact on software systems. In this research question, we test this hypothesis by examining the number of architectural change commits in software systems. Moreover, by studying the introduction frequency, we can reveal architectural change commits are introduced on a day-to-day basis or they are introduced in a concentrated period. The answer to this question can advance our empirical understanding about architectural changes, and it can show the necessity of timely architectural changes detection.

RQ2: What are common and risky intents under which developers introduce architectural changes? This research question investigates the most common intents of developers under which architectural changes are introduced, and the most risky intents under which developers have the highest probability to introduce architectural changes. The answer to this question can extend our empirical knowledge of the impact of developers' intents on architectures.

RQ3: Are modified numbers of lines of code in code commits practical indicators of architectural changes? Lines of code in commits is one of the important features to characterize commits in change prediction approaches [21, 31, 32, 44, 47], as intuitively, it can reflect the complexity behind changes. In this research question, we aim at investigating the correlation between lines of code in commits and architectural changes. The answer to this question reveals whether the large number of code is a symptom for architectural changes.

Table 2: The Information of Selected Subjects. 'Time Span (Months)' indicates how many months that each subjects' commits span over. '#Cmt' and '#StrCmt' indicate the number of commits and architectural change commits.

Subject	Time Span (Months)	#Cmt	#StrCmt
Camel	3/2007 to 1/2019 (142)	43,789	26,825 (61.3%)
Cassandra	3/2009 to 1/2019 (118)	15,008	8,306 (55.3%)
Cxf	4/2008 to 1/2019 (129)	28,969	16,779 (58.0%)
Hadoop	1/2006 to 1/2019 (156)	18,540	10,750 (58.0%)
Hbase	4/2007 to 1/2019 (141)	20,404	4,094 (20.1%)
Hive	9/2008 to 1/2019 (124)	61,193	36,627 (59.9%)
Log4j2	5/2010 to 1/2019 (104)	41,848	24,217 (57.9%)
Solr	9/2001 to 1/2019 (208)	15,262	10,206 (66.9%)
Mahout	1/2008 to 9/2018 (132)	8,070	5,715 (70.8%)
Nifi	12/2014 to 1/2019 (49)	15,790	6,108 (38.7%)
Wicket	9/2004 to 1/2019 (172)	10,299	4,427 (43.0%)
Zookeeper	5/2008 to 1/2019 (128)	3,629	2,126 (58.6%)
Groovy	8/2003 to 1/2019 (185)	4,825	3,124 (64.8%)
Karaf	7/2005 to 1/2019 (162)	47,001	23,608 (50.3%)
Kafka	8/2011 to 1/2019 (89)	31,255	10,804 (34.6%)
Flink	12/2010 to 1/2019 (97)	2,965	1,731 (58.8%)
Total	49 to 208 Months	368,847	196,918 (53.4%)

RQ4: Are involved numbers of source files in code commits practical indicators of architectural changes? Similar to lines of code, the number of modified files in a commit is also an important feature in prevalent change prediction approaches [21, 31, 32, 44, 47]. Hence, in this question, we aim to investigate the correlation between the number of modified files and architectural changes. The answer to this question sheds light on whether the large number of modified files is a symptom for architectural change.

3.1 Data Collection

As we need systems with a large number of code changes, the number of code commits is the overarching criteria used when selecting subject systems for investigation. Moreover, since the Spoon tool used in our framework can only be applied to Java systems, we only select those systems that are primarily written in Java. To this end, we select the 16 most starred Apache open source projects as our study subjects. Notably, these 16 projects cover different domains, and they have been widely used in architecture studies [16, 28]. For each selected subject, we collect code commits from its version control system (Git [3]) by using git log [4] command. We additionally use *JGit* [6], a lightweight programming interface to manipulate *Git*, to automatically check out modified files from each commit. As a result, we collect 368,847 code commits in total, and we use all of them in our study. Table 2 presents the details of each selected subjects and the number of commits we collected for each subject.

3.2 RQ1: How often do developers introduce architectural change commits?

3.2.1 Setup. In this research question, we investigate the number of architectural change commits in a software system and the introduction frequency of them. To answer the question, we analyze all 368,847 commits using our analysis framework and identify architectural change commits from them. To study the frequency, we examine how many architectural change commits are introduced in each day, which is denoted as **architectural change frequency**. As a comparison, we also examine the number of commits introduced in each day, denoted as **code change frequency**. Moreover, we compute the Pearson correlation coefficient between them to understand their correlation.

3.2.2 Results. The column '#StrCmt' in Table 2 presents the number of architectural change commits that are identified by our analysis framework for each subject. Overall, there are 53.4% of code commits (196,918 out of 368,847) introducing architectural changes. Such a large proportion of architectural change commits in a system is surprising, as we may expect that architectural changes are carefully and seldom made by developers. Considering the large proportion of architectural changes, it is highly imperative to monitor architectural changes to effectively avoid architecture decay.

The column 'Commit per Day' in Table 3 presents the code change frequency and architectural change frequency. On average, developers contribute 5.6 commits per day, and 3.0 of them are architectural change commits. This observation means that architectural change commits are highly likely to be introduced on a day-to-day basis instead of in a concentrated period. The Pearson correlation coefficient between them is 0.85, which also indicates that they are strongly correlated.

As an answer to RQ1, by inspecting the proportion and frequency of architectural change commits, we find that developers intensively introduce architectural changes on a day-to-day basis.

3.3 RQ2: What are common and risky intents under which developers introduce architectural change commits?

3.3.1 Setup. In this research question, we aim to understand developers' intents under which they introduce architectural changes. Given the large number of commits we study (368, 847), it is infeasible to manually inspect each of them to determine developers' intents. But we observe that lots of commits refer to corresponding reports in issue tracking systems via issue IDs, and each report is assigned a type to indicate its purpose. (e.g., *Bug*, *Improvement*, and *New Feature*). Intuitively, commits for different types of reports exhibit different intents of developers. For example, when a developer introduces a commit that refers to a report with type *Bug*, it is highly likely that he/she aims to fix the bug described in the report. Such an intuition has already formed the basis of empirical studies on bug fixing [49, 54]. Therefore, we regard the report type as a proxy for developers' intent behind a commit. For all 368, 847 commits, we attempt to link each of them to its corresponding report and extract the report type. In Jira [7], each report is assigned a unique issue number following a "name-number" pattern where name represents the project/component name. We link reports by heuristically searching commit messages with issue numbers following the work of Zhong et al [54]. If a commit can be linked with multiple reports, we select the major one according to sentence structure of commit message using natural language processing techniques. As a result, there are 68% of commits successfully linked to their corresponding reports. In the following intent analysis, we only consider these 68% commits. In this study, we mainly study 5

Table 3: The Experimental Results of RQ2-RQ4. 'Cmt' and 'StrCmt' are short for commits and architectural chagne commits, respectively. 'pc' indicates the Pearson correlation coefficient. 'τ' indicates the Kendall-τ correlation coefficient.

Subject	Intent of StrCmt (%)						LOC per Commit							Commit per Day			File per Commit			
	Bu.	Im.	Ne.	Ta.	Te.	Ot.	t_1	t_2	t_3	Cmt	StrCmt	τ	AUC	Cmt	StrCmt	pc	Cmt	StrCmt	τ	AUC
Camel	36	41	14	8	1	0	90	33	36	159	92	0.49	0.85	10.2	6.3	0.89	6.5	6.0	0.45	0.79
Cassandra	54	31	8	5	1	0	96	70	61	227	127	0.53	0.87	4.3	2.8	0.85	5.8	5.5	0.55	0.82
Cxf	60	27	7	5	0	0	97	59	46	202	117	0.48	0.85	7.4	4.3	0.98	5.9	5.6	0.50	0.80
Hadoop	48	19	2	27	2	1	162	78	68	308	163	0.49	0.85	13	7.7	0.94	6.2	5.8	0.47	0.80
Hbase	58	22	3	15	2	0	135	82	69	286	157	0.51	0.86	9.8	5.7	0.88	5.9	5.7	0.54	0.82
Hive	65	14	2	18	1	0	333	223	162	717	402	0.47	0.85	4.5	3.0	0.97	5.9	5.8	0.46	0.79
Log4j2	46	36	11	2	0	4	78	27	40	145	80	0.45	0.82	3.2	1.4	0.81	5.1	4.6	0.42	0.72
Solr	40	27	5	7	2	18	120	81	63	264	140	0.47	0.83	8.2	4.1	0.88	5.3	4.8	0.49	0.78
Mahout	46	41	9	3	0	1	150	81	103	334	181	0.51	0.86	1.1	0.6	0.83	7.4	6.2	0.44	0.77
Nifi	52	36	5	6	0	0	238	119	108	465	260	0.50	0.87	3.8	2.2	0.89	6.2	5.8	0.50	0.82
Wicket	60	30	4	4	0	1	68	36	44	148	80	0.41	0.81	6.0	2.0	0.75	5.0	4.3	0.55	0.81
Zookeeper	66	20	2	9	3	0	174	82	77	333	200	0.54	0.89	0.7	0.4	0.75	6.6	6.2	0.56	0.84
Groovy	82	13	4	1	0	0	67	67	42	176	85	0.30	0.76	4.9	1.0	0.61	4.2	3.4	0.47	0.76
Karaf	45	36	14	2	1	2	133	46	76	255	125	0.40	0.79	3.3	1.3	0.76	5.1	4.4	0.53	0.79
Kafka	65	17	1	15	1	0	139	103	61	303	182	0.50	0.88	2.8	1.8	0.92	6.6	6.9	0.42	0.79
Flink	53	28	4	13	1	0	188	90	114	392	228	0.47	0.83	5.8	3.4	0.93	6.2	6.0	0.44	0.77
Average	55	27	6	9	1	2	142	80	73	295	164	0.47	0.84	5.6	3.0	0.85	5.9	5.4	0.49	0.79

Table 4: Report Type Definition.

Type	Description
Bug	A defect in source code is founded.
Improvement	An improvement to an existing feature or task.
New Feature	A new feature which is yet to be developed.
Task	A task that needs to be done to achieve team's goal.
Test	An integration of test code.
Other	The other report types defined in the issue system.

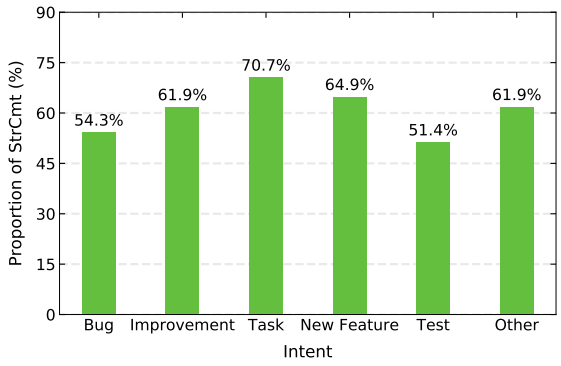


Figure 4: The Proportion of Architectural Change Commits in Different Intents.

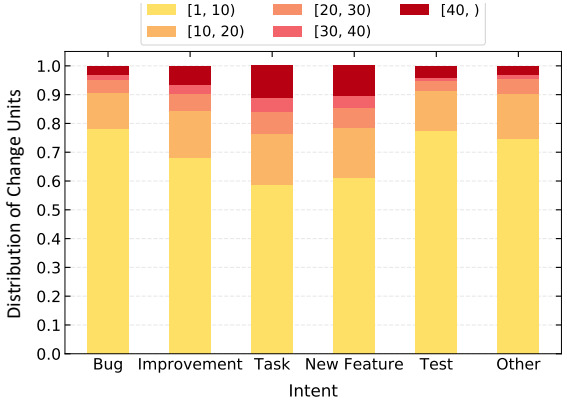


Figure 5: The Distribution of Change units in Different Intents.

standard report types defined in Jira, the issue tracking system of all study subjects, namely, *Bug*, *Improvement*, *New Feature*, *Task*, and *Test*. Other report types (e.g., *Documentation*, *Brainstorming*, and *Dependency Update*) are all categorised in *Other*. Table 4 provides a detailed definition for each of them. Through a careful examination of reports with type *Task*, we find that in most cases, they either require implementing a feature of a system or improve a feature.

3.3.2 Results. Overall, 55% of commits are introduced when developers try to fix bugs (*Bug*), followed by *Improvement* (27%), *Task* (9%), *New Feature* (6%), *Test* (1%), and *Other* (2%). The column 'Intent of StrCmt' in Table 3 reports the distribution of developers' intents in all architectural change commits. We observe that the distribution is nearly the same in all studied subjects, where most of them (55%) are introduced for bug fixing.

To gain a deeper understanding, we break down commits by intents and for each intent, we calculate how many commits introduce architectural changes. As shown in Figure 4, *Task*, *New Feature*, and *Improvement* have a significantly larger proportion of architectural change commits than that of the other three intents. Such a finding, on the one hand, aligns with our empirical knowledge that the code modified when developers implement or improve features of a system, usually has more dependencies to existing code, which eventually has a higher probability to affect the system's architecture [40]. On the other hand, the finding deviates from our understanding that bug fixing would not often alter a system's architecture [40]. But in fact, given a bug fixing commit, it has more than 50% of probability to change the architecture.

As we have described in Section 2, the number of change units in an architectural change commit indicates the number of places

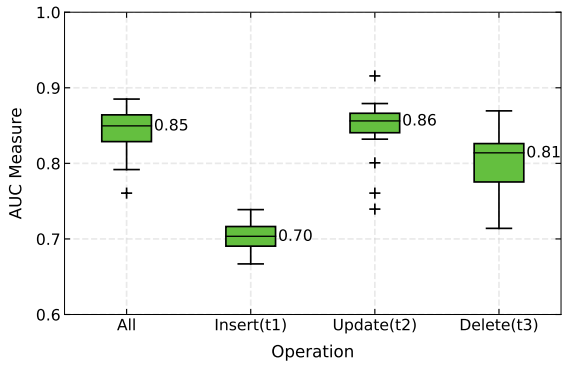


Figure 6: The AUC measure of lines of code in commits.

where architectural changes are introduced in the commit. Hence, by analyzing it, we can explore the characteristics of architectural changes under different intents of developers. We observe that the averaged number of change units in *New Feature* (20), *Task* (18), and *Improvement* (13) are significantly larger than that of *Other* (9), *Bug* (8) and *Test* (8). Moreover, as presented in Figure 5, the number of change units in *New Feature* (20), *Task* (18), and *Improvement* (13), are more sharply distributed in the interval: $[20, +\infty)$, compared with the other three intents. These observations imply that developers tend to introduce much more code that would introduce changes in an architecture when they implement or improve features.

As an answer to RQ2, we find that bug fixing is the most common intent when developers introduce architectural changes. This is partly resulting from the fact that a large number of commits in a system are introduced for bug fixing. In addition, we find that it is when developers try to implement or improve features of a system that they are more likely to introduce architectural changes.

3.4 RQ3: Are modified numbers of lines of code in code commits practical indicators of architectural changes?

3.4.1 Setup. In this research question, we aim to study the correlation between lines of code in commits and architectural changes. We study the correlation with two different methods. On the one hand, we evaluate the Kendall-τ correlation coefficient [19] between lines of code in commits and binary labels for architectural change commits (1 for architectural change commits, otherwise 0). We select Kendall-τ correlation coefficient because we observe that lines of code are not normally distributed.

On the other hand, we propose a simple method to identify architectural change commits based solely on lines of code. It is designed based on the hypothesis that code commits with a large number of code are prone to introduce architectural changes. Intuitively, if this simple method works well, we can conclude that lines of code has a strong correlation with architectural changes, and it can be considered as a symptom for architectural change commits. Specifically, for all commits $CmtSet_i$ in a study subject S_i, we first calculate lines of code *mloc* that are modified in each commit. Then, each commit is assigned a normalized score:

$$score = \frac{mloc}{\max_{j \in |CmtSet_i|} mloc_j}, \tag{7}$$

where *score* apparently ranges over 0 to 1. Given a cut-off value x, we regard those commits that have scores greater than x as architectural change commits. By comparing with the architectural change commits that are identified by our analysis framework, we can measure the performance of this simple method. We select the Area Under the receiver operating characteristic Curve (*AUC*) as our performance measures, as it is independent of a cut-off value and it is not impacted by the skewness of data.

Moreover, to get an in-depth understanding of the correlation, we further break down a commit into several change units, group change units by its operation (**Insert(t_1)**, **Update(t_2)**, **Delete(t_3)**), and calculate lines of code that are modified in each group. Considering the f4b21f5 commit in Figure 2, there are 50 lines of code that are modified, including 32 lines of code in t_1, 9 lines of code in t_2, and 9 lines of code in t_3. Then, we evaluate the *AUC* for each group to understand which group is more closely correlated with architectural changes.

3.4.2 Results. The column 'LOC per Commit' in Table 3 presents the computation results of each study subject. On average, there are 295 lines of code in a commit, including 142 lines of code (48.1%) in t_1(Insert), 80 lines of code (27.1%) in t_2(Update), and 73 lines of code (24.7%) in t_3(Delete). As a comparison, there are only 164 lines of code in an architectural change commits on average, which is much smaller than that of commits. The Kendall-τ correlation coefficient between lines of code and architectural change commits is 0.47, indicating that they are positively but not strongly correlated [15]. The averaged *AUC* measured is 0.84, which also suggests that they are positively but not strongly correlated.

The box plot in Figure 6 shows the *AUC* measure for each group. Through this breakdown of commits, we observe that among the three operations (Insert(t_1), Update(t_2), Delete(t_3)), the Update achieves the highest median *AUC* (0.86), significantly higher than that of Insert (0.70) and Delete (0.81), and even higher than the one using all *mloc* (0.85). This observation suggests that the more lines of code that a developer updates, the more likely that he/she introduce architectural changes.

As an answer to this RQ, through a thorough quantitative analysis, we find that lines of code and architectural changes are positively but not strongly correlated. Among the three operations, Update is the one that most closely correlates with architectural changes.

3.5 RQ4: Are involved numbers of source files in code commits practical indicators of architectural changes?

3.5.1 Setup. In this research question, we study the correlation between number of source files involved in a commit and architectural changes. Similar to the setup in RQ3, we evaluate the Kendall-τ correlation coefficient between the number of source files and binary labels for architectural change commits (The number of source files is not normally distributed).

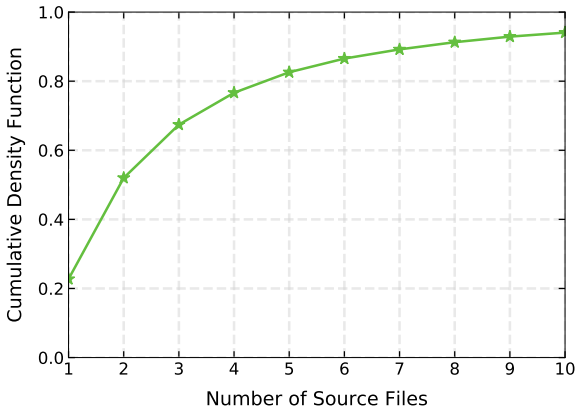


Figure 7: The Cumulative Density Function of the Number of Files of Architectural Change Commits

In addition, we also propose a simple method to identify architectural change commits based solely on the number of source files. Specifically, for all commits $CmtSet_i$ in a study subject S_i, we calculate how many source files are modified in each commit, and we assign a normalized score for each commit based on the number:

$$score = \frac{mfile}{\max_{j \in |CmtSet_i|} mfile_j}, \tag{8}$$

where $mfile$ denotes the number of modified source files. Later on, we evaluate the performance of this method via the *AUC* measure. We present the computation results in the column 'File per Commit' of Table 3. We also visualize the distribution of the number of modified source files in architectural change commits with cumulative density function (CDF) in Figure 7.

3.5.2 Results. The column 'File per Commit' in Table 3 presents the computation results. On average, there are 5.9 source files modified in a commit, while there are 5.4 source files modified in architectural change commits. The Kendall-τ correlation coefficient between number of source files and architectural change commits is 0.47, showing that they only have medium correlation [15]. The low averaged AUC value (0.79) also demonstrates their medium correlation. The CDF in Figure 7 provides some insights for this small correlation: 1) There are 12.3% architectural change commits that only modify one source files; 2) The density curve grows slowly when the number of source files is larger than 5.

As an answer to RQ4, we find that the number of source files in commits and architectural changes are loosely correlated, and there is no strong evidence to suggest that code commits with a large number of source files prone to introduce architectural changes.

4 DISCUSSION

4.1 Correctness Analysis

4.1.1 Setup. We have manually analyzed 400 sampled commits, including 200 samples with detected architectural changes (Positive samples) and 200 samples without undetected architectural changes (Negative samples), to verify the correctness of our implementation.

Table 5: The manual analysis results

Label	Yes	Not Sure	No
Positive Samples (200)	132	25	43
Negative Samples (200)	146	18	36

We invite 3 students to manually review these commits. Each commit undergoes rigorously assessment and is assigned with three labels: '*Yes*', '*No*', and '*Not sure*'. For a commit, Label: '*Yes*' represents that our detection result is correct, and Label: '*No*' represents that our detection result has mistakes. Label: '*Not Sure*' represents that the correctness of our detection result is uncertain. We present the manual analysis results in Table 5.

4.1.2 Results. The column '*Yes*', '*No*' and '*Not Sure*' in in Table 5 present the number of commits under each label. For 200 positive samples, they have 132 samples labelled with '*Yes*', 25 samples labelled with '*Not Sure*', 43 samples labelled with '*No*'. For 200 negative samples, they have 146 samples labelled with '*Yes*', 18 samples labelled with '*Not Sure*', 36 samples labelled with '*No*'. Under these samples, we further calculate that the precision of our tool is 71%. The recall of our tool is 79% and the F1 measure is 75%. This suggests the results of our detection tool is reliable for having relatively high accuracy in detecting architectural changes.

4.2 Implications

We discuss the border implications of our findings for software contributors and research community.

4.2.1 Software Contributors. It is imperative for software architects to monitor architectural changes. We have found in RQ1 that there are 53.4% of commits introducing architectural changes on average and developers contribute 3.0 architectural change commits per day. These findings suggest that a system's architecture is intensively changed on a day-to-day basis. If architects of the system cannot effectively manage changes in the architecture in time, the architecture is likely to degrade under the fast-delivery architectural changes, leading to increased maintenance effort.

In addition, as we have shown RQ2, bug fixing is the most common intent under which developers introduce architectural changes, and when developers intent to implement or improve a feature, they are highly likely to introduce architectural changes. Hence, for the authors of code changes, it is important to notice the architectural changes introduced in their commits, especially when intenting to fix a bug and improve a feature. A recent study also suggests that being aware of architectural changes can lead to high-quality code changes [40].

To facilitate monitoring architectural changes and help developers notice architectural changes, we would like to provide plugins for prevalent continuous integration systems and code review systems based on our architectural change analysis framework.

4.2.2 Research Community. Considering the large number of architectural change commits in a system, it would be impractical to warn architects every other commit that changes the architecture and to require architects to evaluate its impact. To this end, two natural question raises: 1) When should architects evaluate the

compounding effect of continuous architectural changes? 2) Are existing architectural change impact measures ready to be used in practice in terms of their effectiveness and efficiency, to alleviate the burden of architects? The answers to these questions are very valuable. We leave the study of them as our future work.

4.3 Threat to Validity

There are internal, external and constructive threats to validity associated to the results we present. The primary internal threat arises from the definition of structural changes used in our framework. Our architectural change analysis framework only considers code changes may modify structural dependencies among code components. However, recent studies have verified that structural dependencies is one of the best proxies for developers' perception of cohesion and coupling [9, 12]. Hence, it would be safe to use the framework in the empirical study. In terms of external threats, our study only focuses on 16 subjects that are primarily written in Java. It remains unclear whether our findings can generalize to other open-source projects and closed-source industrial projects. To mitigate this threat, our study subjects are selected from top starred Apache open projects. They have been widely studied in software architecture research [28, 34, 35, 50, 51]. In addition, we also study a large number of commits for each subject to further mitigate the threat. To this end, the findings should generalize to other projects. When it comes to constructive threats, our framework is built upon several open source tools. During implementations, we have submitted pull requests to improve the tools.

5 RELATED WORK

5.1 Architecture Change Study

Several studies have been conducted on architectural changes. Based on the methods to detect architectural changes, these studies can be classified into two categories: recovery-based [10, 28] and metric-based [40, 45]. A representative recovery-based approach is the work of Duc et al. [28]. They measure architectural changes by comparing the recovered architectures of a system in two versions using state-of-the-art recovery techniques [20, 48]. Their methods are further applied to detect architectural decay in software systems [29, 45]. The method proposed by Paixao et al. [40] is one of the representative metric-based approaches. They measure architecture changes using structural cohesion and coupling metrics. Their methods are applied to measure the architectural impact of refactoring [39]. Compared with these studies, our work detect architectural changes of commits in a fine grain. With the assist of our tool, we can locate concrete code changes contributing to architectural changes.

5.2 Code Commit Analysis

Over the past decades, numerous approaches are proposed to help developers analyze code changes in commits. These techniques can be classified into three categories including commit diff detection, commit message generation and commit change pattern detection. Commit diff detection techniques aim at computing code changes between two versions of source files. The most influential techniques of this category are tree-based approaches which can capture syntactic code changes [17, 18, 22] compared with text-based approaches [8, 13, 14, 16, 43]. For example, ChangeDistiller [18] use a general tree difference identification algorithm to generate edit scripts based on coarse-grained ASTs of two source files. GumTree [17] further generates edit scripts which can well reflect developer intent using several heuristics. Commit message generation techniques [11, 24, 30, 33, 36, 42] aim at generating natural language description of code changes in commits. For example, ChangeScribe [30, 33] generates commit message using heuristics. Jiang et al. [24] and Moreno et al. [36] further leverage neural networks to improve the performance of this technique. Commit change pattern detection techniques aim at discovering frequent patterns in code commits during software evolution [13, 23, 25–27, 37, 38, 46, 52, 55]. A more recent work is CLDiff [23], a general tool to discover code change in a concise format. Compared with these state-of-the-art techniques, our work aims at studying the architectural impact of code changes.

6 CONCLUSION

In this work, we propose an automated framework to identify changes in architectures from code commits. The framework does not rely on architecture recovery techniques, and it only takes a code commit as input. Based on the framework, we further conduct a large-scale empirical study on 368,847 commits of 16 Apache open projects to characterize architectural changes. Our empirical findings, on the one hand, advance our understanding about architectural changes. On the other hand, the findings provide practical implications for software contributors and shed light on potential research directions on architecture maintenance.

ACKNOWLEDGMENTS

This work was supported by National Key R&D Program of China (2018YFB1004500), National Natural Science Foundation of China (61632015, 61772408, U1766215, 61721002, 61532015, 61833015, 61902306, 62072351), China Postdoctoral Science Foundation (2019TQ0251, 2020M673439), Youth Talent Support Plan of Xi'an Association for Science and Technology (095920201303), Ministry of Education Innovation Research Team (IRT 17R86), and Project of China Knowledge Centre for Engineering Science and Technology.

REFERENCES

[1] 2019. *Cassandra.* https://cassandra.apache.org
[2] 2019. *eclipse.* https://help.eclipse.org
[3] 2019. *Git.* https://git-scm.com
[4] 2019. *Git Log.* https://git-scm.com/docs/git-log
[5] 2019. *GumTree.* https://github.com/GumTreeDiff/gumtree
[6] 2019. *JGit.* https://www.eclipse.org/jgit
[7] 2019. *Jira.* https://issues.apache.org/jira
[8] Muhammad Asaduzzaman, Chanchal K. Roy, Kevin A. Schneider, and Massimiliano Di Penta. 2013. LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines. (2013).
[9] Gabriele Bavota, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. 2013. An empirical study on the developers' perception of software coupling. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 692–701.
[10] Pooyan Behnamghader, Duc Minh Le, Joshua Garcia, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. 2017. A large-scale study of architectural evolution in open-source software systems. *Empirical Software Engineering* 22, 3 (2017), 1146–1193.
[11] Raymond P. L. Buse and Westley R. Weimer. 2010. Automatically documenting program changes. In *IEEE/ACM International Conference on Ase.*

[12] Ivan Candela, Gabriele Bavota, Barbara Russo, and Rocco Oliveto. 2016. Using Cohesion and Coupling for Software Remodularization: Is It Enough? *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 3 (2016), 24.

[13] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. 2008. Tracking Your Changes: A Language-Independent Approach. *IEEE Software* 26, 1 (2008), 50–57.

[14] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. 2009. Ldiff: An enhanced line differencing tool. In *IEEE International Conference on Software Engineering*.

[15] Patricia Cohen, Stephen G West, and Leona S Aiken. 2014. *Applied multiple regression/correlation analysis for the behavioral sciences*. Psychology Press.

[16] Di Cui, Ting Liu, Yuanfang Cai, Qinghua Zheng, Qiong Feng, Wuxia Jin, Jiaqi Guo, and Yu Qu. 2019. Investigating the impact of multiple dependency structures on software defects. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 584–595.

[17] Jeanr E My Falleri, Flor E Al Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. *automated software engineering* (2014), 313–324.

[18] Beat Fluri, M. Wursch, Martin Pinzger, and Harald C. Gall. 2007. Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering* 33, 11 (2007), 725–743.

[19] Ron N. Forthofer and Robert G. Lehnen. 1981. *Rank Correlation Methods*. Springer US, Boston, MA. 146–163 pages. https://doi.org/10.1007/978-1-4684-6683-6_9

[20] Joshua Garcia, Daniel Popescu, Chris Mattmann, Nenad Medvidovic, and Yuanfang Cai. 2011. Enhancing architectural recovery using concerns. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 552–555.

[21] Emanuel Giger, Martin Pinzger, and Harald C Gall. 2012. Can we predict types of code changes? an empirical analysis. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 217–226.

[22] Masatomo Hashimoto and Akira Mori. 2008. Diff/TS: A Tool for Fine-Grained Structural Change Analysis. In *Conference on Reverse Engineering*.

[23] Kaifeng Huang, Bihuan Chen, Xin Peng, Daihong Zhou, Ying Wang, Yang Liu, and Wenyun Zhao. 2018. ClDiff: Generating Concise Linked Code Differences. In *ACM/IEEE International Conference on Automated Software Engineering*. ACM, 679–690.

[24] Siyuan Jiang, Ameer Armaly, and Collin Mcmillan. 2017. Automatically Generating Commit Messages from Diffs using Neural Machine Translation. (2017).

[25] Miryung Kim and David Notkin. 2009. Discovering and representing systematic code changes. In *IEEE International Conference on Software Engineering*.

[26] Miryung Kim, David Notkin, and Grossman Dan. 2007. Automatic Inference of Structural Changes for Matching across Program Versions. In *International Conference on Software Engineering*.

[27] Miryung Kim, David Notkin, Grossman Dan, and Jr Wilson, Gary. 2013. Identifying and Summarizing Systematic Code Changes via Rule Inference. *IEEE Transactions on Software Engineering* 39, 1 (2013), 45–62.

[28] Duc Minh Le, Pooyan Behnamghader, Joshua Garcia, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. 2015. An empirical study of architectural change in open-source software systems. In *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 235–245.

[29] Duc Minh Le, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. 2018. An empirical study of architectural decay in open-source software. In *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 176–17609.

[30] Mario Linares-Vasquez, Luis Fernando Cortes-Coy, Jairo Aponte, and Denys Poshyvanyk. 2015. ChangeScribe: A Tool for Automatically Generating Commit Messages. In *IEEE/ACM IEEE International Conference on Software Engineering*.

[31] Huihui Liu, Yijun Yu, Bixin Li, Yibiao Yang, and Ru Jia. 2018. Are Smell-Based Metrics Actually Useful in Effort-Aware Structural Change-Proneness Prediction? An Empirical Study. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 315–324.

[32] Hongmin Lu, Yuming Zhou, Baowen Xu, Hareton Leung, and Lin Chen. 2012. The ability of object-oriented metrics to predict change-proneness: a meta-analysis. *Empirical software engineering* 17, 3 (2012), 200–242.

[33] Paul W Mcburney and Collin Mcmillan. 2016. Automatic Source Code Summarization of Context for Java Methods. *IEEE Transactions on Software Engineering* 42, 2 (2016), 103–119.

[34] Ran Mo, Yuanfang Cai, Rick Kazman, and Lu Xiao. 2015. Hotspot patterns: The formal definition and automatic detection of architecture smells. In *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*. IEEE, 51–60.

[35] Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, and Qiong Feng. 2016. Decoupling level: a new metric for architectural maintenance complexity. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 499–510.

[36] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. 2017. ARENA: An Approach for the Automated Generation of Release Notes. *IEEE Transactions on Software Engineering* 43, 2 (2017), 106–127.

[37] Meng Na, Miryung Kim, and Kathryn S. Mckinley. 2011. Systematic editing: generating program transformations from an example.

[38] Meng Na, Miryung Kim, and K. S. Mckinley. 2013. Lase: Locating and applying systematic edits by learning from examples. In *International Conference on Software Engineering*.

[39] Mel Ó Cinnéide, Laurence Tratt, Mark Harman, Steve Counsell, and Iman Hemati Moghadam. 2012. Experimental assessment of software metrics using automated refactoring. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*. 49–58.

[40] Matheus Paixao, Jens Krinke, Dong Gyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman. 2017. Are developers aware of the architectural impact of their changes?. In *IEEE/ACM International Conference on Automated Software Engineering*.

[41] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46 (2015), 1155–1179. https://doi.org/10.1002/spe.2346

[42] S. Rastkar and G. C. Murphy. 2013. Why did this code change?. In *International Conference on Software Engineering*.

[43] Steven P. Reiss. 2008. Tracking source locations. (2008).

[44] Daniele Romano and Martin Pinzger. 2011. Using source code metrics to predict change-prone java interfaces. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 303–312.

[45] Arman Shahbazian, Youn Kyu Lee, Duc Le, Yuriy Brun, and Nenad Medvidovic. 2018. Recovering architectural design decisions. In *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 95–9509.

[46] Suresh Thummalapenta, Luigi Cerulo, Lerina Aversano, and Massimiliano Di Penta. 2010. An empirical study on the maintenance of source code clones. *Empirical Software Engineering* 15, 1 (2010), 1–34.

[47] Irene Tollin, Francesca Arcelli Fontana, Marco Zanoni, and Riccardo Roveda. 2017. Change prediction through coding rules violations. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. 61–64.

[48] Vassilios Tzerpos and Richard C Holt. 2000. ACDC: An Algorithm for Comprehension-Driven Clustering.. In *wcre*. 258–267.

[49] Ye Wang, Na Meng, and Hao Zhang. 2018. An Empirical Study of Multi-entity Changes in Real Bug Fixes. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 287–298. https://doi.org/10.1109/ICSME.2018.00038

[50] Lu Xiao, Yuanfang Cai, and Rick Kazman. 2014. Design rule spaces: A new form of architecture insight. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 967–977.

[51] Lu Xiao, Yuanfang Cai, Rick Kazman, Ran Mo, and Qiong Feng. 2016. Identifying and quantifying architectural debt. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 488–498.

[52] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. 2004. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering* 30, 9 (2004), 574–586.

[53] Farida El Zanaty, Toshiki Hirao, Shane McIntosh, Akinori Ihara, and Kenichi Matsumoto. 2018. An Empirical Study of Design Discussions in Code Review. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, New York, NY, USA, Article 11, 10 pages. https://doi.org/10.1145/3239235.3239525

[54] Hao Zhong and Zhendong Su. 2015. An Empirical Study on Real Bug Fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, 913–923.

[55] Thomas Zimmermann, Andreas Zeller, P Weissgerber, and Stephan Diehl. 2005. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering* 31, 6 (2005), 429–445.