Towards Better Understanding of Software Quality Evolution Through Commit-Impact Analysis

Pooyan Behnamghader, Reem Alfayez, Kamonphop Srisopha, and Barry Boehm
Department of Computer Science, University of Southern California
Los Angeles, CA, USA
{*pbehnamg, alfayez, srisopha, boehm*}*@usc.edu*

***Abstract*—Developers intend to improve the quality of the software as it evolves. However, as software becomes larger and more complex, those intended actions may lead to unintended consequences. Analyzing change in software quality among different releases overlooks fine-grained changes that each commit introduces. We believe that studying software quality before and after each commit (commit-impact analysis) can reveal a wealth of information about how the software evolves and how each change impacts its quality. In this paper, we explore whether each commit has an impact on the source code, investigate the compilability of each impactful commit, examine how source code changes affect software quality metrics, and study the effectiveness of using a certain metric as software quality indicator. We analyze a total of 19,580 commits from 38 Apache Java software systems to better understand how change occurs, why, and by who.**

1. Introduction

Version Control Systems, such as Git, Concurrent Versioning System (CVS), and Subversion (SVN), facilitate collaborative software development process. They enable developers to share code easily, and they keep track of each change performed on software systems during development and maintenance. When developers make a change on their working copy of the software, they write it back to the repository using commit. Commits show what changed, who changed it, and when. They contain a wealth of information about software evolution.

Prior research have been focusing mainly on analyzing different versions of software to understand how it evolves [1] [2] [3] [4]. While this approach gives an insight on change over each version, it only shows the major milestones of the software system evolution. Important details between releases may be overlooked. For example, a developer may unknowingly commit broken code to the repository. Since software developers do not ship uncompiled code, this detail may not be revealed in that coarse-grained analysis. We believe that analyzing the impact of each commit (commit-impact analysis) is the best approach to study software evolution because it holds details of every stage in the software evolution, such as the time of each change and who made the change.

Open Source Software (OSS) Development has its unique factors that closed source systems do not have. OSS systems are developed by a large number of volunteers who can choose what to work on. They are geographically distributed and collaborate remotely without face-to-face communication [5]. The work is coordinated through versioning systems, issues tracking systems, such as Jira, and the project mailing list. While OSS development process has its unique way of developing and managing software, it has proven to be effective in producing high-quality software that can compete with the commercial systems. Currently, 50.3% of all the websites whose webserver is known are using Apache server[1] and 5.7% of all operating system usages are Linux[2]. This is credited to the developers' freedom in picking what to work on which allows them to write parts of code that they are expert in and passionate about. This suggests that OSS systems change frequently by developers who have different level of skills and experience. In addition, having the code publicly available makes finding and fixing bugs faster as Linus's Law states [5].

Many software development teams put more emphasis on delivering software on time and within budget than fixing code problems before releasing. This mindset results in difficult to understand and maintain code which can ultimately cause failures and disasters [6] [7]. This can be avoided by focusing more on the code quality as well as on its functionality [7]. In the OSS communities, some systems advise their developers to use static analysis tools to control the code quality.

Static code analysis is the type of software analysis that is performed without running the program. The analysis can be conducted on the source code and/or on the compiled code. This type of analysis can reveal software bugs and issues in the development phase [8]. In an effort to improve software quality, the software development community continuously produces and improves static analysis tools to provide numerous software metrics that can be used to measure software quality. Static analysis tools either measure low level to high level quality metrics, such as code smells [9] and architectural decays [10], or provide the means to run such measurements (e.g., control flow graphs [11] and

1. https://w3techs.com/technologies/details/ws-apache/all/all
2. https://www.w3schools.com/browsers/browsers_os.asp

architectural facts [12]).

In this paper, we leveraged static code analysis tools and version control systems to have a better understanding of how OSS systems evolve. We performed a large-scale analysis on 19,580 commits from 38 Apache Java systems from multiple software domains. We studied each commit to understand how often a commit impacts the module containing most of the code. We investigated the compilability of the revisions created by impactful commits. We furthered our investigation by examining each system to understand why some revisions are not compilable. We summarized our observations to help developers realize what causes and how to avoid these compilation errors.

We then applied multiple static analysis tools to revisions created by impactful commits and calculated the difference between the metric values at each revision and its immediate previous revision. This allows us to see if there is a change in certain quality attributes overtime. Finally, we explored whether using static code analysis tools during the software development improves its quality.

The remainder of the paper is organized as follows. In Section 2, we summarize the foundations that have been brought together to enable the work described in this paper. In Section 3, we explain commit-impact analysis. Section 4 describes the setup for our empirical study. Section 5 presents the key results. Section 6 highlights some related work. Section 7 discusses the threats to validity. Finally, we conclude and describe future work in Section 8.

2. Foundation

Our work discussed in this paper was directly enabled by three research threads: (1) static code analysis, (2) software quality metrics, and (3) version control systems. In this section, we will summarize this foundational work.

2.1. Static Code Analysis

Static code analysis is the process of analyzing a software without executing it. The analysis reveals software defects that are not visible to the compiler which can be useful in detecting developers' mistakes at early stages. It can be performed on source code which is exact reflection of the software itself or on the compiled bytecode [8].

SonarQube [3] is an open source quality management platform that has many source code scanners. While these different analyzers provide similar functionalities, each of them can be integrated into different development environments. In addition, SonarQube offers SonarQube Scanner which is a Java-based command-line tool. SonarQube provides various software quality metrics such as complexity, NCLOC, number of classes, etc. The major software metrics that it provides are bugs, vulnerabilities, and code smells. Sonarqube defines these software metrics as follows:

- Bugs: code that is demonstrably wrong or highly likely to yield unexpected behavior.
- Vulnerabilities: code that is potentially vulnerable to exploitation by hackers.
- Code Smell: code that will be confusing to maintainers or give them pause.

FindBugs [4] is an open source static analysis tool developed at the University of Maryland and distributed under the terms of the Lesser GNU Public License. The tool runs on the binary files. It uses bug pattern detectors to detect actual bugs in Java software systems [13]. The tool categorizes bugs into eight categories: bad-practice, correctness, internationalization, malicious code vulnerability, multi-threaded correctness, performance, security, experimental, and style.

PMD [5] is an open source static analysis tool that runs on the source code. It is distributed under BSD License. The tool supports Java, JavaScript, XML, and XSL programming languages and can detect common development flaws such as empty catch blocks, unnecessary if statement, unused variables, duplicated code, etc. The tool has many rule sets for each programing language that it supports. It has more than twenty rulesets for java. These rulesets can overlap, meaning that a rule might belongs to multiple rulesets.

2.2. Software Quality Metrics

We consider three groups of metrics: Basic, Code Quality, and Security. Basic metrics are simple to calculate in comparison to code quality and security metrics. Table 1 lists the quality metrics we selected in this study.

TABLE 1. QUALITY METRICS

Group	Abbr.	Tool	Description
Basic	LC	SonarQube	Physical Lines excl. Whitespaces/Comments
	FN	SonarQube	Functions
	CS	FindBugs	Classes
Code Quality	CX	SonarQube	Complexity (Number of Paths)
	SM	SonarQube	Code Smells
	PD	PMD	Empty Code, Naming, Braces, Import Statements, Coupling, Unused Code, Unnecessary, Design, Optimization, String and StringBuffer, Code Size
Security	VL	SonarQube	Vulnerabilities
	SG	PMD	Security Guidelines
	FG	FindBugs	Malicious Code, Security

Basic metrics measure the size of a software system. For the basic group, we selected lines of code, functions, and classes. Lines of code (LC) is the number of physical lines excluding whitespaces, tabulations, and comments. Functions (FN) is the number of methods. Classes (CS) is the number of classes including annotations, enums, interfaces, and nested classes.

Complexity, Code Smells, and a subset of PMD violations are our code quality metrics. Complexity (CX) is the number of paths through the code. Whenever the control flow splits, complexity increases by one. Code Smells (SM) are pieces of code that make the system hard to maintain. PMD defines a rich set of violations that covers a variety of

3. https://www.sonarqube.org

4. http://findbugs.sourceforge.net

5. https://pmd.github.io/

quality attributes. PD is the number of issues in a subset of those violations related to code quality.

Security metrics are designed to identify potential security holes. We calculate the number of Vulnerability (VL) issues from SonarQube as our first, and Security Code Guidelines (SG) violations from PMD as our second security metric. FindBugs has two classes of issues for security: Malicious Code and Security. We consider the sum of the number of issues in both of them (FB) as our third security metric.

2.3. Version Control Systems

Version Control Systems (VCS) are designed to track change in an evolving software system. In this paper, we study systems that use Git as their VCS. Git is a distributed VCS designed for collaborative software development environments. Every software system that uses Git has a Git "repository" which contains the whole development history. Each developer has a copy of the repository. Developers can concurrently work on the software system, change it, and eventually persist their changes in the repository using Git "commit"s. A commit contains a variety of information, such as the time of commit, the developer's basic information, the changed files, and a commit message.

3. Commit-Impact Analysis

This section describes the meaning of change and its impact in the context of this paper. We define what an impactful commit is. We also discuss how to calculate the impact of a commit on the quality of the main module of a software system.

Every commit produces a new revision of the software system. In that new revision, the source code may or may not be different from the previous revision. If the source code remains untouched, it suggests that the changes have affected other artifacts, such as documentation and build configuration files.

The source code changes introduced by a commit may affect multiple modules (e.g., "core", "test", and "plugins"). In this study, we are interested in analyzing the module that contains the majority of the source code. We call that module the **"main module"**. For example, the main module in most Apache library systems is the "core" module.

Figure 1 shows a simple case study to illustrate different concepts discussed in the rest of this section. S is an evolving software system. R is its repository that contains 20 commits.

An **"impactful commit"** is a commit that changes the main module of a software system. Impactful commits may also change other modules, such as plugins and tests; however, in Commit-Impact Analysis, we only consider their impact on the main module.

A parent of a commit is its immediate ancestor in the development history. For example, c_7 is the parent of c_8 in our simple case study. There are three types of commits based on the number of parents [14]:

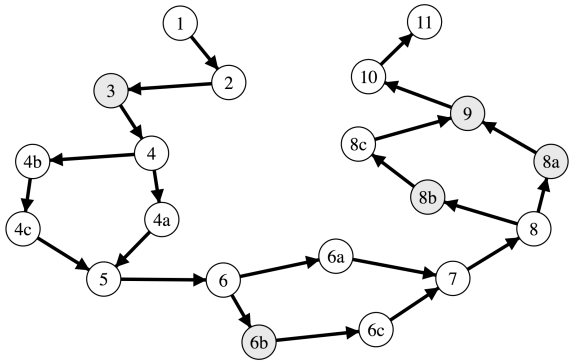

Figure 1. The evolution of system S in the repository R. Impactful commits are denoted in gray.

1) **"Normal"** is a commit with only one parent.
2) **"Merge"** is a commit with two (or more) parents.
3) **"Initial"** (or orphan) is a commit with no parent.

In repository R, c_2 is a normal commit, c_5 is a merge, and c_1 is the initial commit.

A normal commit is impactful if it changes the main module. In repository R, normal impactful commits are c_3, c_{6b}, c_{8a}, and c_{8b}. To find the impact of a normal impactful commit on a certain quality attribute, we measure that quality attribute in two revisions: the revision created by the commit, and the revision created by its parent. Then, we calculate the difference between the measured values to find whether and how the quality attribute has changed in that specific point.

Merge commits are the result of integrating different branches in the repository into a single branch. Developers create branches for various purposes such as adding a new feature or fixing a bug. Each branch separates the main line of the development. When the development on a branch is finished, the developers integrate that branch into another branch (e.g., into the main line of development) using a merge commit.

Consequently, a merge commit does not contain any change itself. It only transfers the changes that are previously committed in the merging branches into a destination branch. There are three cases to consider when a merge happens. The main module after a merge may be identical to:

1) the main modules of all parents.
2) the main module of one of parents.
3) the main module of none of parents.

The first case indicates that no change has happened to the main module in any of the merging branches. We do not consider this merge commit as an impactful commit. For example, there are two branches starting from c_4 and merging at c_5. There is no impactful commit on any of these branches. As a result, the main module resulted from c_5 is identical to the main module resulted from c_4.

The second case indicates that the main module has changed by at least one impactful commit in the branch

leading to its identical parent and remained untouched in other branches. The merge commit only transfers those changes to the main line of the development and does not have any impact. In this case, we do not consider this merge commit as an impactful commit. For instance, in the evolution of S from c_6 to c_7, the main module only changes at c_{6b}, and c_{6c} is identical to c_{6b}. As a result, the main module resulted from c_7 is identical to c_{6c} and c_{6b}, but it is different from c_{6a} and c_6. Particularly, the difference between c_7 and c_{6a} is the same as the difference between c_{6b} and c_6.

The third case indicates that the main module has changed by impactful commits over at least two branches leading to the merge. Consequently, the merge has produced a new revision of the main module which contains the developments in all branches and is different from all the previous revisions. In this case, we consider this merge as an impactful commit; however, we cannot calculate its impact. If we compare the main module after an impactful merge with any of its parents, the difference will be the summation of all developments over all other branches leading to the merge. If we compare it with the main module after the common ancestor commit of all branches leading to the merge, the difference will be the summation of all changes over all branches. For example, c_9 is an impactful merge which contains the changes introduced in c_{8a} and c_{8b}. Since we are interested in studying the changes introduced by one developer in a single commit, we do not calculate the impact in this case.

The initial (or orphan) commit is impactful if it contains the first revision of the main module. In this case, we consider the quality attributes of the first revision of the main module as the impact of the initial commit. If the initial commit does not introduce the main module, we do not consider it impactful; however, in this case, there will be a normal commit that introduces the main module for the first time. We calculate the impact of that normal commit the same way as an initial commit.

4. Empirical Study Setup

Our study targets four research questions regarding software quality evolution. The scalability challenge involved in mining software repositories on a large-scale has resulted in the lack of empirical studies on the impact of each commit on multiple software quality metrics. As a result, the extent to which commits are impactful, to which the revisions created by impactful commits are compilable, to which software quality changes, and to which different software metrics sufficiently indicate change are generally unclear. The next two subsections describe our research questions and our data collection process.

4.1. Research Questions

RQ1: To what extent do developers commit impactful changes? This research question focuses on the impact of individual commits on the main module of a software system. During evolution, developers modify different parts of a software system. For example, a commit may only contain some changes in the documentations and unit tests, which do not change the main module of the system. This question will shed light on how and to what extent this change happens.

RQ2: To what extent and how do impactful commits break the compilability of the project? One of the most basic code quality attributes is compilability. A revision of a software system is expected to be compilable. However, developers knowingly and unknowingly commit uncompilable code. This question investigates the extent to which the revisions after impactful commits are compilable and studies the reasons for compilation errors.

RQ3: To what extent do impactful commits affect software quality attributes? This research question focuses on the impact of impactful commits on different quality attributes. Not all commits that affect the main module change its quality attributes. For example, if a commit introduces only a minor refactoring (e.g., renaming a variable), it may not change most of the quality metrics such as number of classes and number of security issues. This question studies how impactful commits affect different quality attributes.

RQ4: Should developers rely on a single software quality metric as a change indicator? Each software quality metric provides different perspectives about software quality. One metric may measure changes in the software better than others. This research question investigates the sufficiency of each software metric as a change indicator.

4.2. Data Collection

In order to answer these research questions, we analyzed a total of 19,580 commits from 38 Apache software systems.

We utilized GitHub API[6] to fetch the name, the number of commits, the repository URL, the programming language, and the latest update date of all Apache software systems. We excluded non-Java systems and the ones that did not have any update in 2017 at the time of this study. Then we selected all the systems with less than 3,000 commits.

We selected the systems that have one "main" module containing most of the Java files. In many cases, this main module is just the "src" folder under the root directory of the system. However, in multiple cases there is no such a directory. Instead, there are multiple modules which only one of them is the main one. For example, in many libraries, the main module is the "core" module of that library. If we could not find a single big module or the core module, we excluded that system aside as it was not suitable for our study.

Next, we checked each system's documentation for its possible compile commands. If a system required manually installing specific tools (e.g., Tajo requires Protocol Buffer) for compilation, we skipped that system. We modified the default compile commands in order to disable all tests, validations, and enforcements. We also compiled only the

6. https://developer.github.com/v3/

main module and its dependencies to prevent compilation error in other modules affecting our results.

TABLE 2. SUBJECT SYSTEMS

System	Domain	Rev.	Time span	MSLOC
Avro	Data Serialization	188	07/11-01/17	2.43
Calcite	Data Management	650	07/14-02/17	75.46
C-BCEL	Bytecode Eng.	576	06/06-12/16	16.5
C-Beanutils	Reflection Wrapper	139	07/07-11/16	1.71
C-Codec	Encoder/Decoder	368	09/11-09/16	2.32
C-Collections	Collections Ext.	565	03/12-10/16	13.87
C-Compress	Compress Lib.	1229	07/08-02/17	16.84
C-Configuration	Conf. Interface	333	04/14-01/17	8.83
C-CSV	CSV Library	598	11/11-02/17	0.68
C-Dbcp	DB Conn. Pooling	188	12/13-11/16	2.06
C-IO	IO Functionality	914	01/02-12/16	5.99
C-JCS	Java Caching	136	04/14-02/17	3.66
C-Jexl	Expression Lang.	295	08/09-10/16	2.26
C-Net	Clientside Protocol	877	08/06-02/17	14.96
C-Pool	Object Pooling	278	04/12-02/17	1.27
C-SCXML	State Chart XML	335	03/06-08/16	2.53
C-Validator	Data Verification	279	07/07-02/17	1.63
C-Vfs	Virtual File System	611	11/06-01/17	13.32
CXF-Fediz	Web Security	171	04/12-03/17	0.89
Drill	SQL Query Engine	616	04/15-02/17	58
Flume	Data Collection	342	08/11-11/16	2.14
Giraph	Graph Processing	387	11/12-01/17	15.12
Hama	BSP Computing	717	06/08-04/16	7.77
Helix	Cluster MNGMT	762	01/12-06/16	27.36
HTTPClient	Client-side HTTP	925	03/09-01/17	16.54
HTTPCore	HTTP Transport	565	03/09-02/17	7.38
Mina	Socket Library	263	11/09-04/15	2.1
Mina-SSHD	SSH Protocols	818	04/09-02/17	24.66
Nutch	Web Crawler	893	03/05-01/17	17.85
OpenNLP	NLP Toolkit	424	04/13-02/17	18.85
Parquet-MR	Storage Format	345	02/13-01/17	5.2
Phoenix	OLTP Analytics	1260	01/14-02/17	131.13
Qpid-JMS	Message Service	431	02/15-02/17	9.12
Ranger	Data Security	409	03/15-02/17	19.97
Santuario	XML Security	480	01/11-02/17	16.38
Shiro	Java Security	285	03/09-11/16	3.26
Tiles	Web Templating	318	07/06-07/16	1.71
Zookeeper	Distributed Comp.	610	06/08-02/17	14.98
Total		**19580**	**01/02-03/17**	**586.73**

We identified all impactful commits in the selected subject systems by analyzing their Git repository. If a commit changes at least one Java file in the main module, we consider it impactful. We executed the collected compile commands on revisions created by impactful commits using multiple versions of JDK. We also considered different project management and comprehension tools, such as Ant[7] and Maven[8], to compile some older revisions. We also modified the compile command to target the main module in older revisions, if the system was subject to restructuring.

We only included a system in our data set, if we could compile at least 100 revisions created by impactful commits of it. Table 2 summarizes our subject systems, including their domains, number of compiled revisions, timespan between the earliest and latest analyzed revisions, and cumulative size of all compiled revisions.

After finalizing our subject systems, we executed PMD, FindBugs, and SonarQube on the main module of all compiled revisions. PMD and FindBugs are straightforward to execute as they only need the path to the source code (PMD) or to the binary files (FindBugs). Both tools generate an XML report. On the other hand, SonarQube has its own analysis server. It saves the results in an internal database. It also requires generating a configuration file for each revision.

To run SonarQube, we deployed its analysis server, generated configuration files for all compiled revisions, executed SonarQube on each revision, and finally fetched the results using SonarQube API[9]. We also had to delete the result of each revision from SonarQube server after fetching its data since the responsiveness of the server degraded after storing the results of several hundred revisions. Finally, we parsed the results generated by the three analysis tools and stored the quality attributes into our own relational database.

To collect large-scale data from multiple thousands of software revisions, we developed a distributed approach inspired by our previous work [15] [16] whose explanation is out of the scope of this paper.

5. Results

In this section, we present the results obtained from each research question. For each question, we provide a table summarizing our findings. Then we discuss the interesting observations based on these findings.

5.1. RQ1

To answer RQ1, we calculated the total number of commits, impactful commits, developers, and impactful developers for each system within its timespan. We also calculated the ratio of the impactful commits to the total commits and the ratio of the impactful developers to the total developers. Table 3 summarizes the results of our analysis.

We observed that on average 48% of commits are impactful and 69% of developers have at least one impactful commit. This illustrates the centrality of the main module in the development process of our subject systems. Considering our system selection criteria, this observation is expected; however, there are some subject systems with exceptionally low ratios. We looked into these subject systems to further investigate this observation.

Flume and Tiles are frameworks designed to be integrated with a variety of other systems. Hence, the main module in both systems is accompanied with multiple smaller components (e.g., sdk, agents, and plugins). Additionally, there are proportionally large test suites developed for the main module in both systems. These test suites are not analyzed in this study. As a result, the ratio of impactful commits for Flume (29%) and Tiles (24%) are relatively lower than the average. However, in both systems, the high ratio of impactful developers (58% and 70%) suggests that the main module is the focus of the development.

Ranger is a comprehensive data security framework designed to be integrated with different tools across Hadoop

7. http://ant.apache.org
8. http://maven.apache.org
9. https://docs.sonarqube.org/display/DEV/Web+API

platform. Unlike Flume and Tiles, it has a relatively small test suite. However, its repository contains an exceptionally large number of small modules, agents, and plugins. A lot of commits affecting those modules are skipped in our analysis. As a result, the ratio of impactful commits for Ranger (29%) is relatively lower than other subject systems. The ratio of impactful developers (53%) is not within the standard deviation of the mean as well. However, it still shows that more than half of the developers are engaged in the development of the main module.

TABLE 3. THE NUMBER OF ALL COMMITS, IMAPCTFUL COMMITS (AND RATIO), IMPACTFUL DEVELOPERS (AND RATIO), AND COMPILED IMPACTFUL COMMITS (AND RATIO)

	RQ1						RQ2	
System	Commit			Developer			Commit	
	All	Impactful	%	All	Impactful	%	Compiled	%
Avro	757	188	25	39	19	49	188	100
Calcite	1201	673	56	121	95	79	650	97
C-BCEL	900	589	65	13	8	62	576	98
C-Beanu.	392	139	35	9	7	78	139	100
C-Codec	706	368	52	6	5	83	368	100
C-Collect.	890	570	64	16	10	63	565	99
C-Comp.	2037	1240	61	30	22	73	1229	99
C-Config.	637	340	53	4	4	100	333	98
C-CSV	1032	606	59	11	7	64	598	99
C-DBCP	393	188	48	8	5	63	188	100
C-IO	1941	943	49	42	32	76	914	97
C-JCS	401	139	35	7	4	57	136	98
C-Jexl	675	317	47	8	4	50	295	93
C-Net	1588	902	57	11	7	64	877	97
C-Pool	546	278	51	11	8	73	278	100
C-SCXML	811	348	43	16	8	50	335	96
C-Validat.	639	287	45	16	10	63	279	97
C-VFS	1242	614	49	21	13	62	611	99
CXF-Fediz	1211	190	16	12	6	50	171	90
Drill	995	636	64	84	65	77	616	97
Flume	1194	347	29	45	26	58	342	99
Giraph	585	392	67	33	26	79	387	99
Hama	1582	732	46	23	16	70	717	98
Helix	1521	800	53	31	21	68	762	95
H-Client	1916	934	49	13	9	69	925	99
H-Core	1354	566	42	7	6	86	565	99
Mina	628	276	44	13	9	69	263	95
M-SSHD	1092	843	77	22	21	95	818	97
Nutch	2221	926	42	40	28	70	893	96
OpenNLP	665	438	66	14	14	100	424	97
P-MR	1647	349	21	119	41	34	345	99
Phoenix	1908	1290	68	78	62	79	1260	98
Qpid-JMS	921	431	47	5	3	60	431	100
Ranger	1412	415	29	47	25	53	409	99
Santuario	743	496	67	3	3	100	480	97
Shiro	700	289	41	22	13	59	285	99
Tiles	1345	327	24	10	7	70	318	97
Zookeeper	1339	618	46	31	25	81	610	99
AVG			48			69		98
DEV			14			15		1

Avro is a data serialization system that is implemented in 10 different programming languages. All those implementations are located in one repository. Since we are interested in the Java implementation, we skipped the commits that contain developments for other programming languages. Consequently, the ratios of impactful commits (25%) and impactful developers (49%) are relatively low.

Parquet-MR has an exceptionally low ratios for impactful commits (21%) and impactful developers (34%). Its repository contains multiple subprojects which are the conversion and integration of Parquet (a columnar storage format for Hadoop) with other technologies, such as Avro and Hive. Each subproject is assigned to a couple of reviewers and has its own developers. As a result, in comparison to other projects, a relatively smaller subset of developers contribute to the main module.

CXF-Fediz is a web security framework that has two relatively large and architecturally independent components: cxf.fediz.core (the main module) and cxf.fediz.service.idp. The first one is a federation plugin for web applications. The second one is an identity provider and security token service. Since our focus is on the main module of this system, we did not analyze a proportionally large number of commits that had an impact on the second module. As a result, the ratio of impactful commits are relatively low (16%). While the ratio of impactful developers (50%) is not within the standard deviation of the mean, it still shows a high level of engagement in this module.

C-SCXML and C-Jexl are the only two systems that have relatively low impactful developer ratios (50%, 50%) while their impactful commit ratios (41%, 47%) are within the standard deviation of the mean. By looking at C-SCXML's development history, we found that half of the developers have less than 5 commits within the period considered in this study. All of those commits are none-impactful changes such as adding dependencies or fixing repository's configuration. On the other hand, the three most active developers have committed more than 86% of all commits. We observed a similar pattern in C-Jexl. The two most active developers have committed more than 89% of the commits while half of the developers have 17 non-impactful commits combined. This shows a meaningful distinction between the role of the key developers and others in these two subject systems.

Our analysis reveals that on average half of the commits have an impact on the source code of the main module. In addition, more than two-third of the developers are directly engaged in the development of the main module. Our result suggests that the architecture of the system, the level of its integration with other systems, and the distribution of tasks during the development may significantly affect the ratios of impactful commits and developers.

5.2. RQ2

To answer RQ2, we executed the retrieved compile commands on all revisions created by impactful commits. We call those revisions **"impactful revision"**s. For each system, we counted the number of compilable impactful revisions within its timespan. We also calculated the ratio of compilable impactful revisions to all impactful revisions. Table 3 summarizes the result of our analysis.

We found that on average, 2% of the impactful revisions are not compilable. In 4 subject systems, we could compile all impactful revisions using their default compile

commands. There were 6 subject systems with missing dependencies over a relatively long period of time. After fixing the missing dependencies, we could compile all impactful revisions for one of those systems (Qpid-JMS).

If a commit causes compilation error, we call it an **"error commit"**. We looked at the error commits to further investigate what causes the error. Our investigation reveals a variety of interesting phenomena.

5.2.1. Relying on snapshot versions of dependencies may cause compilation error. We observed that some revisions of our subject systems depend on snapshot versions of their dependencies. Snapshots are what might become an actual release and might get updated frequently.

For example, 28 of Qpid_JMS, 122 of Tiles, 56 of Phoenix, 26 of CXF-Fediz, 28 of Flume, and 13 of C-Validator impactful revisions depend on different snapshot versions of their dependencies (e.g., proton-j, struts-master, hbase, wss4j, hadoop-core, and common-parent). In all of these instances, we resolved the dependency issues by automatically altering the build configuration and using stable versions of those dependencies.

In one exceptional case in CXF-Fediz, APIs in a snapshot version of its dependency are slightly different from the official version. This makes 8 consecutive impactful revisions (from commit e75b76c) completely uncompilable. In this period, one class uses a method call that does not exist in the official release of its dependency. This problem is fixed at d39f7f0 which is dedicated to resolving this issue.

This suggests that using snapshots is not suitable for repeatable build since snapshots may get changed, moved or replaced, particularly when the actual release is ready.

5.2.2. Compiling the project in a new environment from scratch may reveal missing dependencies. One of our most interesting observations was a compilation error that is introduced at Zookeeper commit 1d2a786 and fixed after 52 subsequent commits at 4ee7c1d. In this case, a modification in build.xml file breaks the execution of one the dependencies. However, that dependency will not be triggered unless the system is being built from scratch. The developers had not realized that problem for almost 6 months. We fixed this dependency issue before including Zookeeper's result in our dataset.

In another example, at Nutch commit 0032314 a namespace change happens as one of the dependencies (org.gora) migrates to Apache foundation (org.apache.gora). This causes a missing dependency that have remained in the repository for 8 months until that branch is merged into another branch at commit 0460d89. In that period, we manually pre-compiled Gora to our analysis server. As a result, we could successfully compile all those revisions.

This suggests that compiling a system from scratch can help finding missing dependencies when the build configuration files change.

5.2.3. After a file is added, compiling the project in a new environment may reveal an error. In some cases, we observed that developers forget to stage and commit new files in their local repository. For example, in Giraph commit c33ea10, the compilation error is caused by missing files. The subsequent commit only contains introducing those files.

In Helix commit 917af3e, a relatively large set of modifications affects 60 files in the system. This introduces a compilation error because of some missing classes that lasts for 23 commits over a period of 22 days. The missing classes are added to the system at commit c589fb8.

In Mina-SSHD commit f26b5f7, a relatively large refactoring happens which introduces a mismatch between a class name and its corresponding file name. This error is fixed after 12 subsequent commits over a period of 35 days at commit 070a8e7. The reason may be that the working directory and the remote directory are not completely synced as renamed files are not automatically staged in Git.

This suggests that developers need to compile the new revisions in a new environment other than their own working repository when they add new files.

5.2.4. When contributing to the system alone, do compile the system in a new environment once in a while. We observed that over the period that only one developer works on the system, different types of compilation errors may remain in the repository for several days. For example, at C-Net commit 44cec10c, a small patch containing a setter functions with missing parenthesis is applied to the source code. This typo is fixed 4 impactful commits later at commit b1147fd after 12 days. The fixing commit only contains one line of code, and its message is "Fix typo."

In another example, in C-IO commit 5064788, the compilation depends on the absolute path of a directory in the local machine of one of the developers. The problem remains in the system for 9 subsequent impactful commits over 4 days and is finally fixed at commit 2eb161d. Only one developer is working on the system during this period.

In another instance, at C-Jexl commit 86b3e40, a refactoring causes a name clash which lasts for 9 subsequent commits over a period of 13 days. The error is fixed at commit 000c48d. Only one developer is working on the system over this period.

This indicates that when the level of collaboration decreases, developers may fail to follow good practice.

5.2.5. Committing too early may result in an error commit. We observed that developers may introduce a compilation error when they commit too early. For example, in C-VFS commit 54b824f, the developer intends to increase the performance by avoiding Boolean instantiation in one line of code. While changing that line, a typo (missing parenthesis) is introduced in the source code. In less than a minute, the following commit (26af957) resolves this problem.

In another instance, in C-BCEL commit 9640239, the developer intends to change a comment (according to the commit message). However, two lines of code are modified along with that comment and a method name mismatch

is introduced. Two minutes later, the subsequent commit (676ca07) fixes the problem.

In Hama commit bac5191, a small refactoring (variable rename) causes a compilation error. The compilation error is fixed after 4 minutes at commit 6948f4d which only contains a change in one line of code.

At OpenNLP commit 740d5a5, 10 new files are added to the repository without any modification to the other files. The next two commits over the next two minutes have the same commit message as the first one and contain the missing parts to make the code base compilable.

In some cases, the developers mention that the compilation is broken, still, they commit the changes then fix the problems in the subsequent commits. For example, in CXF-Fediz commit cae5c37, the developer mentions that the changes are not compilable yet. The compilation errors are fixed in the subsequent impactful commit db7b4ea after a day. Similarly, C-Collections commit 67c51ed's message is "Fix compilation error.". The problem is fixed in less than an hour in the next impactful commit 9acc3e8. The message of C-Configuration commit 5aaa012 indicates that the revision has a compilation error. This compilation error is fixed after four impactful commits over a period of 2 minutes at commit 3a771aa.

This suggests that committing too early may result in an error commit.

5.2.6. Committing too often may introduce error commits. In some cases, developers tend to produce multiple error commits in a short period of time followed by a commit fixing the problem. For example, at Santuario-Java commit 1fe126e, the system undergoes refactoring which breaks its source code compilability. The next commit also introduces more refactoring to the already broken source code. Finally, all compilation errors are fixed at commit 8d7cec1. These three commits are pushed to the repository within a period of 2 minutes.

In Mina 02f02e7, a piece of uncompilable code is added and followed by two error commits cleaning up the first one within 2 minutes. The compilation error is finally resolved the next day at commit 2cc1d14.

In C-Net commit 2d5c8f8, a refactoring in order to use an interface instead of an implementation introduces a compilation error which affects the next 12 impactful commits over a period of 2 hours. This compilation error is resolved at c2ceeda. The commit contains multiple changes in the same class file. This indicates that sometimes developers do not compile the code before committing their changes, even if their changes impact multiple lines of code.

The preceding two observations suggest that although it is recommended by some experts to commit early and often[10], committing too early and too often can degrade the quality of the source code by introducing basic compilation errors.

5.2.7. Large refactorings may introduce error commits. At Phoenix commit c98cda7, the system undergoes refactoring which impacts 47 files. This introduces a compilation error because of a type mismatch in overriding an abstract method. The compilation error remains in the system for 10 subsequent commits over a period of 6 days. It finally gets resolved at ad2ad0c which only changes 3 lines of code.

In another example, a relatively large refactoring impacting 11 files breaks the compilation at CXF-Fediz commit 3c0a524c. The problem remains in the system for 4 commits over a period of two days. It finally gets fixed at db7b4ea.

In Drill, commit d068b3e introduces multiple compilation errors after a relatively large refactoring changing 15 files. The next 8 subsequent commits in the next 2 hours are dedicated to mostly fixing those errors.

In C-SCXML, 10 subsequent commits from 094fefe over a period of 2 days introduce different compilation errors. The following commit (1b32987) fixes those errors. The system undergoes refactoring for type safety improvement and removing unnecessary casts.

This observation suggests that developers should compile the source code in a new environment after a large refactoring before producing a new commit.

5.2.8. Maintenance and code cleanups may introduce error commits. At C-Jexl commit 9dcf08d, removing depreciate code causes a compilation error that is resolved after two consecutive commits at 0fae596. In another example, Hama commit 893ba2a introduces a compilation error while removing depreciate code which stays in the repository for 4 subsequent impactful commits. The issue finally gets resolved at 17e0cbb.

In some cases, such as Nutch commit 9714c44, some clean up in the old code introduces a compilation error which is immediately fixed in the next commit, 024b6a6.

We could not compile 3 impactful commits of C-JCS. Interestingly, all of them are related to maintenance. A patch is applied to the main line of code at commit c2b9de1, A simple refactoring happens at cfa533f9. A change in the access modifiers of a class is introduced at 0e55dab.

In C-BCEL commit 25d3c6f, a large number of classes undergo maintenance for adding proper annotations. This introduces a compilation error because of a mistaken double annotation. The extra annotation is removed after 5 subsequent commits at bd70827 in less than an hour.

After a major rewrite in one of the core classes at Nutch commit 45b5cf7, an incompatible type comparison is introduced. This problem remains in the system for 6 subsequent commits over a period of 2 days. It finally gets resolved at a17464e.

In summary, our analysis suggests that, in many cases, developers introduce compilation errors which last for several commits in the development history. Interestingly, we observed that most of the compilation errors that remain in the repository for multiple commits happen when the source code undergoes maintenance, such as refactoring and removing unused/depreciated code.

This shows that developers do not necessary follow good practice when committing their code. We suggest that

10. http://sethrobertson.github.io/GitBestPractices/#commit

developers should compile the code, even if the changes are minimal.

We would like to point out that some revisions of some systems could not be compiled using their default build command. The reason is that either tests or some validation checker are failing. We skipped tests and other validation checkers in our compile command as we are only interested in the source code of the main module. Studying these failing test cases and validation checkers can reveal interesting phenomena; however, it is out of the scope of this study.

5.3. RQ3

To understand the impact of commits on different quality attributes, we executed SonarQube, PMD, and FindBugs on the main module of all compiled impactful revisions. Then we calculated the absolute value of each quality metric. We computed the value difference between each revision and its previous revision to see if the quality attribute changes. We define $All(s)$ as the set of all impactful commits in a system s, $Const(s, m)$ as the set of all impactful commits that do not change the quality attribute m in s, and $Change(s, m)$ as the set of all impactful commits that change m in s. For each subject system, we calculated the ratio of impactful commits that change a quality attribute to all impactful commits:

$$T4(s, m) = \frac{|Change(s, m)|}{|All(s)|} \times 100$$

Table 4 summarizes the result of our analysis.

Our analysis reveals that on average only 70% of impactful commits change the number of physical lines (LC). All systems with exceptionally low LC ratio are Commons systems that are mainly libraries. We further investigated this observation by looking into these systems.

The main reason for the low LC ratio in Commons systems might be that their Java classes are heavily documented using JavaDocs. A lot of impactful commits only change those documentations. In multiple instances, we observed that a small change in a class is followed by several commits fixing JavaDocs. For example, at C-CSV commit c43e8fa, a logic change is followed by 17 subsequent impactful commits, from 10b1110 to 3740067, refining the documentations.

On the other hand, there are some systems with very few impactful commits that are only dedicated to documentations. For example, we studied Calcite for the commits that do not change LC. We observed that only 1.5% of commits change JavaDocs without affecting LC. This suggests that the concentration of development in terms of changing code and documentation can significantly vary among different subject systems depending on their domain (e.g., framework, library, etc.).

One interesting observation is that on average, the ratio of commits that change the number of functions (FN) to the commits that change LC is 50%. All five systems with exceptionally lower ratio of FN to LC are Commons systems.

TABLE 4. THE PERCENTAGE OF IMPACTFUL COMMITS THAT CHANGE A QUALITY METRIC TO ALL IMPACTFUL COMMITS

	Basic			Code Quality			Security		
	LC	FN	CS	CX	SM	PD	VL	SG	FG
Avro	80	44	21	70	47	73	5.3	2.7	2.1
Calcite	89	57	34	75	58	83	4.1	0.9	0.9
C-BCEL	54	16	5	31	45	53	4.6	3.0	4.1
C-Beanu.	47	17	7	27	28	40	0.0	0.7	0.0
C-Codec	42	20	6	23	26	36	0.8	0.5	0.0
C-Collect.	51	22	11	30	29	38	0.9	1.6	1.6
C-Comp.	64	32	10	49	39	54	3.5	2.3	1.8
C-Config.	65	34	9	37	28	43	0.0	0.9	0.9
C-CSV	39	16	3	25	20	32	0.7	1.2	0.7
C-DBCP	56	19	5	35	40	46	2.7	1.6	2.1
C-IO	55	29	8	38	35	45	1.4	1.8	1.9
C-JCS	79	47	17	57	48	66	8.3	3.0	3.8
C-Jexl	64	40	19	56	46	59	4.2	1.7	1.7
C-Net	59	21	6	39	39	51	4.5	1.4	0.3
C-Pool	44	18	8	27	27	36	3.6	0.7	0.7
C-SCXML	74	32	14	52	49	63	4.2	0.0	0.6
C-Validat.	56	12	5	26	36	43	0.4	0.7	0.7
C-VFS	51	21	6	29	31	44	2.0	0.2	1.0
CXF-Fediz	78	39	19	62	50	68	9.3	1.2	3.1
Drill	88	48	29	74	58	80	8.3	1.5	3.2
Flume	88	47	29	66	59	72	7.4	1.2	1.8
Giraph	85	58	39	69	49	74	7.7	2.6	1.6
Hama	79	44	25	60	60	71	18.9	6.3	7.7
Helix	87	47	24	70	67	78	9.0	1.4	2.9
H-Client	73	34	14	54	40	56	2.9	2.0	1.6
H-Core	69	40	17	50	41	54	0.4	1.8	1.2
Mina	79	40	15	58	54	67	9.6	1.2	0.8
M-SSHD	85	52	30	75	58	75	17.0	2.6	3.7
Nutch	81	33	17	62	59	74	10.1	3.2	4.9
OpenNLP	68	39	22	54	56	61	6.8	4.6	4.1
P-MR	76	45	29	63	52	66	1.9	4.8	2.9
Phoenix	84	43	20	72	58	77	10.2	4.0	7.5
Qpid-JMS	75	33	14	60	43	60	5.8	1.9	2.3
Ranger	88	44	14	74	63	81	17.3	0.3	3.8
Santuario	77	35	17	66	54	73	6.7	3.0	4.3
Shiro	70	43	21	52	42	56	2.8	2.1	2.5
Tiles	82	54	28	67	56	68	7.1	0.6	2.9
Zookeeper	83	39	15	71	53	69	8.0	1.8	4.0
AVG	70	36	17	53	46	60	5.7	1.9	2.4
DEV	15	13	9	17	12	15	4.7	1.4	1.8

The reason for this might be that Commons systems have more stable APIs as they are libraries. As a result, most impactful commits change the code in the body of already existing methods, rather than introducing or removing any method.

The ratio of commits that change the number of classes (CS) to the commits that change FN is 44%. Similar to FN/LC, all 7 systems with exceptionally lower ratio of CS to FN are Commons systems. This suggests that libraries typically undergo less architectural changes. In our previous work [15] on architectural evolution, we observed that the architecture of library systems is more stable, in comparison to non-library systems.

The complexity in our study is the number of paths in the main module. Change in the number of paths means change in the control flow of the program. On average, 53% of impactful commits change the complexity (CX) of the main module. This emphasizes the importance of running regression tests after each commit as a change in the control flow may result in failing test cases.

Code smells are pieces of code that are confusing for the

maintainers or give them a pause. A change in the number of code smells (SM) have a direct impact on the maintainability of a software system. On average, 46% of impactful commits either introduce new code smells or resolve already existing ones. This illustrates that developers are constantly changing the number of code smells in their impactful commits. If they do not utilize proper tools to detect and realize new smells, they will need to dedicate more time and effort to maintain the system.

PMD has a broad set of issue categories. We selected a subset of them that are related to basic code qualities. Those categories are listed in Table 1. On average 60% of impactful commits change the issues detected by PMD (PD). We further investigated this by studying our subject systems to see if they actually use PMD or not. Interestingly, 64% of our subject systems have the PMD plugin in their configuration files. Considering that, the high average number of PD illustrates that developers do not necessarily fix PMD issues before every commit. We looked at commit messages to investigate whether developers mention either introducing new PMD issues in the code or removing some existing ones. Only 0.2% of the commits directly mention PMD in their message, most of which indicate fixing issues.

We utilized all three tools to measure the number of security issues. VL is the ratio of impactful commits that change the number vulnerabilities detected by SonarQube to all impactful commits. SG is the ratio of impactful commits that change the number of security code guideline violations detected by PMD. FG is the ratio of impactful commits that change the number of security and malicious code bugs detected by FindBugs.

We observed that VL has a higher ratio on average in comparison to SG and FG. However, there are instances where this does not hold. In two subjects systems with zero VL, at least one of the other two metrics shows change in the number of issues. For example, at C-Beanutils commit e307883, a setter method that stores an array directly and a getter method that returns that array are introduced. This may become a security threat if these methods are called by an untrusted code as the caller can keep a copy of the array and change its content later.

There are other instances that one security metric shows no change in the number of issues over the period that we analyzed the software system while at least another one shows change. This motivates us to investigate whether and how one software quality metric changes while another one is constant.

5.4. RQ4

To find how much a single quality metric reveals about how the system and its quality evolve, for each quality metric m_1, we counted all impactful commits in which m_1 does not change while another metric m_2 changes. Then we calculated the ratio of these commits to the total number of impactful commits in which m_1 does not change:

$$T5(m_1, m_2) = \frac{|Const(m_1) \cap Change(m_2)|}{|Const(m_1)|} \times 100$$

$Const(m)$ is the set of all impactful commits that do not change m; and $Change(m)$ is the set of all impactful commits that change m. Table 5 summarizes the results of our analysis.

TABLE 5. THE PERCENTAGE OF $const(X) \cap change(Y)$ TO $const(X)$

Const	Change								
	Basic			Code Quality			Security		
	LC	FN	CS	CX	SM	PD	VL	SG	FG
LC	-	1.2	0.4	5.5	13.3	17.0	0.7	0.2	0.6
FN	55.1	-	2.0	29.6	31.0	43.2	2.0	0.3	0.7
CS	65.1	24.5	-	45.7	38.4	54.0	3.1	0.9	1.2
CX	40.1	1.8	1.6	-	22.8	30.2	1.2	0.3	0.7
SM	52.6	17.1	3.9	33.5	-	38.8	1.3	0.5	0.6
PD	37.9	6.5	1.7	17.7	16.2	-	1.0	0.3	0.5
VL	69.1	32.8	13.8	51.5	43.7	58.8	-	1.6	1.4
SG	70.2	34.5	15.5	53.1	45.6	60.2	5.7	-	1.4
FG	70.1	34.4	15.2	53.0	45.3	60.0	4.9	0.7	-

For each pair of metrics, there exists at least one impactful commit in which one metric changes while the other one remains constant. This illustrates that no single software quality metric alone suffices to show how software quality changes. Based on this finding, we advise using a combination of software metrics to measure change and ensure software quality during the software development process.

Interestingly, code quality metrics may change frequently, even if LC remains constant. For example, C-SCXML was under maintenance (refactoring) over a period of 7 days. In that period, there are 11 commits (from 9ff8050 to 947b18d) in which at least one of CX, SM, and PD changes while LC remains the same. This recommends that even when the change to the code is minimal, running static analysis techniques can reveal change in quality attributes.

Our results show that when the code quality metrics do not change, the probability of change in one of the security metrics drops significantly. However, there is still a chance that a security issue is added or removed. For example, in C-BCEL commit bfb12d3 the developer changes one line to avoid calling an overridable method from a constructor. However, this creates a security issue because an array being stored directly in the object. None of the code quality metrics changed in this case, but a security issue was created.

On the other hand, when one security metric remains constant, the probability of change in other metrics does not significantly drop. For example, as Table 4 shows, the average number of VL and SG are 5.7% and 1.9%. Those numbers drop to 4.9% and 0.7% when FG is constant. This shows that using a combination of different security metrics can improve software quality since each metric reveals different types of security issues.

Based on our findings, we advise developers to not rely on a single static analysis tool as quality indicator. Using metrics from different tools can reveal software quality issues early in the development process as the system evolves.

6. Related Work

The idea of gaining an insight into the evolution of software systems through mining their repositories is not new. However, due to effort and scalability challenges involved in mining software repositories on a large-scale, researchers are forced to adjust the scope which oftentimes results in coarse-grained analysis, for example, looking at only the latest major or minor releases [1] [2] [3]. Our approach is not limited to study the stable releases of a system but to take a step further by analyzing the state of the system after each impactful commit using 3 static analysis tools to better understand the evolution of software quality.

There are numerous works that attempt to address the challenges in mining repositories which directly influence our work in data collection. Dyer at el. [17] develop a query system specifically for mining repositories called Boa. It can efficiently aggregate and query information by executing tasks on a Hadoop cluster. Although Boa indexes metadata information about projects, authors, revision history and files, it stores no source-code facts [18].

Diamantopoulos et. al [19] develop QualBoa, a recommendation system, built on top of Boa, which helps developers to identify reusable components in the source code. The tool evaluates each component against static analysis tools such as PMD and CodePro AnalytiX to obtain its functional score and reusability index based on quality metrics. In comparison, we also utilize widely-used static analysis tools such as PMD, Findbugs, and SonarQube to assess the quality of a system, as well as, compile each revision created by commits.

Gousios et al. [20] develop GHTorrent, a tool that can help querying information about a system on GitHub through the GitHub REST API. The tool also works in a distributed manner. Although we do not directly use GHTorrent, we use similar approach to collect our data.

Spacco et al. [21] study warnings reported by FindBugs and implemented techniques for tracking warnings across software releases (versions). Their approach is similar to ours in that they focus on the changes of warnings over the development history. However, our approach is more fine-grained in that we study the revision of software before and after each commit rather than each release.

Ruter et al. [22] compare warnings and study the correlation between warnings reported by static analysis tools, such as PMD and FindBugs. However, they do not attempt to study the changes in the warnings overtime.

Multiple studies have shown that there exists a strong correlation between defects identified by a static analysis tool and defects identified by manual inspection [23] [24] [25] [26]. Many researchers claim that static analysis tools should be used to cheaply detect defects and ensure software quality, or developers should incorporate them with other defects detection techniques to maximize the benefits [27] [28] [29]. Our empirical results show that utilizing static analysis tools can reveal issues early in the software development process and increase the software quality.

7. Threats to Validity

Relying on static code analysis tools to measure software quality is an internal threat to validity of our study. The accuracy of our results is highly dependent on the tool results which might have false positives and false negatives. However, we chose three well-known static analysis tools that are widely-used by open source community.

Another internal threat to validity is identifying the main module of the software system. In order to mitigate that threat, in addition to counting the number of Java files in each module, we studied the documentations and architectural facts of every projects. This ensures that the module we study is in fact the main module containing the core functionalities.

The main threats to external validity in our study involve our subject systems. We only studied open source Java software systems. However, we selected one of the most popular open source communities, Apache Software Foundation, that has a wide variety of software domains. In addition, we were unable to find closed-source systems to expand our study on, and some of the tools that we used only support Java.

8. Conclusion

The analyses presented here for open-source software development can be applied by any organization wishing to improve its software and its software engineering, by having its projects gather and analyze the types of data presented here. At the organization level, managers can determine which of its divisions and project types have better or worse quality; which quality attributes are being achieved poorly or well; and how do these correlate with customer satisfaction and total cost of ownership. At the department or project level, managers can better understand which types of projects or personnel contribute most to quality problems or excellence, and which types of project events correlate with which types of quality increase or decrease.

For examples, our study reveals that on average, 48% of all commits impact the main module that contains most of the source code, and 69% of all developers contribute to the source code in that main module. We found that on average, 2% of the impactful commits are not compilable. We investigated when, how, and why developers commit uncompilable code. Our results suggest that the impactful commits have different impact on the software quality metrics. We found that different quality attributes may change even if the code count does not change. On average, only 5.7% impactful commits change VL, 1.9% change SG, and 2.4% change FG. We found that although the security metrics change less frequently, it is crucial to utilize them as they can reveal the introduction of different kinds of security problems.

In the future, we will study how software quality aspects increase and decrease as a software system evolves. Using commit-impact analysis, we will examine differences between developers of a software system in terms of impacting software quality. We will further include dynamic analysis techniques, as well as regression tests, to understand

what other software factors change and why they change. Using fine-grained information that commit-impact analysis provides, we will build defect prediction models. We will utilize machine learning and natural language processing techniques on the commit messages to understand the effects vs. intent of the changes, and how developers respond to the effects of their changes.

9. Acknowledgment

This material is based upon work supported in part by the U.S. Department of Defense through the Systems Engineering Research Center (SERC) under Contract H98230-08-D-0171. SERC is a federally funded University Affiliated Research Center managed by Stevens Institute of Technology.

References

[1] Qiang Tu et al. Evolution in open source software: A case study. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 131–142. IEEE, 2000.

[2] Anita Ganpati, Arvind Kalia, and Hardeep Singh. A comparative study of maintainability index of open source software. *Int. J. Emerg. Technol. Adv. Eng*, 2(10):228–230, 2012.

[3] Marco DAmbros, Harald Gall, Michele Lanza, and Martin Pinzger. Analysing software repositories to understand software evolution. In *Software evolution*, pages 37–67. Springer, 2008.

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

[5] Eric Raymond. The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23–49, 1999.

[6] Barry W Boehm, John R Brown, and Mlity Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, pages 592–605. IEEE Computer Society Press, 1976.

[7] B Mexim and Marouane Kessentini. An introduction to modern software quality assurance. *Software quality assurance: in large scale and complex software-intensive systems. Morgan Kaufmann, Waltham*, pages 19–46, 2015.

[8] Alexandru G Bardas et al. Static code analysis. *Journal of Information Systems and Operations Management*, 4(2):99–107, 2010.

[9] A Campbell. Sonarqube: Open source quality management, 2015. *Website: tiny. cc/2q4z9x*.

[10] D. M. Le, C. Carrillo, R. Capilla, and N. Medvidovic. Relating architectural decay and sustainability of software systems. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 178–181, April 2016.

[11] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[12] M. Langhammer, A. Shahbazian, N. Medvidovic, and R. H. Reussner. Automated extraction of rich software models from limited system information. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 99–108, April 2016.

[13] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.

[14] Scott Chacon and Ben Straub. *Pro git*. Apress, 2014.

[15] Pooyan Behnamghader, Duc Minh Le, Joshua Garcia, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. A large-scale study of architectural evolution in open-source software systems. *Empirical Software Engineering*, 22(3):1146–1193, 2017.

[16] Sonal Mahajan, Bailan Li, Pooyan Behnamghader, and William GJ Halfond. Using visual symptoms for debugging presentation failures in web applications. In *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*, pages 191–201. IEEE, 2016.

[17] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 422–431. IEEE Press, 2013.

[18] Raoul-Gabriel Urma and Alan Mycroft. Expressive and scalable source code queries with graph databases. 2013.

[19] Themistoklis Diamantopoulos, Klearchos Thomopoulos, and Andreas Symeonidis. Qualboa: reusability-aware recommendations of source code components. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 488–491. ACM, 2016.

[20] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. Lean ghtorrent: Github data on demand. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 384–387. ACM, 2014.

[21] Jaime Spacco, David Hovemeyer, and William Pugh. Tracking defect warnings across versions. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 133–136. ACM, 2006.

[22] Nick Rutar, Christian B Almazan, and Jeffrey S Foster. A comparison of bug finding tools for java. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 245–256. IEEE, 2004.

[23] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 580–586. ACM, 2005.

[24] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[25] Gholamreza Safi, Arman Shahbazian, William GJ Halfond, and Nenad Medvidovic. Detecting event anomalies in event-based systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 25–37. ACM, 2015.

[26] Youn Kyu Lee, Gholamreza Safi, Arman Shahbazian, Yixue Zhao, Nenad Medvidovic, et al. A sealant for inter-app security holes in android. In *Proceedings of the 39th International Conference on Software Engineering*, pages 312–323. IEEE Press, 2017.

[27] James A Kupsch and Barton P Miller. Manual vs. automated vulnerability assessment: A case study. In *First International Workshop on Managing Insider Security Threats (MIST)*, pages 83–97, 2009.

[28] Stefan Wagner, Jan Jürjens, Claudia Koller, and Peter Trischberger. Comparing bug finding tools with reviews and tests. In *IFIP International Conference on Testing of Communicating Systems*, pages 40–55. Springer, 2005.

[29] Panagiotis Louridas. Static code analysis. *IEEE Software*, 23(4):58–61, 2006.