Purpose-oriented Study on Commits For Better Software Quality — A Research Agenda

Jincheng He

Department of Computer Science

University of Southern California

jinchenh@usc.edu

***Abstract*—Developing software with the source code open to the public is very common; however, similar to its closed counterpart, open-source has quality problems, which cause functional failures, such as program breakdowns, and non-functional, such as long response times. Previous researchers have revealed when, where, how and what developers contribute to projects and how these aspects impact software quality. However, there has been little work on how different categories of commits impact software quality. To improve open-source software, we propose this research agenda to investigate how its quality is impacted by commits of different purposes. By identifying these impacts, we will establish a new set of guidelines for committing changes, thus improving the quality.**

***Index Terms*—software engineering, software maintenance, software quality, open source software**

I. Introduction

Before explaining any details of this agenda, we introduce the context of this research, including open source software, version control systems and different aspects of software repository mining that should be considered. These aspects include commit impacts, purposes, commit messages, code patterns and software quality. We conclude this section with an outline of the rest of this white paper.

A. Open Source Software and Version Control Systems

Using open-source repositories has long been a common way to develop software. Some of these projects are on an industrial scale. As the scale has grown far beyond the level that an individual can control and manage, how to efficiently conduct quality control and project management is critical.

Most industrial-scale software is developed by iterative contributions from project teams, through ICSM [1], Agile [2], DevOps [3] or other process models. In the iterations, version control systems, such as Git and SVN, play a critical role by enabling and facilitating the concurrent contributions from developers. Each revision, or commitment (hereafter "commit"), contains diffs which are the lines developers change.

These changes can be made by developers from different areas of the world, at different times, have different purposes and have different impacts on the software [4], be they negative or positive. Thus, it is necessary to investigate how these differences influence software quality, and thus to better control the quality during the development and maintenance phases.

Focusing on the different purposes of commits, this white paper investigates how different types of commits impact software quality and propose guidelines for improving.

B. Level of Commit Impacts

In projects, some commits impact software quality more than others. For example, commits that change core modules, which modify system functionalities are more impactful than those that contain only a few lines of documentation fixes.

The level of impact can be defined in various ways to specify what to investigate. For example, in previous studies, researchers have defined impactful commits by whether they are in the core module [5], [6]. We believe that the more critical the commits are, the earlier they should be taken care of, in the sense of quality control and management.

C. Purposes of Commits

While the levels of impact differ, the commits also vary in their purposes. For example, some commits merely add a few lines of documentation or comments to code while others refactor the entire code structure or made module-level modifications.

[TODO: A better transition here.] It is common for developers to upload single-purpose commits[1]. However, in commits where developers refactor code, add new dependencies, or apply minor fixes, the commits tend to grow beyond their intended task. In this case, commits become multi-purpose, and it has been proven in a previous study [4] that multi-purpose commits have negative impacts on software quality, compared to single purposes ones. In addition, it has been shown that some types of commits, such as "feature add", are more likely to have negative impacts on software quality. Thus, it is important to investigate how different types of commits impact various aspects of quality and how they are related to the other metadata of software to create new guidelines for developers, thus help them improve the quality.

To achieve this, we review previous works that categorize commits and find that some of them have produced taxonomies for commit purposes [4], [7]–[12] which we will evaluate and adopt.

D. Commit Message

One critical piece of the project metadata is the commit message. When developers push changes to online repositories, they are required to add commit messages to explain their changes.

These messages provide important clues for understanding the purposes of those commits. As a result, it is important to analyze commit messages to help understand the purposes of the commits and categorize them.

E. Code Pattern

The commit messages are, in effect, summaries from developers about their code changes.

Also, there are works aiming at how to generate code automatically which lead to inventions of code generation tools. Both hand-typed and auto-generated code are related to this research. For hand-written, it is possible to use them to train a prediction model by using NLP techniques to automatically extract keywords from code changes, which explain their purposes. On the other hand, the outcome of this research, for example the guidelines, are useful when developing code generation tools. [TODO: Think about whether to add this application into abstract.]

[1] https://www.freshconsulting.com/atomic-commits/

F. Software Quality

The final goal of the research is to improve software quality. Thus, defining assessment guidelines for software quality is one of the most important issues of this research.

Software quality is evaluated using different tools depending on one's purpose. For example, COCOMO and COCOMO II [13] evaluate software with respect to their cost. PMD [2], SonarQube [3] and FindBugs [4] define metrics based on software metadata and algorithms to evaluate security, vulnerability and bugs. CAST software [5] provides architecture evaluations in addition to other metrics.

In this research, we analyze software quality using tool-based metrics and compilability and will create a new metric showing the category distribution of commit purposes.

G. Outline of This Research White Paper

In this white paper, we evaluate previous work on related areas of this research, such as commit messages, code patterns and software quality representation, as well as propose a purpose-oriented categorizational analysis approach to analyze commits and software quality.

Our goal is to construct a reasonable categorization for commits based on their purposes, and investigate their relations to software quality and consequently find a way to improve them.

The sections of this white paper are organized as follows:

- Section II-A introduces our current data set and additional data we plan to collect for current projects and from new projects.
- Section III discusses previous works on categorizing commits, either by their purpose, size or other criteria and how we can apply them to analyzing how different types of commits impact software quality. Furthermore, the section also discusses the critical issues in the process of categorization that need extra effort to deal with.
- Section IV discusses previous works in analyzing and extracting information from commit messages, auto-generating them, as well as using commit messages to help categorize commits and analyzing software quality.
- Section V illustrates how we can use code changes to help categorize commits.
- Section VI explains our way of assessing software quality, including tool-based analysis and additional metrics.
- Section VII presents the plan and final goals of this research and how this research will serve to improve software quality across the field.
- Section VIII concludes the white paper.
- Section IX lists potential threats to the validity of this research plan.

[2]https://pmd.github.io/
[3]https://www.sonarqube.org/
[4]http://findbugs.sourceforge.net/
[5]https://www.castsoftware.com

II. Data

To conduct this empirical study, we will need a sufficient data set from various open source projects. The data set should contain the basic meta-data of projects and quality metrics reflecting the quality of those projects. In this section, we will introduce our current data set and our plan to extend the data set.

A. Current Data Set

The initial data set was introduced in previous studies [5], [6]. It includes data from 68 projects of Apache, Google, and Netflix and provides compilability information as well as software quality metrics of over 130000 commits, around 30000 out of which are considered as impactful commits, which are commits that developers change code in their core modules.

In addition, we manually tag 314 uncompilable commits and 1600 compilable commits, each of which are cross-validated by at least two different researchers.

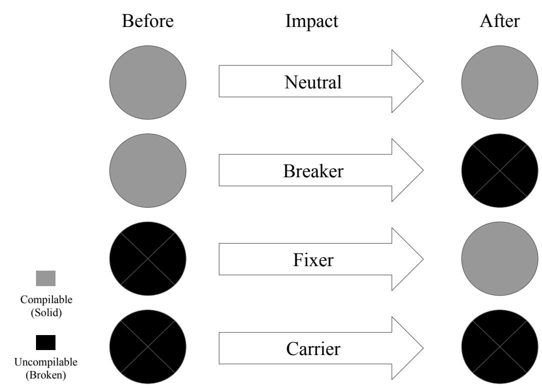

Fig. 1. Four Types of Impactful Commit

We use the following terminology, as shown in Fig. 1, to categorize commits based on their impact on compilability. As we consider one of the quality aspects is compilability, we also define all relevant terms.

- A core module is a module that contains the majority of the source code, such as the main modules in most Apache library systems.
- A commit is *impactful* if it changes a core module. An impactful commit is *broken* if it creates an uncompilable revision; otherwise, it is *solid*.
- A broken commit is a *breaker* if it breaks the compilability of its solid parent; otherwise, it is a *carrier*.
- A solid commit is a *fixer* if it fixes its broken parent; otherwise it is *neutral*. Fig. 2 shows an example of a commit sequence with these four types of impactful commits.

We do not categorize orphan or merge commits into the aforementioned categories their impact is not identifiable by comparing two software revisions (i.e., before and after).

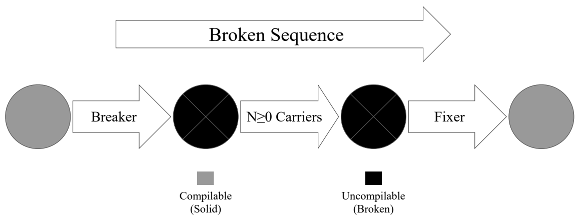

Fig. 2. An Example of a Broken Sequence

For selecting subject systems, we retrieve all Java projects owned by Google and Netflix from GitHub. We select a system only if it 1) requires Ant, Maven, or Gradle for compilation and is not an Android, a Bazel, or an Eclipse project, 2) does not require manual installation of other tools (e.g., Protoc) for compilation, 3) is an official product of the organization, and 4) has a core module which contains a substantial amount of code. We target the core module and identify "impactfuls" in each system and exclude those with fewer than 100 distinct revisions. For selecting Apache subject systems, we use the same criteria as Netflix and Google, except that we only consider subject systems with fewer than 3000 commits by April 2017.

To develop the commit purpose taxonomy, we analyze 100 commits, using both the code and commit message in deriving the categorization. We consult with Hindle et al. to clarify category definitions and determine the categories included in our final taxonomy. In order to apply this taxonomy to small commits, we create further refinements of the definitions for four kinds of tasks.

We use the resulting taxonomy to tag our dataset of 914 commits. Each commit is labelled and cross-validated by multiple individuals. Initial inconsistencies in tagging arise from ambiguities in the original taxonomy. To resolve these ambiguities, we study each inconsistent tag, identify the source of confusion and further refine the taxonomy definitions. The tag definitions are now more narrow in scope, and overlapping meaning between category definitions is reduced. This methodology ensures that our taxonomy can be applied consistently across varied tagging interpretations.

We also define a threshold for identifying commit size, and use this to label each commit. Some related works have introduced thresholds to differentiate between large and small commits; however, our dataset exhibits a more narrow range in commit size. The average commit size is also reduced due to our focus on the changes within a commit, as opposed to cumulative commit size. Thus, based on our dataset, we create a new threshold which defines commit size as the sum of added and deleted lines of code. The 314 uncompilable commits were split into two parts, with the smallest 157 commits labeled *small*, and the remaining 157 labeled *large*. The resulting threshold – 184 lines of code – is used to label the 600 compilable commits. This threshold is based on the distribution of uncompilable commits in order to motivate discussion on the relative size and categorization of uncompilable commits. The final dataset contains 157 each of large and small uncompilable commits, and 138 & 462 large and small compilable commits, respectively.

To analyze software quality evolution over uncompilable and compilable commits, we examine metrics on code complexity, maintainability, and security, provided by SonarQube[6] and PMD[7] and plan to utilize CAST[8]. As uncompilable commits cannot be directly assessed by dynamic software quality analysis, we measure the overall quality change in the compilable commits before and after uncompilable sequences, extrapolating to determine the individual quality of each commit.

B. The Plan to Extend Current Data Set

Choosing the data set is a critical issue for software repository mining research since, if the chosen data set is not representative, the conclusions will not be generally applicable. To avoid this situation, we identify limitations of our current data set and how we plan to extend it as follows:

- The current data set only contains projects from three large corporations, Google, Netflix and Apache. We plan to collect data from other corporations with different coding conventions.
- We focus on impactful commits in the current data set. To resolve this limitation, we plan to investigate non-impactful commits and evaluate whether it is necessary to extend this research to those commits.

III. Commit Change Types

The core of this research is purpose-oriented commit analysis. Thus, the first and most critical problem we plan to address is the establishment of a generally applicable categorization of commits. In this section, we present related works in this problem domain, how we plan to create our own categorization, what we have done to achieve it and what is left to be done.

A. Taxonomies for Commits

Previous works primarily characterize commits based on metadata, including commit size and messages, which are the comments that developers write before pushing the changes to an online repository.

[TODO: Some transitions between sentences.] Purushothaman et al. [14] propose a categorization based on whether a commit adds or deletes lines of code. Alali et al. [8] examine nine open-source software systems to and characterize commit properties by size — lines of code, number of code blocks, and file count — as well as extracted terms from commit messages. Arafat et al. [15] and Hattori et al. [16] also analyze commits categorized by their sizes. Dragan et al. [9] perform categorization over commit code, stereotyping added and removed methods to form a descriptor

[6]https://www.sonarqube.org/
[7]https://pmd.github.io/
[8]https://www.castsoftware.com/

for commit changes. There are also studies which adopt the categories of maintenance tasks, thus the change types, to categorize the commits. We will also categorize commits in this way and conduct our analysis. The reason is that a previous study [4] has shown there is a statistically significant relation between the change types and the software quality by analyzing the compilability.

B. *Purpose-oriented Categorization*

In this research, we will conduct a purpose-oriented categorization for commits based on previously-established categorizations for maintenance tasks. Swanson [10] introduced maintenance task categories by dividing the work from developers into adaptive, corrective, and perfective. Purushothaman et al. [14] add one more category, "inspection," based in addition to previous three.

[TODO: transition] Wang et al. propose a categorization, shown in Table I, based on the purpose of commits.

Kaur et al. [17] propose a classifier for labeling commit type — bug repair, feature addition, and general — based on commit messages. However, commit message alone is not enough to effectively categorize commits. While a commit message can indicate developer intent, it will not necessarily address all the major code changes.

TABLE I
WANG'S CATEGORIZATION

Types of widespread changes	Description
Argument Addition/Removal	Add or drop argument(s) for a method.
Argument Change	Change the value, type, name of the argument(s) for a method.
Method Addition	Extract a piece of codes to be a method, or add methods to get or set variables.
Method Change	Change the name, access control, return type for a method or change deprecated methods.
Data Structure	Change the name, access control or the field of data structure, or change the data structure to be used(e.g., Change map to set).
Feature Addition	An addition/implementation of a new feature.
Algorithm Change	Algorithm or implementation change.
External Change	The change caused by the change of external dependent library or interacting system.
Non-functional	Other kinds of code changes that do not change the code functionalities, documentation change, comment change, or fix warnings.
Bug Fix	Fix a bug. It includes all kinds of code changes that are made for the purpose of fixing a bug.
Others	Except above categories.

Solely focusing on code to determine commit characteristics can often introduce additional noise. While commit messages are brief summaries indicating the central goal of the commit, code diffs capture details that do not affect high-level categorization. Further, determining commit type, based entirely on code, becomes difficult on extremely large commits, where a commit message may provide all of the information needed.

Hindle et al. [7] propose a taxonomy based on the maintenance tasks to categorize large commits. This categorization uses maintenance attributes first introduced by Swanson et al. [10]. They further map the categories to the taxonomy of Mauczka et al. [11], dividing changes into high-level classes of software maintenance: *adaptive*, *perfective*, and *corrective*. The taxonomy is then used for automatic categorization [12]. Based on the results and methodology, our work proposes a refinement of the taxonomy presented by Hindle et al, adapted to reduce ambiguity between categories, and to support tagging small commits.

C. *What we have done — A Refined Categorization*

In our previous study, to analyze the difference in purpose between breakers and neutrals, we needed an accurate categorization for commits. Although we leveraged Hindle's categorization, it was originally designed for usage only on large commits. Several change types, such as maintenance, bug fix, debug, and refactoring were too broadly defined when applied to smaller commits, and at times needed to be accompanied by other categories to accurately classify the commit change.

Our analysis leads to a refinement of Hindle's categories, and the identification of the commits that warrant defining new categories for comprehensive classification. These modifications result in variations in category frequency from those found by Hindle et al. [7]. This indicates, in part, significant changes in category definitions. For example, out of the 2000 commits that they analyzed, 2% were tagged as *maintenance* while in our analysis it is more than 40%. This result is surprising, as other studies [18]–[20] report that 75% of software development budgets are dedicated to maintenance. Another contributing factor for the variation in category frequency is our inclusion of small commits. Along with the new taxonomy, we provide results that describe the relationship between commit size and purpose.

Table II shows the refined taxonomy. Specifications for types are as follows:

Bug Fix: In small commits, the lack of descriptive code makes it hard to differentiate between *bug fix* and *maintenance*. As a result, we have to rely on commit messages and project change logs. For example, if a commit message mentions fixing an existing issue, it should be tagged a *bug fix*. We also thoroughly examine previous software revisions to see if the commit is added in response to flaws in recent code. While we take these steps to better identify bug fixes, some commits are still too vague to be clearly tagged *maintenance* or *bug fix*. Further improvement and definition of rules are needed to make the tagging more definitive on small commits.

Feature Add: We found that Hindle's definition of a Feature Add is under-specified for smaller commits. This may be because new features are easily identifiable in larger commits via the commit message, whereas in small code changes, new

TABLE II
REFINED CATEGORIZATION

Type	Hindle's Definition	Our Explanation
Branch	If the change is primarily to do with branching or working off the main development trunk of the version control system.	
Bug fix	One or more bug fixes.	A change that is reported in the developing log, change log or with expressions in commit message that indicates it is a correction of unexpected behavior.
Build	If the focus of the change is on the build or configuration system files. (such as Makefiles).	
Clean up	Cleaning up the source code or related files. This includes activities such as removing non-used functions.	
Legal	A change of license, copyright or authorship.	
Cross	A cross cutting concern is addressed (like logging).	
Data	A change to data files required by the software (different from a change to documentation).	
Debug	A commit that adds debugging code.	
Documentation	A change to the system's documentation.	
External	Code that was submitted to the project by developers who are not part of the core team of the project.	
Feature Add	An addition/implementation of a new feature.	New function/methods/class implemented, impacting functionality.
Indentation	Re-indenting or reformatting of the source code.	
Initialization	A module being initialized or imported (usually one of the first commits to the project).	
Internationalization	A change related to its support for languages other-than-English.	
Source Control	A change that is the result of the way the source controls system works, or the features provided to its users (for example, tagging a snapshot).	
Maintenance	A commit that performs activities common during maintenance cycle (different from bug fixes, yet, not quite as radical as new features).	Functional changes without feature add and do not have evidence to be considered a bug fix, including performance Improvement and feature extension.
Merge	Code merged from a branch into the main trunk of the version control system; it might also be the result of different and non-necessarily related changes committed simultaneously to the version control system.	
Module Add	If a module (directory) or files have been added to a project.	
Module Move	When a module or files are moved or renamed.	
Module Remove	Deletion of module or files.	
Platform Specific	A change needed for a specific platform (such as different hardware or operating system).	
Refactoring	Refactoring of portions of the source code.	Relocation of part of code. Restructuring. Extract a part of code out of a function and create a new function.
Rename	One or more files are renamed, but remain in the same module (directory).	
Testing	A change related to the files required for testing or benchmarking.	
Token Replace	An token (such as an identifier) is renamed across many files (e.g. change the name or a function).	
Versioning	A change in version labels of the software (such as replacing "2.0" with "2.1").	

features can be a single non-utility function or method. We include this in our refined definition of Feature Add.

Maintenance: This is the most under-specified type in the original taxonomy. Cross validation indicates more than half of the commits with inconsistent tags are tagged *maintenance*. Thus, we consulted with Hindle regarding this tag and decided it should be translated as "minor perfective changes" which is different from the common definition of maintenance. [TODO: Rethink about how to write this part. To make the definition clear. Mention different people understand this tag in different ways, causing this inconsistency.]

Since the original definition can was unclear [TODO: Some more details of why it is unclear], we define it as function & efficiency improvements, additions of utility functions, or minor modifications without new methods or error corrections. For example, memory cleaning is tagged as Maintenance because it is a performance improvement.

Refactoring: We define *refactoring* as relocating or restructuring of code without altering how the code functions, such as extracting code to form a new function, relocating code hunks within or across files, or other changes to reduce code redundancies. They do not change functionality or efficiency.

Note that by our definition, breakers and neutrals can not be merge or orphan commits, as we study the impact of a commit by comparing two revisions: the one it changes and the one it produces. As a result, we did not have commits that were assigned the tags *branch* or *initialization*.

1) Independent Change: To improve our tagging, we first note that the original change type definitions overlap. For example, a documentation change may occur as part of a feature add commit. This increases tagging ambiguity and complexity, which we solve by introducing the concept of an *independent change*.

A change is independent if it is not a part of other changes. Fig. 3 explains this concept. The figure illustrates a commit with multiple changes in which a new feature contains three sub-changes. Each block represents a code change. In the figure, a new feature change contains debugging code, Documentation 1 for the functionality of the feature, Documentation 2 for the debugging code, an irrelevant Documentation 3, and an irrelevant maintenance change. In this case, the debugging code and Documentation 1 and 2 are sub-changes to the new feature. As a result, we only assign a "Feature Add" tag to the new feature instead of assigning tags to all sub-changes. As Documentation 3 and maintenance are both unrelated to the new feature, they will be assigned *documentation* and *maintenance*. The new feature, Documentation 3, and maintenance groups are independent changes with the associated three independent tags forming a label the commit. With the introduction of independent change types, we reduced the ambiguity in the initial taxonomy, resolved cross-validation errors, and improved tagging efficiency.

2) Single-tagged Commit And Multi-tagged Commit: When reviewing manual tagging, we analyze commits with different numbers of tags separately by dividing them into two groups: *single-tagged* and *multi-tagged*. *Single-tagged* commits con-

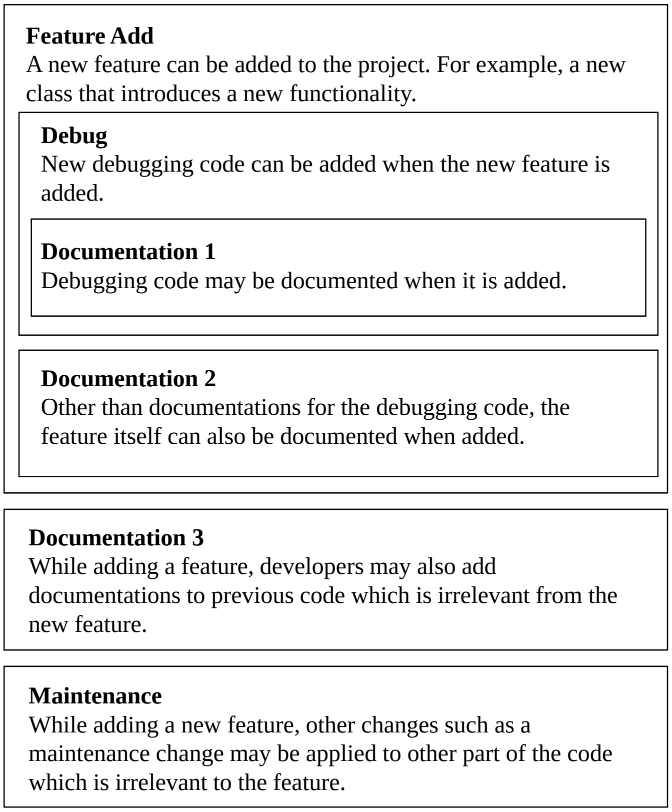

Fig. 3. Explanation For Independent Changes

tain only one independent change, and *multi-tagged* more than one.

Fig. 4 shows the tag-distributions of single and multi-tagged commits by purpose. In the figure, black bars represent the rates when a certain commit is tagged a single category while gray bars are used to denote multi-tagged. For example, approximately 80% commits are tagged Testing and more than 95% of them are multi-tagged commits. As shown in the figure, Build, Feature Add, Indentation, Refactoring, Testing arise more frequently in multi-tagged commits, indicating these tags tend to appear together with other tags. More than 95% of testing commits have multiple tags, which is consistent with development practices of including the testing code along with most changes. Documentation is more frequent in single-tagged commits, indicating that many documentation changes are added independently. As the change of documentation is usually used to explain changed code, this may also imply that contributors often forget to add enough documentation when they make a change. For other categories there are no apparent differences between single-tagged and multi-tagged commits. In addition to comparing these two sets, we also study how many tags each commit has and its distribution.

Recall that we call a commit with multiple independent changes *multi-tagged* and a commit with only one independent change *single-tagged*.

Almost 50% of our analyzed commits are multi-tagged. We summarize the number of tags for each commit independently for the breaker and the neutral set, as shown in Fig. 5. The figure shows the distributions of commits with different numbers of tags in the breakers or neutrals.

For example, 55% of neutrals only have one tag. The curves are Kernel Density Estimations of these two distributions, indicating they are long tail distributions. The number of instances decreases as the number of tags increases. As for its relation with compilability, we can see a neutral is more likely to have a few tags than a breaker. We also use the Kernel Density Estimation (KDE) to estimate these two distributions. The distribution density curve of neutrals is sharper than the breakers' which means the proportion of commits decreases more sharply in the neutrals than in the breakers with increase of the number of tags. We interpret it as that a commit is more likely to break its compilability when it has more tags.

3) Tag Distribution: We analyze the differences between the breakers and the neutrals in terms of tag distributions. Recall that we call a commit a breaker if it breaks the compilabilty of a compilable revision. If a commit changes an uncompilable revision and produces another uncompilable revision we do not consider it as a breaker. We also call a commit a neutral if it changes a compilable revision and produces another compilable revision. If a commit fixes compile errors of an uncompilable commit we do not consider it as a neutral.

Fig. 6 shows the tag-distributions of breakers and neutrals with regard to their commit purpose. Each bar stands for the occurrence rates of that category. For example, around 40% of breakers and neutrals are tagged Testing. Since a commit could have multiple independent changes, values for one color may add up to more than 100%. The presence ratios for *bug fix* and *documentation* are higher in the neutrals while the ratios of *feature add*, *build*, *refactoring*, *clean up* and *maintenance* are higher in breakers. The breakers have 1.90 tags on average while neutrals have 1.56.

Again, we perform odds ratio tests (the Fisher's Exact Tests) to study the correlation between categories and compilability. According to the test results shown in Table III, we found that *bug fix*, *build*, *documentation*, *feature add*, *maintenance*, *module move*, *module remove*, *refactoring*, and *rename* have significant differences between breakers and neutrals, which are consistent with the results of Fig.6.

4) Small And Large Commits: We also analyze the relation between uncompilability and whether a commit is large or small. We draw a boxplot of changed Lines of Code (LOC) for breaker set and neutral set which is shown in Fig. 7. Fig. 7(a) shows the distributions of neutrals and breakers respectively. (b) shows distributions of commits with different number of tags respectively. In boxplot, each box represents the inter-quartile range (IQR) of data points, the range of major part of data. The circles represent outlier of data points which are outside 1.5 IQR. As shown in Fig. 7(a), the distribution of breakers shifts upward from the distribution of neutrals and it means breakers tend to have larger changed LOC than neutrals.

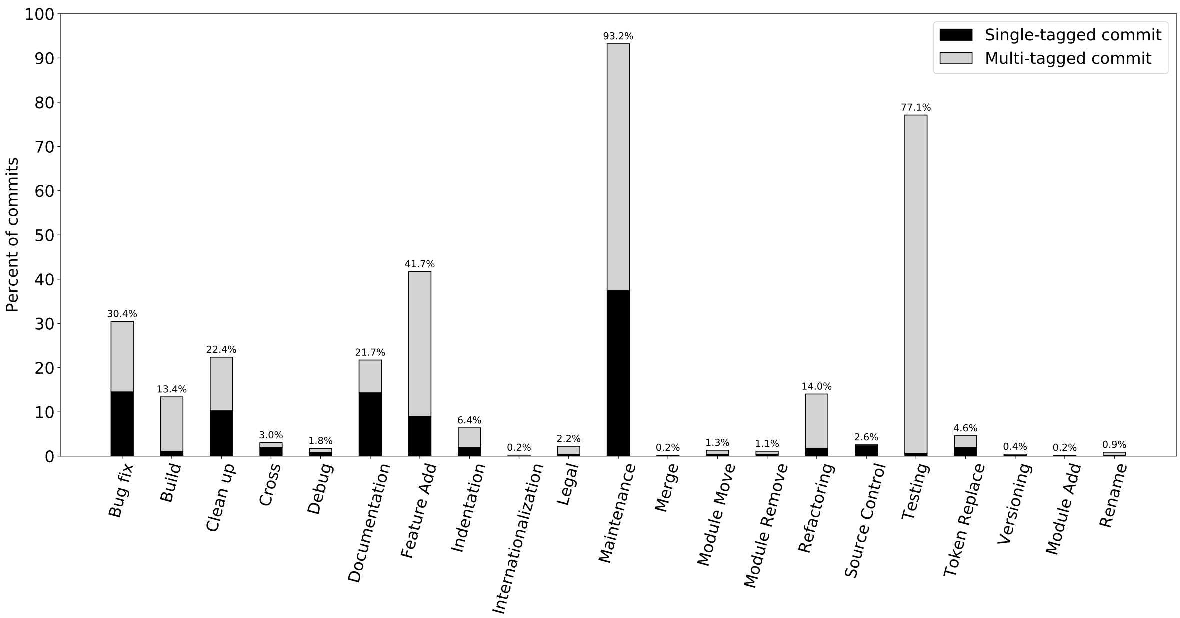

Fig. 4. Change Type Distribution Between Single-tagged and Multi-tagged Commit

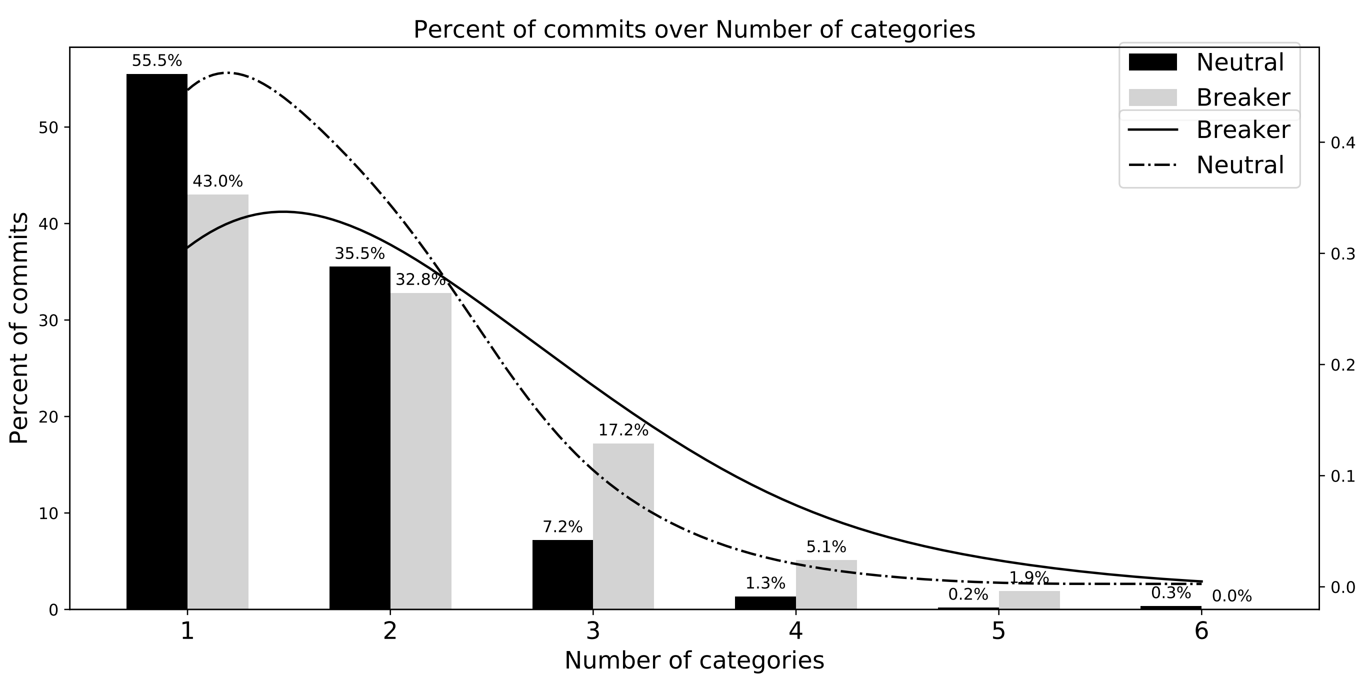

Fig. 5. Compilability With Number of Tags in Each Commit

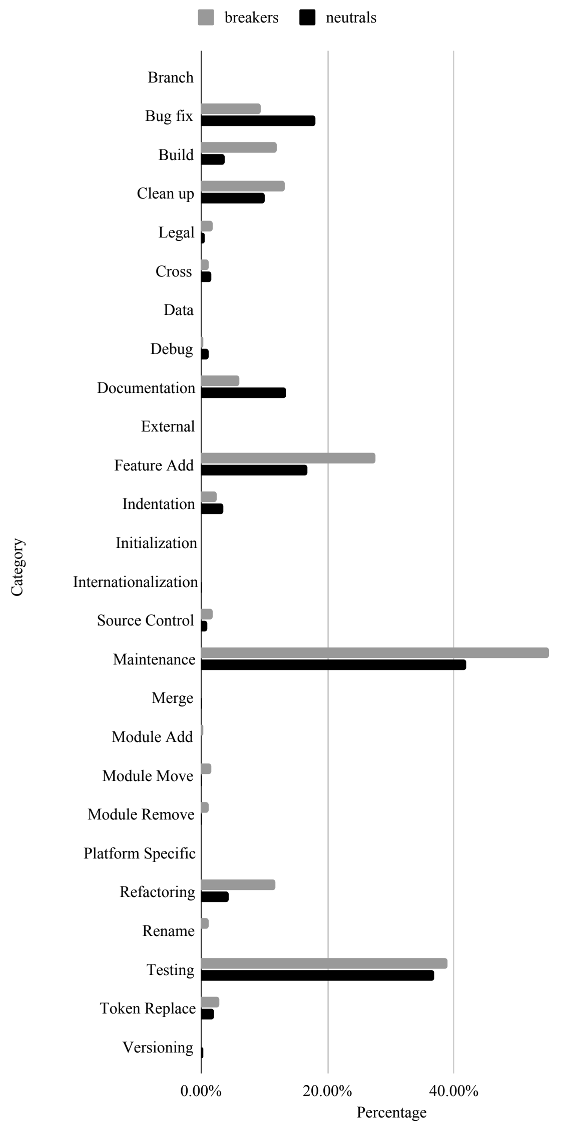

Fig. 6. Tag-distribution for Neutrals and Breakers

This indicates larger commits are more likely to result in compilation breaches.

We also analyze the distribution of changed LOC (Lines of Code) for commits which contains different numbers of tags as shown in Fig. 7(b). The results show that if a commit contains more tags, it tends to have more lines of code changed.

D. Automated Tagging

An important application for which we establish the categorization is to automatically tag the commits based on their changes types. Only if we succeed in tagging the commits efficiently and accurately will we be able to provide potential risk evaluations for them. In this way, it will be possible to integrate our work into existing software development tools. Previous studies have created prediction models to achieve this based on their own categorizations. [TODO: Add some transitions, and explain more about how we should care about these works.] Hindle et al. [12] build their model based on commit messages and author identities. Yan et al. [21] present a discriminative Probability Latent Semantic Analysis(DPLSA) model for automated categorizing. Levin et al. [22] introduce their novel method to predict three types of maintenance tasks. Mariano et al. [23] adopt XGBoost, a boosting tree learning algorithm for classification. Honel et al. [24] achieve a high accuracy by adding code density to their prediction model. Dos et al. [25] combine natrual language processing techniques to help train their machine learning model. Ghadhab et al. [26] apply deep neutral network classifier and BERT model to predict the categories.

TABLE III
RATIO OF COMPILABLE COMMITS OVER NEUTRALS AND BREAKERS

Category	Compilablity Neutral(%)	Breaker(%)	p-value
Branch	0.00	0.00	-
Bug fix	18.17	9.55	**0.000**
Build	3.67	12.10	**0.000**
Clean up	10.00	13.38	0.150
Cross	1.67	1.27	0.781
Data	0.00	0.00	-
Debug	1.17	0.32	0.276
Documentation	13.50	6.05	**0.005**
External	0.00	0.00	-
Feature Add	16.83	27.71	**0.000**
Indentation	3.50	2.55	0.552
Initialization	0.00	0.00	-
Internalization	0.17	0.00	1.000
Legal	0.67	1.91	0.101
Maintenance	41.83	55.10	**0.000**
Merge	0.17	0.00	1.000
Module Add	0.00	0.32	0.344
Module Move	0.17	1.59	**0.020**
Module Remove	0.17	1.27	**0.050**
Platform Specific	0.00	0.00	-
Refactoring	4.33	11.78	**0.000**
Rename	0.00	1.27	**0.014**
Source Control	1.00	1.91	0.358
Testing	36.83	39.17	0.518
Token Replace	2.00	2.87	0.486
Versioning	0.33	0.00	0.549

Their models achieve high accuracy but the categorizations they adopted are too simplistic to clearly characterize all types of commits. For example, Levin's model uses only three categories: "adaptive," "corrective," and "perfective" while we have 27.

As we want to propose our own more complicated categorization, it will be necessary to construct a new model for automated categorization.

E. Research Challenges in Categorizing Commits

Work has been done to categorize commits in different ways and apply the categorizations to train prediction models. However, there are problems that remain unresolved in categorizing commits. The major one is the ambiguity of the terminology.

For example, the term "bug," used in some of the categorizations we mentioned, was first coined by Grace Hopper, used to describe a failure in an early electromechanical computer

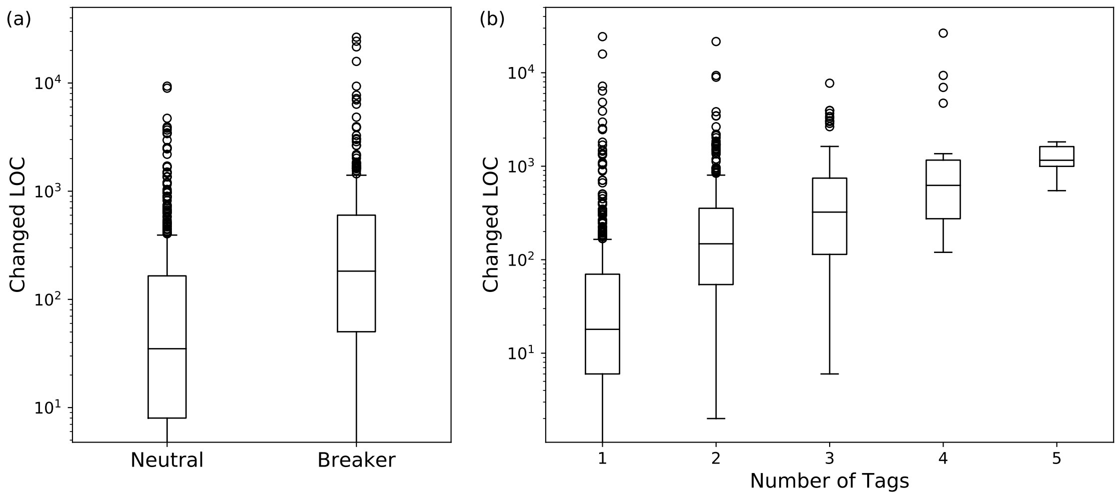

Fig. 7. Commits Distribution Over Lines of Code.

back in 1946. Later, the term “bug” began to be used in software engineering to describe an error, flaw or fault in a computer program or system that leads to an incorrect or unexpected result, or to behave in unintended ways. While this term aims to clarify, it often leads to further confusion. The word “bug” may be divorced from a sense that a human being caused the problem, and instead implies that the defect arose on its own. Sometimes, it is better to describe a mistake in software more accurately using other terms in software mistake metamorphism to avoid ambiguity. [TODO: Rethink about above two sentences.]

This problem also arises in other categories, such as the “maintenance” in Hindle’s categorization. In our cross validation, the term “maintenance” caused the most confusion.

Thus, the core of this research is a more fine-grained and less ambiguous categorization. Instead of relying majorly on commit messages, we will characterize code changes and use them as the major source for categorization, while also using commit messages and meta-data as supporting materials.

IV. Commit Message

One of the most important sources of information to help us classify commits is the commit message. Commit messages are explanations provided by developers when they push their changes to shared developing repositories. The messages include the rationales of those changes and help others to understand what the developers did. Thus, the commit messages are important materials that research can use to help categorize commits. In fact, in the most studies we mentioned in the previous section, they use commit messages as a major resource for automating classifying commits. Beyond this, there are works in which researchers automate generating commit messages for commits to support development. We believe this is also an important application for this research, since a well-established categorization will provide important and well-structured information of what the commits do.

A. Use Commit Messages to Classify Commits

We will use code as our major source for categorization. But we will also make use of commit messages. One of the major challenges in this step will be removing useless information. To achieve this, we will apply natural language processing techniques at this stage.

B. Automated Message Generation

There are existing works for automatic commit message generation. For example, Cortes et al. [27] introduce Change-Scribe approach for automatic message generation and Linares et al. [28] demonstrate the approach. Jiang et al. [29] automatically generating commit messages from code changes by applying neural machine translation techniques.

However, they don’t have a core, well-grained categorization to support message generation. Our categorization will provide support for it.

V. Linking Code to Commit Categories

As we mentioned in Section III, we will use code changes as our major source for manual and automated classification, because code changes are the most direct information about what the commits do. While commit messages provide information about the initiatives of developers, the code changes provide direct information about what they exactly do. There is a gap between the initiatives and the exact work done. When we review the commits during manual categorizing, we find inconsistency between the code changes and what they report in their commit messages. For example, a developer may make a small change and a major change but only report the major change. However, the unexpected small one may introduce a defect in the software. But as it is not included in the commit message, it will be hard for maintainers to track this kind of changes. We will investigate into this as an important step in our research to demonstrate the importance of our categorization work, thus the importance of establishing a good categorization, linking it to every single code change, and supporting the development and maintenance.

VI. Software Quality

The final goal of this research is to improve software quality. Either the automation of the classification of commits or commit messages will in the end contribute to improved maintenance process, thus higher software quality. In this section, we will introduce how we evaluate software quality in our research.

A. Tool-based Quality Metrics

Researchers use static analysis tools and other tools to assess the quality of software. For example, PMD, SonarQube and FindBugs give basic statistics of software, such as the lines of code, the number of functions, as well as quality-related metrics, such as vulnerability and code smell. CAST software provide additional architecture evaluation. We will evaluate software quality by using these metrics.

B. Other Independently Defined Quality Aspects

The software quality and quality metrics are not completely shown by the tools. For example, they can't evaluate the quality when the software are not compilable. And the compilability is one of the most basic quality software should possess.

1) Compilability: A software revision created by a commit is expected to be compilable. However, uncompilability can occur due to careless development — failure to compile the software locally prior to pushing to the shared repository. It can also result from variations in build environments, incompatibility across overlapping changes made by multiple developers, or changes in upstream dependencies. The presence of compile errors inhibits bytecode and dynamic software analysis, as well as static analysis when the code is unparsable [5]. Previous studies [5], [30]–[32] have shown that even in popular open-source projects maintained by major software organizations, build-breaking commits can occur.

Behnamghader et al. [6] explore the qualitative properties of uncompilable commits. Further insight into the types of commits that most frequently cause build errors can help to inform better development practices. Additionally, this correlation can be used as a part of future methods for predicting and preventing uncompilable commits. Analyzing the degradation of software quality over multiple uncompilable commits highlights the long-term negative impact of careless development, and the importance of fixing build errors immediately after they take place. Also, understanding the purposes of those uncompilable commits can result in preparing guidelines to avoid such degradation in the future.

Multiple recent studies [5], [6], [30]–[34] have assessed the compilability of software repositories. To our knowledge, none address the effects of developer purpose on uncompilability, or how software quality evolves when the code is uncompilable. Hassan et al. [30] focus on automatically building the last commit for the top 200 Java repositories on GitHub. Seo et al. [33] analyze 26.6 million builds produced over a period of 9 months by Google engineers, reporting the build failure frequency and cause, as well as how long it takes to remediate. Hassan et al. [31] propose a build-outcome prediction model, based on combined features of build-instance metadata and code changes, to predict whether a build will be successful. Macho et al. [34] identify 125 commits in 23 repositories that repair a missing dependency, qualitatively and quantitatively analyze how the fix is applied, and propose an approach to fix dependency build breakage automatically. Tufano et al. [32] study the compilability of 219,395 snapshots of 100 Java projects from the Apache Foundation, analyzing the frequency and possible causes of broken snapshots. Benamghader et al. [6] qualitatively study why developers commit uncompilable code, and design an approach [5] to increase compilation ratio over commit history and to study sequences of uncompilable commits in terms of their length and interval.

To understand how software quality evolves over uncompilability, we identify all the broken sequences in our dataset. Each broken sequence starts with a breaker and ends with a fixer. The broken commits in the sequence cannot be analyzed using software quality tools. However, we can compare software quality between two revisions to understand how software quality evolves over the sequence: the revision produced by the impact-parent of the breaker and the one produced by the fixer. The former is the last solid revision before the sequence begins and the latter is the first solid revision after the sequence ends.

C. Our Approach

In our research, we will combine the quality metrics with other quality aspects, such as compilability to reflect the overall quality of the software. We will investigate how different categories of changes impact the quality and how we can avoid the defects.

VII. Research Plan and Goals

In previous sections, we present the data set and tools we will use in this research, investigate the sub-areas related to this research topic, and identify the corresponding research challenges. To resolve the challenges, we come up with the following research plan and explain the benefits of this research:

A. Stage One

Few researchers have studied the correlation between the change type and the code, or how change type impacts quality. In this stage, we start by refining the existing categorization of commit changes in open source software repositories. We evaluate the quality of those changes by obtaining quality metrics from static analysis tools, to demonstrate the importance of this research. To assess the correlation between the quality and the categories, we will train a machine learning model, in addition to applying standard mathematical correlation analyses.

B. Stage Two

In this stage, although we have categorized the commit changes, further work distinguishing between different categories is required. This is because high-level categories overlap. In this stage, we will remove the ambiguity of the categories by analyzing the code changes within the commits rather than the commit messages and manual categorizing. Once this is done, we will investigate the correlation between the categories and changes in code to reveal whether they correlate and how those changes impact software quality.

C. Stage Three

In the final stage, we will apply our contributions in different ways. We will construct guidelines which will help developers develop software as well as automate commit message generation and automate classification to support development and maintenance. In addition, we will create an index which explains how different code patterns impact the quality. We will conclude this research by releasing them.

D. Feasibility

This project is feasible because the requisite data, tools and techniques are readily available:

- Data: Open-source software and Git provide sufficient meta-data from Google, Netflix and Apache projects.
- Tool: PMD, SonarQube, FindBugs and CAST provide various quality metrics.
- Techniques: Machine learning and natural language methods.

With all above, we believe this plan will succeed in three to five years. The midterm milestone is the reasonably high prediction accuracy from the machine learning model, which indicate the categorization is of high quality. The final milestone is the releases of the new, systematic coding standard and development guidelines and automation tools.

E. Benefits

We will be able to provide guidelines on how open-source software developers, when contributing to projects, can improve quality. In addition, the results of the second stage will allow us to provide more reliable coding standards and will improve overall code quality. Improved quality will help to reduce cost and improve software service quality.

F. Intellectual Advancement

The first goal of this research is to make concrete improvements in the quality of open-software by providing guidelines for developers and, thus, to improve code quality. We believe this will also change the way people think, code and develop software.

VIII. Conclusions

In this paper, we first explain the context and the major goal of this research. Then, we present the data set we currently have and how we will extend it to make our conclusion generalizable. After that, we introduce the three aspects of this research: categorization, commit message and code and how we plan for different phases of this research and explain the benefits.

IX. Threats to Validity

We discuss the threats to the validity of our empirical study based on the guidelines by Wieringa et al. [35].

External Validity. The main threat is our subject systems. We study 1) a limited number commits from 2) a limited number of 3) open-source 4) Java systems. To address 1 and 2, we use a data set that contains all uncompilable commits among 68 subject systems that are selected from a variety of domains. To address 3, although we do not have access to close source projects, we select major for-profit and nonprofit organizations. Further research needs to be done to assess the generalizability of our conclusions for systems developed in other languages.

Conclusion Validity. The main threats are related to the manual inspection of the compilability of commits and the manual tagging of commits based on their purpose, which are subject to human error. In order to mitigate, two authors have crossed examined to confirm the uncompilability of software over the periods that we report it is broken. Also, four authors have done cross-validation of tagging on 100 commits, and two have done cross-validation on all 1914 commits.

Internal Validity. Relying on Hindle's categorization which has ambiguities can be a threat to internal validity. We use this categorization since it is highly cited and is the most relevant to our analysis. At the same time, we succeeded in addressing the ambiguities with further refinement in this research. Using static analysis techniques that may have false positives and false negatives in measuring quality metrics is another threat. To mitigate this threat, we employ two well established and widely used open-source techniques (PMD and SonarQube).

Construct Validity. The main threat is that we do not study the effectiveness of the refined taxonomy in comparison with other existing taxonomies. This is partially because the other existing taxonomies are not applicable to categorize both small and large commits in terms of their purpose. To mitigate this threat, we were in contact with the authors of the original taxonomy and cross-examined the ambiguities we found.

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 No. HQ0034-13-D-0004 Research Task WRT 1016 – "Reducing Total Ownership Cost (TOC) and Schedule." SERC is a federally funded University Affiliated Research Center managed by Stevens Institute of Technology.

In addition, we thank Dr. Abram Hindle, from University of Alberta, for his helpful comments and thank Dr. Barath Raghavan, Chuizheng Meng, Pooyan Behnamghader, Elaine Venson, Mary Ann Murphy for their valuable feedback and advice.

References

[1] B. Boehm, J. A. Lane, S. Koolmanojwong, and R. Turner, *The incremental commitment spiral model: Principles and practices for successful systems and software*. Addison-Wesley Professional, 2014.

[2] D. Cohen, M. Lindvall, and P. Costa, "An introduction to agile methods." *Adv. Comput.*, vol. 62, no. 03, pp. 1–66, 2004.

[3] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "Devops," *IEEE Software*, vol. 33, no. 3, pp. 94–100, 2016.

[4] J. He, S. Min, K. Ogudu, M. Shoga, A. Polak, I. Fostiropoulos, B. Boehm, and P. Behnamghader, "The characteristics and impact of uncompilable code changes on software quality evolution," in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, 2020, pp. 418–429.

[5] P. Behnamghader, P. Meemeng, I. Fostiropoulos, D. Huang, K. Srisopha, and B. Boehm, "A scalable and efficient approach for compiling and analyzing commit history," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '18. New York, NY, USA: ACM, 2018, pp. 27:1–27:10. [Online]. Available: http://doi.acm.org/10.1145/3239235.3239237

[6] P. Behnamghader, R. Alfayez, K. Srisopha, and B. Boehm, "Towards better understanding of software quality evolution through commit-impact analysis," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, July 2017, pp. 251–262.

[7] A. Hindle, D. M. German, and R. Holt, "What do large commits tell us?: A taxonomical study of large commits," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, ser. MSR '08. New York, NY, USA: ACM, 2008, pp. 99–108. [Online]. Available: http://doi.acm.org/10.1145/1370750.1370773

[8] A. Alali, H. Kagdi, and J. I. Maletic, "What's a typical commit? a characterization of open source software repositories," in *2008 16th IEEE International Conference on Program Comprehension*, June 2008, pp. 182–191.

[9] N. Dragan, M. L. Collard, M. Hammad, and J. I. Maletic, "Using stereotypes to help characterize commits," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, Sep. 2011, pp. 520–523.

[10] E. B. Swanson, "The dimensions of maintenance," in *Proceedings of the 2Nd International Conference on Software Engineering*, ser. ICSE '76. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, pp. 492–497. [Online]. Available: http://dl.acm.org/citation.cfm?id=800253.807723

[11] A. Mauczka, M. Huber, C. Schanes, W. Schramm, M. Bernhart, and T. Grechenig, "Tracing your maintenance work – a cross-project validation of an automated classification dictionary for commit messages," in *Fundamental Approaches to Software Engineering*, J. de Lara and A. Zisman, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 301–315.

[12] A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt, "Automatic classication of large changes into maintenance categories," in *2009 IEEE 17th International Conference on Program Comprehension*, May 2009, pp. 30–39.

[13] B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby, "Cost models for future software life cycle processes: Cocomo 2.0," in *Annals of Software Engineering*, 1995, pp. 57–94.

[14] R. Purushothaman and D. E. Perry, "Towards understanding the rhetoric of small changes-extended abstract," in *International Workshop on Mining Software Repositories (MSR 2004), International Conference on Software Engineering*. IET, 2004, pp. 90–94.

[15] O. Arafat and D. Riehle, "The commit size distribution of open source software," in *2009 42nd Hawaii International Conference on System Sciences*. IEEE, 2009, pp. 1–8.

[16] L. P. Hattori and M. Lanza, "On the nature of commits," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops*. IEEE, 2008, pp. 63–71.

[17] A. Kaur and D. Chopra, *GCC-Git Change Classifier for Extraction and Classification of Changes in Software Systems*. Singapore: Springer Singapore, 2018, pp. 259–267.

[18] G. L. Dodaro, "Government efficiency and effectiveness: Opportunities to reduce fragmentation, overlap, and duplication and achieve other financial benefits," GOVERNMENT ACCOUNTABILITY OFFICE WASHINGTON DC, Tech. Rep., 2015.

[19] Q. Redman, "Weapon system design using life cycle costs," *Raytheon Presenatation*, 2008.

[20] J. Koskinen, "Software maintenance fundamentals," *Encyclopedia of Software Engineering, P. Laplante, Ed., Taylor & Francis Group*, 2009.

[21] M. Yan, Y. Fu, X. Zhang, D. Yang, L. Xu, and J. D. Kymer, "Automatically classifying software changes via discriminative topic model: Supporting multi-category and cross-project," *Journal of Systems and Software*, vol. 113, pp. 296–308, 2016.

[22] S. Levin and A. Yehudai, "Boosting automatic commit classification into maintenance activities by utilizing source code changes," in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2017, pp. 97–106.

[23] R. V. Mariano, G. E. dos Santos, M. V. de Almeida, and W. C. Brandão, "Feature changes in source code for commit classification into maintenance activities," in *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*. IEEE, 2019, pp. 515–518.

[24] S. Hönel, M. Ericsson, W. Löwe, and A. Wingkvist, "Importance and aptitude of source code density for commit classification into maintenance activities," in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2019, pp. 109–120.

[25] G. E. dos Santos and E. Figueiredo, "Commit classification using natural language processing: Experiments over labeled datasets," 2020.

[26] L. Ghadhab, I. Jenhani, M. W. Mkaouer, and M. B. Messaoud, "Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model," *Information and Software Technology*, vol. 135, p. 106566, 2021.

[27] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, and D. Poshyvanyk, "On automatically generating commit messages via summarization of source code changes," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014, pp. 275–284.

[28] M. Linares-Vásquez, L. F. Cortés-Coy, J. Aponte, and D. Poshyvanyk, "Changescribe: A tool for automatically generating commit messages," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 709–712.

[29] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 135–146.

[30] F. Hassan, S. Mostafa, E. S. L. Lam, and X. Wang, "Automatic building of java projects in software repositories: A study on feasibility and challenges," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Nov 2017, pp. 38–47.

[31] F. Hassan and X. Wang, "Change-aware build prediction model for stall avoidance in continuous integration," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Nov 2017, pp. 157–162.

[32] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "There and back again: Can you compile that snapshot?" *Journal of Software: Evolution and Process*, vol. 29, no. 4, p. e1838, 2017, e1838 smr.1838. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1838

[33] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: A case study (at google)," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 724–734. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568255

[34] C. Macho, S. McIntosh, and M. Pinzger, "Automatically repairing dependency-related build breakage," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 106–117.

[35] R. Wieringa, *Design science methodology for information systems and software engineering*. Springer, 2014, 10.1007/978-3-662-43839-8.