Software Design Change Artifacts Generation through Software Architectural Change Detection and Categorisation

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
in Partial Fulfillment of the Requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By

Amit Kumar Mondal

Permission to Use

Abstract

Software is solely designed, implemented, tested, and inspected by expert people, unlike other engineering projects where they are mostly implemented by workers (non-experts) after designing by engineers. Researchers and practitioners have linked software bugs, security holes, problematic integration of changes, complex-to-understand codebase, unwarranted mental pressure, and so on in software development and maintenance to inconsistent and complex design and a lack of ways to easily understand what is going on and what to plan in a software system. The unavailability of proper information and insights needed by the development teams to make good decisions makes these challenges worse. Therefore, software design documents and other insightful information extraction are essential to reduce the above mentioned anomalies. Moreover, architectural design artifacts extraction is required to create the developer's profile to be available to the market for many crucial scenarios. To that end, architectural change detection, categorization, and change description generation are crucial because they are the primary artifacts to trace other software artifacts.

However, it is not feasible for humans to analyze all the changes for a single release for detecting change and impact because it is time-consuming, laborious, costly, and inconsistent. In this thesis, we conduct six studies considering the mentioned challenges to automate the architectural change information extraction and document generation that could potentially assist the development and maintenance teams. In particular, (1) we detect architectural changes using lightweight techniques leveraging textual and codebase properties, (2) categorize them considering intelligent perspectives, and (3) generate design change documents by exploiting precise contexts of components' relations and change purposes which were previously unexplored. Our experiment using 4000+ architectural change samples and 200+ design change documents suggests that our proposed approaches are promising in accuracy and scalability to deploy frequently. Our proposed change detection approach can detect up to 100% of the architectural change instances (and is very scalable). On the other hand, our proposed change classifier's F1 score is 70%, which is promising given the challenges. Finally, our proposed system can produce descriptive design change artifacts with 75% significance. Since most of our studies are foundational, our approaches and prepared datasets can be used as baselines for advancing research in design change information extraction and documentation.

Acknowledgements

First of all, I would like to express my heartiest gratitude to my respected supervisors Dr. Chanchal K. Roy, Dr. Kevin A. Schneider and Dr. Banani Roy for their constant guidance, advice, encouragement and extraordinary patience during this thesis work. I would like to give special gratitude to my research collaborator Sristy Sumana Nath for working with me. Without four of them, this work would have been impossible.

I would like to thank Dr. Ramakrishna Gokaraju, Dr. Zadia Codabux, Dr. Manishankar Mondal and Dr. Norihiro Yoshida for their willingness to take part in the advisement and evaluation of my thesis work. I would also like to thank them for their valuable time, useful suggestions and critical insights. Their comments helped improve my thesis significantly.

Thanks to all of the members of the Software Research and ISEA Lab with whom I have had the opportunity to grow as a researcher. In particular, I would like to thank Mohammad Masudur Rahman, Shamima Yeasmin, Kawsar Wazed Nafi, Debasish Chakroborti, Saikat Mondal, Muhammad Asaduzzaman, and other lab members.

I am also thankful to Muhammad Mainul Hossain of TechOptions for his support and guidance during my thesis study.

I am grateful to the Department of Computer Science of the University of Saskatchewan for their generous financial support through scholarships, awards and bursaries that helped me concentrate more deeply on my thesis work.

I thank the anonymous reviewers for their valuable comments and suggestions in improving the papers produced from this thesis.

I would like to thank all of my friends and other staff members of the Department of Computer Science who have helped me to reach at this stage. In particular, I would like to thank Gwen Lancaster, Shakiba Jalal, Findley Shopie, Maurine Powell, and Heather Webb.

I also express my deepest love to my wife Joyshree Mallick and my son Adrij Mondal for their sacrifices. I express my heartiest gratitude to my father Anukul Chandra Mondal and my mother who are the architects of my life. Their endless sacrifice has made me reach at this stage of my life. My parents, and my wife have always inspired me in completing my thesis work.

I dedicate this thesis to my father, Anukul Chandra Mondal, whose inspiration helps me to accomplish every step of my life.

Contents

List of Tables

List of Figures

List of Abbreviations

DEVEM	Development and Maintenance
DIS	Design Impactful Changes
OSS	Open source software
API	Application programming interface/ 3rd party library
DDARTS	Descriptive Design Change Artifcats
SSC	Semantic Structural Change Relations
AST	Abstract Syntax Tree
SDK	Software Development Kit
VCS	Version Control System
CV	Commercially Valuable
DANS	Directory and Naming Structure
DFS	Discriminating Feature Selection
ADL	Architecture Definition Language
ADDL	Architecture Design Definition Language
UML	Unified Modeling Language
TD	Technical Debt
ACDC	Architectural Change Detection and Classification
SACCS	Software Architecture Change Characterization Scheme
JDK	Java Development Kit
M2M	Module to Module
JPMS	Java Platform Module System
TR	Text retrieval
AM	Ambiguous Message
NI	Non-informative
TFIDF	Term frequency Inverse Document Frequency
DPLSA	Discriminating Probabilistic Latent Semantic Analysis
LLDA	Labelled LDA
SemiLDA	Semi-supervised LDA
RF	Random Forest
DNL	Deep Neural Learning
RNN	Recurrent Neural Network
LSTM	Long-short Term Memory

Chapter 1

Introduction

1.1 Motivation

Our daily life is becoming more dependent on various software applications and autonomous systems, starting from transportation to healthcare services [70, 117]. At the same time, people all over the world have noticed that software anomalies are causing problems in various dimensions of their daily life. These include healthcare systems[1], deadly transportation crashes[2], private data leaks, disruption of energy supply[3], denial of social networking services[4], denial of regular socioeconomic activities[5], and so on. However, these anomalies originate from software bugs, software security holes, problematic integration of changes, complex-to-understand, and so on [257, 154, 243, 167, 217, 189, 175]. Besides, regulatory inspections by the government authority of internal software functions are also increasing due to various concerns such as hidden unethical business practices, data privacy and human values (racial bias) [123, 188]. Sometimes, the software industries require extra time and manpower (cost) for regular development and maintenance activities, late-life-cycle change, and continuous integration glitches[6] to cope with the rapidly evolving technologies and requirements [52, 17]. Moreover, software projects are solely designed, implemented, tested, and inspected by expert people, unlike other engineering projects where the project is mostly implemented by the workers (non-experts) after designing by the engineers. Therefore, people involved with software projects have different and constant mental pressure (and cognitive load). In this scenario, analysts, CEOs, and CTOs are also warning of severe problems in retaining the software/IT workforce in upcoming years (85.2M workers shortage by 2030), which will cause severe business and economic damage to many developed nations[7] [8] [9]. They indicated this problem as the result of inverse socioeconomic trends where software development and maintenance complexities (cognitive loads) are increasing, but the psychological interests of people in complex jobs are decreasing[10]. Researchers

[1]https://tinyurl.com/yc5aftpy
[2]https://tinyurl.com/3s4cbc9j
[3]https://tinyurl.com/4ec89ypf
[4]https://tinyurl.com/3b5fkn74
[5]https://tinyurl.com/5n8ate55
[6]https://tinyurl.com/3b5fkn74
[7]https://tinyurl.com/32jcbs9w
[8]https://tinyurl.com/4p8tmtvz
[9]https://tinyurl.com/yd57f79a
[10]https://tinyurl.com/48w6tamh

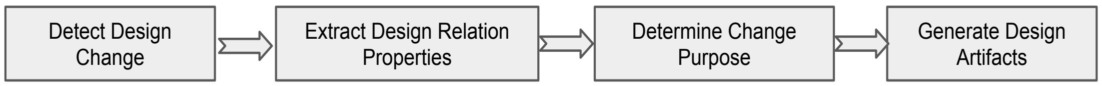


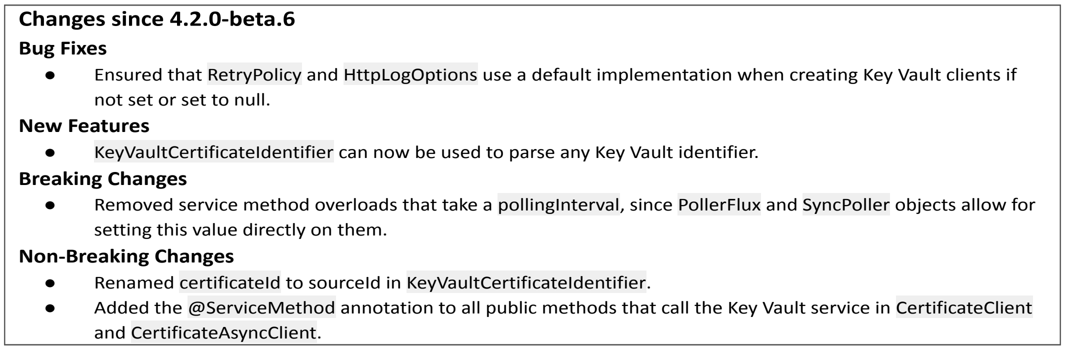
Figure 1.1: Operational phases of design artifacts generation.

have linked all the mentioned problems in software development and maintenance to inconsistent and complex design and a lack of proper ways to easily understand what is going on and what to plan in a software system (code comprehension). This is due to the fact that *there is a significant gap between the information and insights needed by project managers and developers to make good decisions (with little effort) and that which is typically available to them [45]*. Hence comes the necessity of generating design documents through architectural change detection and categorization because it is considered the primary artifact to trace other software artifacts [34].

Design consistency is crucial for both commercial and open-source software (OSS) projects. Because, apart from the development and maintenance challenges, practitioners and researchers are reporting incremental performance and security risks[1] within the architectural component's intra and inter dependency relations [257, 154, 243, 87], elevating the design concerns. In addition to risk reduction, proper design increases developer productivity (i.e., improved maintainability, portability, and flexibility) [52, 209]. Therefore, in order to better reflect the software design challenges, the development teams review the architectural design changes (probably with the motto of *"Prevention is better than cure"*) either on a regular basis or after completing certain milestones or releases [227]. That said, software design concerns are captured through software architecture.

To better cope with the aforementioned challenges, the development and maintenance teams spend significant efforts on code comprehension, code review, and software documentation [13, 59, 256]. Frequent software modifications raise the necessity to comprehend it frequently [50]. However, software comprehension is a human-intensive process, where sufficient knowledge about a software artifact or an entire system is essential to successfully implement a task through analysis, guessing, and formulation of ideas. Inadequate and obsolete software documentation creates bottlenecks in this process and negatively affects the entire team [13]. On the other hand, limited skills or resources and timing constraints compel inadequate documentation and design change information. Additionally, design impactful (or architectural) changes (DIS) are involved in wider spectrum of code components and their dependencies [213, 173], despite focusing on a single issue in such changes. Comprehending their scopes and impacts is more complex for the reviewers [227, 232], elevating the challenges of proper information extraction for documentation. The fact that 40% of the rejected pull requests in OSS projects have design issues indicates that special support is required to understand them

[1]github.com/advisories/GHSA-jfh8-c2jp-5v3q


Changes since 4.2.0-beta.6

Bug Fixes

- Ensured that RetryPolicy and HttpLogOptions use a default implementation when creating Key Vault clients if not set or set to null.

New Features

- KeyVaultCertificateIdentifier can now be used to parse any Key Vault identifier.

Breaking Changes

- Removed service method overloads that take a pollingInterval, since PollerFlux and SyncPoller objects allow for setting this value directly on them.

Non-Breaking Changes

- Renamed certificateId to sourceId in KeyVaultCertificateIdentifier.
- Added the @ServiceMethod annotation to all public methods that call the Key Vault service in CertificateClient and CertificateAsyncClient.

Figure 1.2: Partial release changelogs for DIS of AzureSDK.

through proper sources of information compared to the local code changes [232]. Therefore, this thesis focuses on automatic design change information extraction for that support.

Software architecture can be changed during regular development and maintenance activities – new feature addition, bug fixing, refactoring, and so on. Williams and Carver linked the causes of architectural changes to four Lehman's law of software changes [132] – continuing software change, increasing complexity, continuing growth, and declining quality. A typical software change may contain local code change or architectural change or both. Associating and measuring the scopes and impacts of architectural changes are more complex to track and maintain [227, 232], and elevate the change and maintenance cost and effort across a system's lifecycle [247]. In this regard, architectural change management process helps predict what must be changed, facilitates context for reasoning, specifying, and implementing change, and preserves consistency between system design and changes [189, 175]. In this process, development and maintenance teams categorize the changes based on different criteria, such as the causes of the change, the concept of concerns/features, the location of the change, the size of the code modification, or the potential impact of the change [75, 101, 67]. For example, causes of architectural changes are [246, 63]: *perfective* – indicates new requirements and improved functionality, *corrective* – addresses flaws, *adaptive* – occurs for new environment or for imposing new policies, and *preventative* – indicates restructuring or redesigning the system. Different categories trigger different strategies for change management.

Aparts from frequent code review, code comprehension and change management, a development and maintenance team requires to assess and produce (sometimes frequently or sometimes only on-demand) various design change information to track and plan software releases considering various perspectives – design documents, release notes, descriptive change logs, design decision associativity with the relevant or impacted components, and so on [208, 13, 178]. In this context, empirical findings indicate that more than 85% software project managers and 60% developers are likely to use architecture documents, component dependencies information, change type, and other software documents [45]. Furthermore, 40.5% of the major and 14.5% of the minor software releases contain high-level design change information [33]. These are also valuable entities for the design reviewers and various stakeholders. Proper generation of these entities requires proper categorization of changes [33] such as the categorical information *Bug Fixes, New Features, and Breaking*

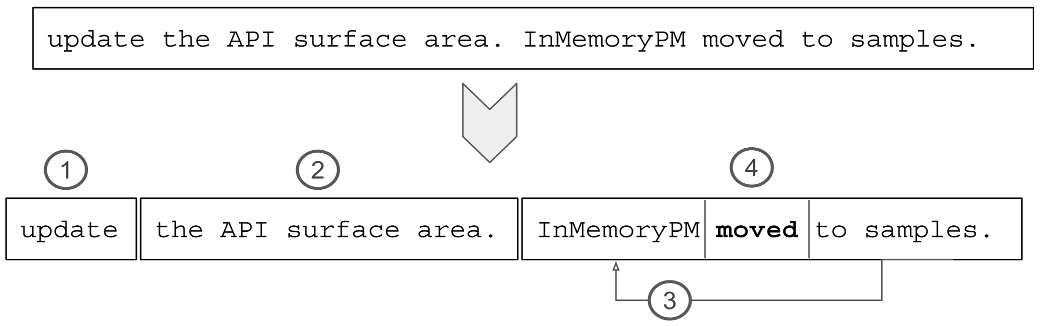


Figure 1.3: Structure of a design change text.

Changes for a release as shown in Fig. 1.2. In addition, a part of the design change information (as shown in Table 1.2) for the feature improvement commit – *"update the API surface area"* is *"InMemoryPM moved to samples"*. This change information indicates architectural components modification in a short descriptive form. Although it is a short form, it contains very crucial design change information. Its information structure is summarized in Fig. 1.3. Here, information in ① and ② indicate the purpose of the change. ② indicates the main theme or parent issue. On the other hand, ③ indicates the design component and relations, and ④ indicates the change operations. From this structure, the reviewers and testers can get quick information about the major components, and they can look into *InMemoryPM* and *Samples* modules first without analyzing all the change code segments. They also get the information that the *InMemoryPM* component moved into the *samples* module without analyzing the code change relations. Thus, a descriptive design change artifact is an elegant source of quick and crucial design change information that contains (at least) - *"why the design change has happened," "how and what high-level program properties are changed and impacted," and "what will be the probable descriptive summary/logs of that change"*. A few of the examples of design change summaries in a real-world project are discussed later in Figures 1.2 and 1.4. All the examples suggest that the basic phases of generating such design change information are – change detection, categorization, and aggregation into natural texts. They are shown in Fig. 1.1. As the steps suggest, it is quite insightful that following these steps for all the change revisions of a release is almost impossible for humans considering the time, costs and benefits [33]. Therefore, generating such documents requires efficient approaches.

There are many implicit implications of architecture change detection and categorization (ACDC) in software evolution support, maintenance support, and fault detection and change propagation [246, 205, 108]. Instead, we discuss three scenarios from real-world cases where architectural change detection and categorization are essential.

Scenario 1: Adequate Information for Change Comprehension and Review: Since code reviewers are not the developers of the implemented tasks, they need to extract accurate information considering various perspectives. However, message description does not contain intended and other information, as shown in some real-world examples in Table 1.1. *Motive in the Description* column represents possible motives of the

Table 1.1: Design impactful changes (DIS) in various projects and their description.

Serial	Description	Motive in description	Change in Code
1.Aion	*update p2p logging & fetch headers based on td*	Either new feature or refactoring or both. Which part is design impactful?	Feature improvement
2.Aion	*some PR changes*	Does it improve feature or refactor? Which issue is linked with this change?	Design restructuring
3.Azure	*Refactored ADLS set access control and added builders for different types*	Two tasks, which one impacted the design	Both of them impacted the design. [**Feature improvement and design refactoring**]
4.Azure	*Storage InputStream and OutputStream*	Looks like new feature	Contents of some classes moved into new classes. [**Improve and simplify design**]
5.webfx	*Improved Action API*	Could be a feature or design improvement	Contents of a class moved into new class. [**Design improvement**]

developers. Reviewers or document writers can only get the real motive by discussing with all the involved developers. Instead, our understanding through code analysis is summarized in *Change in Code* column that appears to be inconsistent with the developer's writing. That said, developers may write a change task inconsistently. However, meaningless or empty messages make the situation worse [148]. Hence, all the information in Table 1.2 is valuable for the reviewers (as well as the description of the SSCs (semantic structural change relations) and modules). Potential information of the design impactful changes may contain this format (at least) - *Actual purpose*, *Reasoning of change*, *Design change relations*, and *Design Change Activity*. An example format is described in Fig. 1.5. Descriptive design changelogs written by the Azure team are shown in Fig. 1.4 (e.g., *A has been flattened to include all properties from C and D classes*). Here, the *Breaking Changes* are related to the preventive change [63]. Furthermore, a detail comment for reasoning the change commit is shown in the *Change Summary Comment* column of Table 1.2. In summary, simplified and useful information by an automated tool is more useful than manually analyzing several hundreds of code segments, methods, classes, and modules (as is required for the 3rd sample in Table 1.1). Once the change purposes and causal relations in the code are extracted, many other design artifacts like these can be generated.

Scenario 2: Planning for next phase/release activities: Category of changes and associated components are helpful for planning the next phase of development. For instance, components and features that have gone through design restructuring in recent times may not be considered for design improvement in the next phase. In the current phase, components with the new feature can be selected for design simplification in the next phase. Because, choosing the sub-optimal solution in the initial stages is common in practice. Moreover, categorical change information is required for decision-making on backporting the changes [54]. However, a recent study found that refactoring related changes are less delayed than flaw fixing and feature improvement. Flaw fixing and improvement have higher frequencies (i.e., priority) compared to other changes for backporting

Table 1.2: Detail comment of a DIS. Underline tokens are the most important info to the developers.

Commit Message	Change Summary Comment (Partial)	SSCs
Updates to event processor surface area for preview 5 (#6107)	The intent of this change is to update the API surface area with the following changes: - Handlers for partition processor methods (event, error, init, close) - InMemoryPartitionManager moved to samples - Event processor takes all params required to build the client ... - New types for each event type (PartitionEvent, Exception-Context... - PartitionManager renamed to EventProcessorStore - Added fully qualified namespace to store interface and a getter in EventHubAsyncClient	7

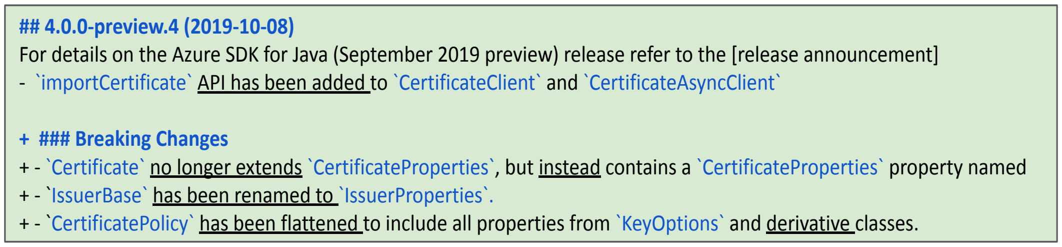
by the development and maintenance team. Thus, change type determination within a completed phase is crucial here.

Scenario 3: Design document generation: A development and maintenance team wants to generate a design document for the upcoming release. Parts of the document require extracting the main features and its associated components. However, without efficient technique, the last two in Table 1.1 could be falsely processed as feature improvement using description, but originally it was for design simplification. Fig. 1.4 shows a partial release log for the developers of AzureSDK *4.0.0-preview.4* in the log change[1] for corresponding feature level description of the release note[2]. Which includes the design-related changes. Such release changelogs are different from usual release notes. A good example that contains change group information can be found in Azure SDK release logs[3], which is shown in Fig. 8.2. Many changelogs, as we have explored ∼200 instances, contain even more information (Fig. 1.4). Writing such logs requires more effort and analysis of various information (i.e., all commits, codebase, commit message, issue reports, and review reports) than writing the commit messages. As many as 17 people (mostly the core architects of a project) are involved in producing logs for a single release [33]. Consequently, tool support for descriptive design changelogs (a subset of design change artifacts) is more critical than for commit message generation.

1.2 Problem Statement

Architectural design change information extraction and documentation are important for software comprehension and code review. To that end, architectural change revisions need to be detected and categorized. Lightweight techniques for high-level design change detection in terms of human intervention, the number of revisions and frequency of usage, and categorizing them properly is challenging [174]. A number of studies

[1]https://tinyurl.com/4xvh2s8j
[2]azure.github.io/azure-sdk/releases/2019-10-11/java.html
[3]https://tinyurl.com/yc2fp3rx


```
## 4.0.0-preview.4 (2019-10-08)
For details on the Azure SDK for Java (September 2019 preview) release refer to the [release announcement]
- `importCertificate` API has been added to `CertificateClient` and `CertificateAsyncClient`

+ ### Breaking Changes
+ - `Certificate` no longer extends `CertificateProperties`, but instead contains a `CertificateProperties` property named
+ - `IssuerBase` has been renamed to `IssuerProperties`.
+ - `CertificatePolicy` has been flattened to include all properties from `KeyOptions` and derivative classes.
```

Figure 1.4: Descriptive changelogs of DIS of AzureSDK.

are available for architectural change detection, but they are not feasible to deploy them for many revisions [24, 245, 253, 65, 29, 69, 12]. Many studies are available for categorizing and generating typical commit messages [101, 95, 160, 86, 255, 135, 104]. However, categorizing and summarizing the design impactful change commits are not specifically focused on. According to our systematic literature review [174], architectural change detection, categorization, and summarization suffer five major limitations as follows:

(a) Experimental Datasets Unavailability: The proposal of automated techniques in recovering, mining, and analyzing architecture information to generate design change artifacts depends on the experimental datasets' quality and quantity. A lack of sufficient datasets affects the deep insights extraction and performance of the architectural change information analysis [34]. Unfortunately, annotated datasets for high-level architectural change revisions are unavailable for research. It requires substantial experts human efforts to annotate a good quantity of samples. Moreover, to ensure the quality of the annotation, more than one people's involvement in the dataset analysis is required. This thesis contributes to this area significantly.

(b) Sole Reliance on Coding Properties for Architectural Change Detection: Most of the existing studies consider either conversion of byte code or other formats of architectural definition coding as the primary step of architecture change detection [24, 245, 253, 65, 29, 69, 12]. As a result, they are either heavyweight or require substantial human intervention. Therefore, frequent usage in real-world scenarios and large-scale empirical studies for architectural change analysis is impaired because a project typically contains hundreds and thousands of change revisions that continuously increase [150]. However, 88.9% of architecture information is written by natural language texts whose size does not vary significantly from project to project [64, 115]. But, no study focused on textual properties for developing lightweight techniques. This thesis reduces that gap to a greater extent.

(c) Directory and Naming Structure with String Patterns Overlooked for Change Detection: Existing change detection studies overlooked directory, naming structure properties, and string patterns in the code change information returned by the *diff* tool of each codebase revision. As a result, existing studies that focus on implemented architecture process all the source code of each change revision, and the execution time is several hours for each revision for extracting the high-level design change [130, 81, 129, 151, 150]. In this thesis, we study the mentioned properties to detect architectural change revisions and extract component change relations.

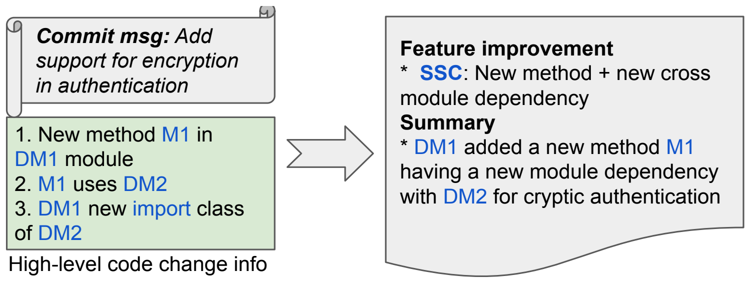

Commit msg: Add support for encryption in authentication
1. New method M1 in DM1 module
2. M1 uses DM2
3. DM1 new import class of DM2
High-level code change info
Feature improvement
* SSC: New method + new cross module dependency
Summary
* DM1 added a new method M1 having a new module dependency with DM2 for cryptic authentication

Figure 1.5: Example summary from a DIS commit.

(d) Semantic Tokens (theme) and Change Relations Overlooked for Change Categorization: Most of the existing studies are typical change commits categorizations that consider traditional text properties and source code properties [101, 95, 160, 86, 255, 135, 104]. As a result, adapting them for architectural context can not handle the commit triad (tangled, ambiguous, and non-informative or empty message) challenges properly. However, they consider neither more intuitive semantic tokens/themes nor semantic code change relations properties. Such properties need to be explored for commit triad handling in proper architectural change categorization. This thesis extracts semantic tokens and proposes novel approaches for architectural change classification leveraging them.

(e) Design Change Summaries are not Specifically Focused: Some excellent studies exist that automatically generate commit summary descriptions and release notes [44, 144, 112, 239, 148, 178, 120, 33]. However, change summaries and change logs for high-level design changes are not covered by them due to cost, efforts and substantial experts intervention. Thus, they are not helpful in getting high-level design change information and impacts. Our study fills this gap to a certain extent.

1.3 Our Contribution

Software design inconsistency and complexity are directly linked to software bugs, cyber-attacks, problematic integration, and increased cognitive pressure on the developers, which have severe consequences. To reduce design inconsistency and to make the system easy to understand, changes need to be tracked, reviewed and documented regularly. Therefore, architectural change detection, categorization, and documentation are extremely important for consistent and less complex system design. For these tasks, manual and heavyweight automated techniques are employed, which are costly and can not be deployed frequently. In this thesis, we tackle the mentioned challenges described in the previous *Section 1.2* and significantly advance the research of design change tracing and document generation. In particular, we conducted six different studies (Table 1.3) where the first and fourth studies focus on lightweight architectural change detection, the second, third, and fifth studies handle the architectural change categorization, and the sixth study focuses on automated design change document generation. Finally, we combine many of these studies to develop a novel tool, namely DDARTS (Descriptive Design Change Artifacts). This tool helps the development and maintenance team to

Table 1.3: Thesis Contribution Overview.

Approach	**Task**	**Input**	**Output**	**Novel Contribution**
S1:ACCOTERM	Change detection	Commit message, developer's discussion	Architectural Changes	Co-occurred terms for archi change
S2: ArchDFM	Change classify	Commit message	Four change types	Four set of keywords for four change types
S3: ArchiNet	Change classify	Commit message	Four change types	Concept-tokens and their strength calculation
S4: ArchSlice	Change detection	VCS codebase revisions	Architectural Changes, semantic change relations	Identify challenges of directory and naming structures of codes
S5: ArchSSC	Change classify	Commit message, codebase revision	Four change types	Several models for handling message triad
S6: DDARTS	Design documentation	Commit message, codebase revisions	Design Change Summaries	Precise change context extraction and static rules formation

get the architectural change information frequently. The short introduction of these studies is as follows.

1.3.1 Architectural Change Detection

(a) Architectural Change Detection from Textual Descriptions

First, we explore textual descriptions such as change revision (commit) messages and developer discussions for a task for architectural change detection. Because software codebase size can vary from project to project, the detection complexities also increase proportionately. In contrast, the range of the textual description for per change revision is almost constant. However, we first followed the study of Ding et al. [63] where we investigated the source code and commit messages of three projects (Galaxy, iPlant, and ImageJ) and mailing threads of two projects (Hibernate and ArgoUML). After that, we reviewed a number of studies related to architecture documentation and change scheme descriptions [246, 115, 11, 194]. Then we compiled four steps: (i) keywords extraction, (ii) keywords refinement, (iii) co-occurred terms extraction, and (iv) detection model development. Through the first three steps, we manually extracted around 130 co-occurred terms that may represent architectural change related activities. Finally, deploying the TF-IDF and term graph, we proposed an architectural change commits detection technique with those co-occurred terms. This study contributes to the research challenges (a) and (b) in *Section 1.2*.

(b) ArchSlice: Architectural Change Detection and Semantic Change Relations Extraction from Source Code

In the previous study, we found that architectural change detection leveraging the textual properties is promising, but it has limitations. Directory and naming structure patterns and *diff* tool change information can complement the challenges. During the manual analysis of the changed code of 3,647 commits from 10

OSS projects, we noticed that DANS (directory and naming structures) properties play a crucial role in determining the high-level architectural or design impactful change instances. In this process, we observe various DANS properties and their variations across the change revision commits of the projects. The list of these properties (and types) is later discussed in Chapter 6. However, implemented architectural change instance can be detected in two ways: (i) comparing ASTs from byte code of two consecutive commits and (ii) processing the provided information by the commit *diff* tool of the VCS [88, 5]. In the first technique, a complete codebase for each committed version needs to be compiled into byte code, which mostly requires manual intervention to resolve 3rd party library dependencies. Moreover, the intensive computation could be a bottleneck for a normal purpose machine for analyzing changes at the statement levels for large systems. Usually, each release of a project may contain hundreds of commits. Thus, AST-based techniques might far exceed the time and efforts (of manual analyses) for architectural change detection. Therefore, we propose to detect high-level architectural change revisions by processing the DANS properties, codebase, and *diff* tool information using the *String* processing techniques. This study contributes to the research challenges (a) and (c) in *Section 1.2.*

1.3.2 Architectural Change Categorization

(c) Architectural Change Categorization using Discriminating Feature Model

We observed that the developers write change category information in almost all the release logs. Therefore, for descriptive summary generation, it is mandatory to categorize the architectural changes. As soon as the architectural change revision is detected, it is also required to determine the cause or purpose of the change to better represent the change knowledge and generate various software documents. To that end, we first explore the existing techniques for software change classification. In this study, we extracted four discriminating sets of keywords for four types of architectural changes (perfective, corrective, preventive and correct) from experimental training data using DPLSA [255]. Then these keywords sets are used for generating prediction models using LLDA [149], SemiLDA [80], and DPLSA [255]. These models are referred to as discriminating feature selection models (DFS). However, this is the first study that specifically focused on architectural change categorization. This study contributes to the research challenges (a) and (d) in *Section 1.2.*

(d) ArchiNet: A Concept-tokens based approach for Architectural Change Categorization

In the previous study, we observe that many keywords are overlapped among the change categories. Additionally, tacit variation of intention cannot be captured with the discriminating sets of keywords. To eradicate this limitation, we explore intuitive text properties. In this study, we define and extract concepts [200] from the commit messages of all the annotated samples that express the corresponding intention of a task. Even the top words (such as support) among the defined concepts contain many overlapping words. However, we have found some patterns in many samples for expressing different concepts when these terms co-occurred with other tokens, which are stop words, code elements, and API/library/framework name. To handle this issue, we train a model by assigning weights to the concept tokens using a specialized normalized frequency model from a set of pre-classified commits into four change categories. This is motivated by the core idea of how the model for a word's sentiment is generated [21]. These weights represent the strengths of the presence of the tokens within the concepts of the categories. Finally, the trained model produces a collection of unique concept tokens, which are then used to predict the change message to an expected category. This study contributes to the research challenges (a) and (d) in *Section 1.2*.

(e) Handling the challenges of Architectural Change Categorization leveraging Structural Change Properties of Source Code

Our previous study is promising for change categorization. However, many commit messages do not contain concept tokens, and those messages cannot be measured with concept tokens. Therefore, we further enhance the previous techniques with code properties for handling the message triad (empty, meaningless, and tangled) for architectural change categorization reliably. Many commit descriptions do not contain previously explored text properties. But, meaningful source code properties can complement the gap of this issue. The relationships between code and their architecture knowledge are hard to reveal [34]. The issue is a lack of overview of the structure of the system, linking to the source code and program file. To improve this scenario in message triad handling, through manual analysis of the source code of the high-level architectural change samples, we extract 17 semantic change relations (SSC) from code operations. We have explored various classification models with these SSCs. Then, we explored various classification models combining these SSCs with concept tokens. Finally, we have proposed approaches to handle the challenges of commit triad – tangled changes, ambiguous messages, and non-informative or empty message descriptions. Thus, our proposed models are promising to apply in real-world applications such as release change logs or release note generation. This study contributes to the research challenges (a), (c) and (d) in *Section 1.2*.

1.3.3 Design Change Document Generation

(f) DDARTS: An Automated Approach for Design Change Artifacts Generation

In the aforementioned studies, we have explored several approaches for architectural change detection and categorization that can be utilized for efficient design change information generation. In this study, we explore an automated technique to generate architectural change summaries (textual) leveraging the most promising change detection and classification techniques we have proposed in previous studies. Context-aware descriptive design change summary (a subset of design change artifacts) generation is a challenging task. To that end, we consider precise and meaningful contexts based on the SSCs, change purposes, and relevant concepts related to them for generating the descriptive design change summaries. In this process, we first generate SSCs and concept tokens mapping models from the training dataset created in the previous study. This mapping model contains separate models for each change group (with weighted ranked words). Then, during change description and release change logs generation, we determine all the possible change types using the uniform distribution models proposed in our fifth study. After that, the top-ranked concept tokens of those SSC mapping models from the relevant categories are included in the number of unique sets (according to the predicted types). The messages can be generated considering the first 1-3 top-weighted concept tokens. We also integrate commit theme information using various contextual rules. Moreover, we have defined four static rules for generating four types of change descriptions. Finally, we developed a tool for the development and maintenance team for design change information extraction and change log documentation. This study contributes to the research challenges (a) and (e) in *Section 1.2*.

1.4 Related Publications

- **Amit Kumar Mondal**, Chanchal Roy, Banani Roy, and Kevin A. Schneider. *"Automatic Components Separation of Obfuscated Android Applications: An Empirical Study of Design Based Features"*, In 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASE 2019), pp. 23-28.

- **Amit Kumar Mondal**, Banani Roy, and Kevin A. Schneider. *"An exploratory study on automatic architectural change analysis using natural language processing techniques"*, In 2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM 2019), pp. 62-73.

- Debasish Chakroborti, Banani Roy, **Amit Kumar Mondal**, Golam Mostaeen, Chanchal K. Roy, Kevin A. Schneider, and Ralph Deters. *"A Data Management Scheme for Micro-Level Modular Computation-Intensive Programs in Big Data Platforms"*, In Data Management and Analysis, Springer, Cham, pp. 135-153.

- **Amit Kumar Mondal**, Banani Roy, Sristy Sumana Nath, and Kevin A. Schneider. *"ArchiNet: A Concept-token based Approach for Determining Architectural Change Categories"*. 33rd International Conference on Software Engineering and Knowledge Engineering (SEKE 2021), pp. 7-14.

- **Amit Kumar Mondal**, Chanchal K. Roy, Kevin A. Schneider, Banani Roy, and Sristy Sumana Nath. *"Semantic Slicing of Architectural Change Commits: Towards Semantic Design Review"*, In Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2021), pp. 1-6.

- **Amit Kumar Mondal**, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. *"Large Scale Image Registration Utilizing Data-Tunneling in the MapReduce Cluster"*, In Proceedings of the International Conference on Big Data, IoT, and Machine Learning (BIM 2021), pp. 167.

- **Amit Kumar Mondal**, Kevin A. Schneider, Banani Roy, and Chanchal K. Roy. *"A survey of software architectural change detection and categorization techniques"*, Journal of System and Software (JSS), 194 pp.
- **Amit Kumar Mondal**, Banani Roy, Sristy Sumana Nath, Chanchal K. Roy, and Kevin A. Schneider. *"DDARTS: An Automated Tool for Descriptive Artifacts Generation of Design Changes"*, ACM Transactions on Software Engineering and Methodology (**TOSEM 2023**, **Under Submission process**).
- **Amit Kumar Mondal**, Kevin A. Schneider, Banani Roy, and Chanchal K. Roy. *"A survey of software architectural change detection and categorization techniques"*, 20th IEEE International Conference on Software Architecture (ICSA 2023 Journal First Track, accepted).
- **Amit Kumar Mondal**, M. Mainul Hossain, Chanchal K. Roy, Banani Roy and Kevin A. Schneider. *"FSECAM: A Semantic Theme-based Approach for Feature to Muilti-level Architectural Component Mapping"*, Journal of System and Software (**JSS 2023, Under Submission process)**.

1.5 The Organization of the Dissertation

The thesis contains nine chapters in total. To address the challenges of architectural change detection, categorization, and summarization, we conduct six independent but interrelated studies. The core of this dissertation starts with a description and discussion of software architecture and changes.

- In **Chapter 2**, I discuss the basics of software architecture, architecture modification, change categories, design change artifacts generation steps, and implications of change detection and categorization.
- In **Chapter 3**, I describe the first study (ACCOTERM) for software architectural change detection from commit messages and developer's discussion using text retrieval technique.
- In **Chapter 4**, I describe the second study that categorize the architectural changes using discriminating feature model and text classification techniques.
- In **Chapter 5**, I describe the study of architectural change categorization using concept tokens and machine learning techniques.
- In **Chapter 6**, I describe the challenges of developing a lightweight technique for detecting architectural change revisions from source code properties. Then, I discuss the definition and detection of architectural semantic change relations from given change information by the *diff* tool leveraging directory and naming structure and string patterns.
- In **Chapter 7**, I describe the architectural change categorization using SSCs and various classification techniques to handle message triad challenges.

- In **Chapter 8**, I describe the descriptive design change summary generation by forming static rules embedding concept tokens, SSCs and change classification model.
- In **Chapter 9**, I conclude my thesis with future research directions.

Chapter 2

Literature Background

2.1 Software Architecture

Usually, software architecture is considered the structures of a software system and the process (associated with different development activities) [229, 84]. It documents the shared understanding of a system design [78]. This understanding involves how the system is partitioned into components and how the components interact through interfaces. According to Grady Booch[1], *a software architecture represents the significant design decisions that shape a system, where significance is measured by the cost of change*. Frequent change analysis prevents the drifts of software architecture from the original design decisions. However, various other forms of definition for software architecture are available in the literature [229]. According to Richards and Ford [207], *like much art software architecture can only be understood in context*. A list of definitions by the practitioners is available at Carnegie Mellon University archive[2]. Instead, we summarize some of the selected definitions of software architecture which are shown in Table 2.1. The table also indicates the key points of the definitions. From this summary, we can observe that architecture is about structural construction [56], design decisions implementation [229], evolution (IEEE-1471) and knowledge sharing [258] mechanisms of a system. Furthermore, these definitions introduce the importance of reviewing, tracking, and documenting architectural changes due to the substantial impacts of various dimensions of software development and maintenance.

Table 2.1: Definition of Software Architecture

Author	**Key point**	**Definition**
Andreas Zwinkau [258], 2019	Knowledge sharing	*Software architecture documents the shared understanding of a software system*

[1] https://tinyurl.com/2bu5fvfu
[2] https://tinyurl.com/bddft4y6

Table 2.1: Definition of Software Architecture

Author	**Key point**	**Definition**
Bodje and Nasira [184], 2013	Design to optimize	*Software Architecture helps to shape the design, which is used to communicate and collaborate on the implementation of the functional and non-functional requirements when producing an application while optimizing common quality attributes.*
Clements et al. [56], 2010	Structure	*The software architecture of a system is the set of structure needed to reason about the system, which comprises software elements, relations among them, and properties of both.*
Eoin Woods [252], 2016	Design decisions	*Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled.*
Taylor et al. [229], 2009	Design decisions	*A software system's architecture is the set of principal design decisions made about the system.*
Grady Booch [258], 2006	Design and change impact	*All architecture is design but not all design is architecture. Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change.*
Jansen and Bosch [109], 2005	Design decisions	*We do not view a software architecture as a set of components and connectors, but rather as the composition of a set of architectural design decisions.*
Jan Bosch [39]	Design decisions	*Software architecture is, fundamentally, a composition of architectural design decisions. These design decisions should be represented as first-class entities in the software architecture and it should, at least before system deployment, be possible to add, remove and change architectural design decisions against limited effort.*
Rozanski and Woods [211], 2005	Structure	*The architecture of a software-intensive system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.*
Bass et al. [27], 2003	Structure	*The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.*
Kruchten [126], 2003	Structure and collaboration	*An architecture is the set of significant decisions about the organization of a software system, the selection of structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these elements into progressively larger subsystems, and the architectural style that guides this organization – these elements and their interfaces, their collaborations, and their composition.*

Table 2.1: Definition of Software Architecture

Author	**Key point**	**Definition**
Kari Smolander [218], 2002	Knowledge metaphors	*Four general metaphors for architecture, "architecture as blueprint", "architecture as literature", "architecture as language", and "architecture as decision"*
IEEE-1471, 2000	Structure for evolution	*Architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.*
SEI, 1994	Structure for evolution	*The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.*
Garlan [85], 1993	Structure	*Software architecture is the organization of a software system as a collection of components, connections between the components, and constraints on how the components interact.*

2.1.1 Documenting Software Architecture

According to literature [64] and IEEE 1471-2000 standard [11], a software architecture is mostly presented in the following ways,

Natural Language

Information for documenting architectural elements can be expressed with natural language, which may use specific technical or domain terms to interpret various aspects of the architectural elements. For example, a brief architectural document about Hadoop and HDFS [1] is available in a natural language format.

Diagrams

In many cases, architectural information is presented with informal graphical diagrams (boxology). It can provide an abstract snapshot of the system.

UML (Unified modeling language)

UML model is a notation-based standard language for documenting architecture description. UML mainly captures the structural aspects of a software.

[1]https://tinyurl.com/3sshtrch

ADL (Architecture description language)

ADL provide formally-specified [64] modeling notations and constraints to describe an architecture with the help of set of tools. An ADL provides a high level of abstraction that can be read by both human and machine. It can support automatic generation of parts of software systems [26].

Among the ways mentioned above, natural language is the most significant [64, 115] medium to represent a system architecture (at least for open source projects). Researchers [64] observe that among 108 open source projects, 88.9% describe the system architecture using natural language. A software architecture document is viewed [229] as (i) prescriptive architecture, and (ii) descriptive architecture. Prescriptive architecture documents what is intended prior to a system being built. It can be documented with the unified modelling language (UML), the architecture design definition language (ADDL), graphs, or natural language. In contrast, descriptive architecture captures/ documents what has already been implemented. Note that UML, ADDL, natural language, and graphs can be used to document either prescriptive or descriptive architecture. A descriptive architecture is also documented with implementation level entities such as packages.

2.1.2 Architecture Views and Abstraction Levels

A software architecture is viewed [229] as (i) prescriptive architecture, and (ii) descriptive architecture. Prescriptive architecture documents what is intended prior to a system being built; can be documented with the unified modelling language (UML), the architecture design definition language (ADDL), graphs, or natural language. In contrast, descriptive architecture captures/ documents what has already been implemented. A key point is prescriptive is prior to a system being built and descriptive is after the fact. Note that UML, ADDL, natural language, and graphs can be used to document either prescriptive or descriptive architecture. A descriptive architecture is also documented with implementation level entities such as packages. Descriptive architecture is studied at three abstract levels: high level, intermediate level, and low level. Implementation level entities such as modules or libraries are considered as a high-level abstraction. Packages, classes, and program files are regarded as an intermediate-level abstraction. Methods, functions, and procedures are a low-level abstraction. Low-level abstraction is frequently studied for understanding procedural and legacy systems. Detecting higher-level abstractions from an implementation is challenging. A change analysis process can utilize software artefacts from various sources [36, 64, 195]: design and requirement documents, the codebase, issue trackers, review comments, commit messages, developer discussions and messaging lists.

2.1.3 Primary Elements of Architecture

Literature has discussed architectural aspects from various perspectives. An architectural perspective is a nonempty set of types of architectural design decisions. Principle architectural elements for describing these design decisions are components, connectors, configuration, and constraints.

Components:

A component is an implementation unit of software that provides a coherent functionality unit [56] as a black box. For forming a proper architecture, usually, a component should be modular, portable, replaceable, and reusable set of well-defined functionality [189]. Thus a component reflects the principles of encapsulation, abstraction, and modularity. From the perspective of architecture, it could be a module, a package, a class, a code file, or even a cluster of methods.

Module:

A module is a software unit where low-level code components are connected and aggregated for a function/-common set of functions. Code components within a module are used or connected to other modules. These modules are building blocks where the interfaces and interchangeability are important for the possibility of combining them to create modular products that could fulfill various overall functions. Modularity is an early defined quality property of software architecture [25, 46] . It is considered the high-level abstraction unit of architecture. However, literature has referred the modules in different ways. In this thesis, we refer to a module as the cluster of packages or classes or code files (mostly aligned to Java and Kotlin programming) [35]. Furthermore, a 3rd party library is considered as a module. However, a cluster of code components in general, also called a module [150].

Connectors:

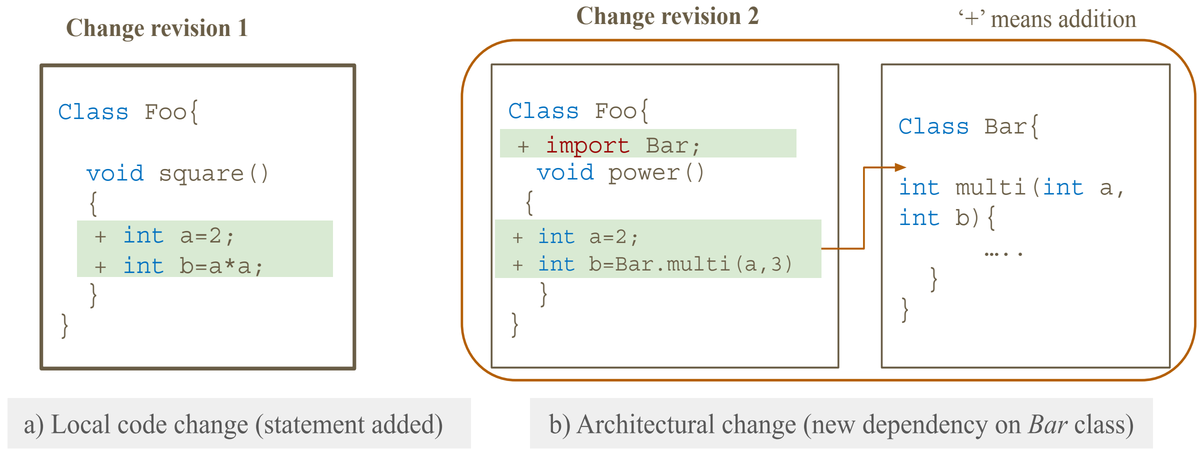
Connectors are explicit architectural elements that bind components together and act as mediators between them [84]. It ensures interaction protocol and a communication mechanism between components independent of their functional behaviours. However, dependency is analogous to a connector. That said, a connector is used for describing runtime architecture, whereas dependency is used as an alternative for static architecture. Procedure calls, database access protocols, and client-server protocols are some of the examples of connectors.

Configuration:

In software architecture, a configuration is used to specify connectors, dependencies and access restrictions among components and modules. A file within a software describing the dependency relations among the modules is an example of architecture configuration [116]. It may be represented as a graph wherein nodes represent components and connectors, and edges represent their associations (topology or procedure call).

Constraints:

Constraints or rationale in software architecture documentation express restriction or permission, generality or particularity, necessity and luxury, and relativeness and absoluteness [198]. They are properties of, and the


Change revision 1
Change revision 2
'+' means addition
a) Local code change (statement added)
b) Architectural change (new dependency on *Bar* class)

```
Class Foo{

  void square()
  {
   + int a=2;
   + int b=a*a;
  }
}
```

```
Class Foo{
 + import Bar;
   void power()
  {
 + int a=2;
 + int b=Bar.multi(a,3)
   }
}
```

```
Class Bar{

int multi(int a,
int b){
      …..
   }
}
```

Figure 2.1: Class level architectural change.

relationships among, the architectural elements. However, constraints could be of any type, such as technical, managerial, economic, or political constraints.

2.2 Software Architecture Modification

Parts/elements of software architecture can be changed during the regular development and maintenance activities. Two different abstraction level architectural changes are summarized in Figures 2.1 and 2.2. In Fig. 2.1, the first change (a) is the local change, while the second change (b) is a class-level architectural change because it added a new dependency from another component. In Fig. 2.2, in later version (2) of the architecture, a new component called *"Controller"* is added and dependency relations changed for implementing a new functionality for the admin of the *"Managed Server"*. In these examples, we can notice their impacts and increased complexities of reviewing them for flaws and security vulnerabilities compared to local code change. In the next subsections, I will discuss more about architectural changes.

2.2.1 Why Do Changes in Architecture Happen?

In this section, I briefly discuss various perspectives of architectural change. A few decades ago, change or evolution of E-type[1] software was captured by eight laws [134] by Lehman and Ramil. These laws implicitly and explicitly cover almost all aspects of software changes and concerns related to them. Four of them can be related to software architectural change and evolution [246]. These laws are valuable indications of why software architecture changes.

- **Law I, Continuing Change:** *A software that is used must be continuously adapted else becomes pro-*

[1]Software that mechanize a human or societal activity [133] such as operating systems, business administration software, inventory management, stock trading, etc.

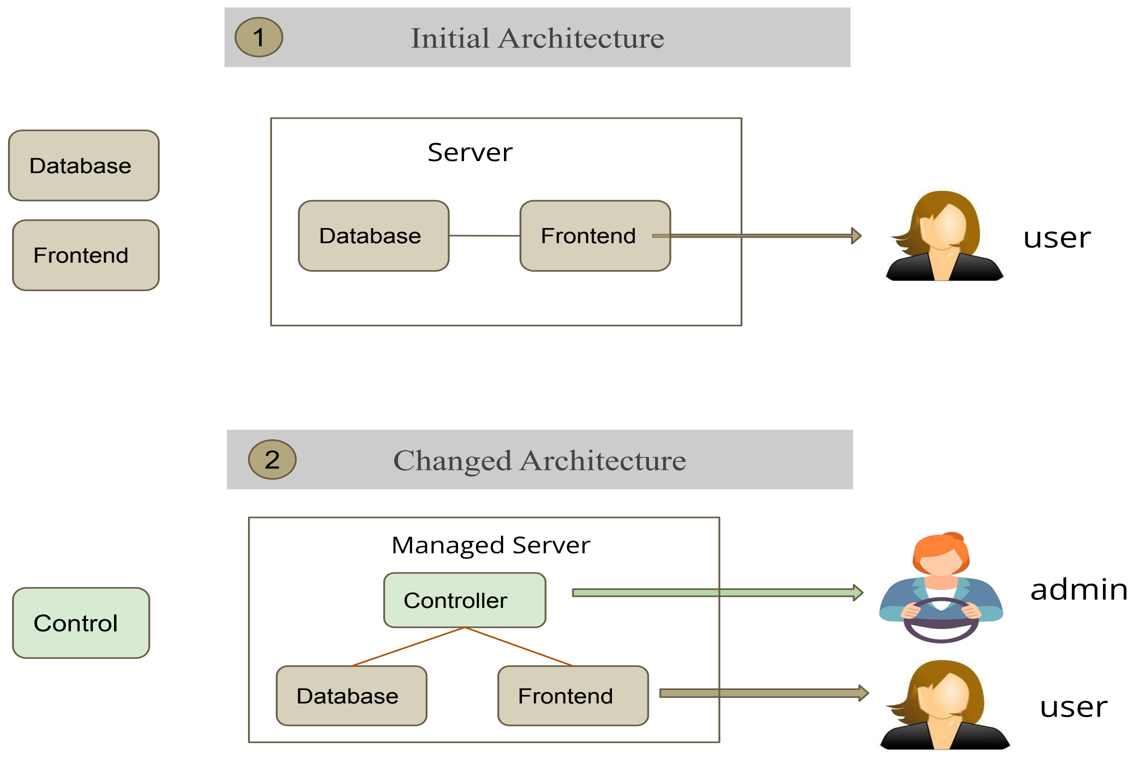


Figure 2.2: High-level architectural change.

gressively less satisfactory. Continuing adaptation and evolution is because of the fact that development, installation and operation of the software change the application and its operational domain within an organization. This creates a mismatch between the two. Evolution happens both from the feedback and control maintenance process.

- **Law II, Increasing Complexity:** *As a software is evolved its complexity is increased unless work is done to maintain or reduce it.* From the necessity for adaptation, changes are implemented. That may cause an increment of interactions and dependencies between the system components in an unstructured way, ultimately increases system complexity. Change happens to reduce the structural complexity.
- **Law VI, Continuing Growth:** *Functional content of program must be continually increased in its life time to meet the customer satisfaction.* This law is associated with change deriving from a different source. During the development and upgradation, many functional, behavioural and other attributes are excluded due to budget, delivery dates, technology and understanding of the application in its domain. These items cause bottlenecks when the user has to replace automated operations with human intervention. Hence they also lead to demand for change. As such, the growth of the software codebase may happen due to the requested functionalities perceived after real-world usage and the integrated applications into the system for removing bottlenecks and flaws.
- **Law VII, Declining Quality:** *A software product is perceived as a declining of quality unless rigorously maintained and adapted to a change in an operational environment.* The seventh law states

that uncertainty related to quality increases with time unless the maintenance activity detects and rectifies the integration. This law indicates the fact that *familiarity breeds contempt*. For example, customers' perceptions and expectations may change with time and availability of alternative products.

2.2.2 Changes in Static Architecture

Software systems have a static and a dynamic architecture [90]. The static architecture is represented by a collection of modules that are structured in a module hierarchy during design or implementation time. On the other hand, the dynamic architecture represents their configurations and actions during the execution of a system at runtime [90]. That said, changes in static and runtime architecture occur in different ways. Thus, different strategies are required for managing different changes. Updating operating systems and adding new features in a running web application are examples of runtime architecture change. Static changes also cause runtime changes. In this section, we discuss how static (or design-time) architectural change operations are defined in the literature. There are many ways that architecture views and properties can be modified. Among them, some common ways in which architecture may be modified after it has been built are [192]:

Common operations: Common change operations of software architecture are the addition of new components, upgrading existing components (e.g., performance tuning), removal of unnecessary components (both temporary and permanent). The change operations are also referred to as [246]: kidnapping, splitting and relocating. Kidnapping is moving an entire module from one subsystem to another. Whereas, splitting involves dividing a module into two distinct modules. Relocating involves moving functionality from one module to another.

Configuration change: Software configuration is also a property of software architecture [72]. Configuration may control dependency, access rules and restrictions of both design-time and run-time components [87]. Therefore, change operations include reconfiguration of application architecture (reconnection of components and connectors) and reconfiguration of system architecture (e.g., modifying the mapping of components to processors). Moreover, modifying values of many configuration characteristics is also considered an architectural change operation.

Source-code layers: Source-code layers are directories, package structures, and location of code files within the directories [151]. Static changes affect system structure and consist of changes to systems, subsystems, modules, packages, classes, and relationships. Class hierarchy changes consist of modifications to the inheritance view. Class signature changes describe alterations to system interfaces.

Design model change: Changes can be made to UML diagrams, and other informal models (such as component and package diagrams) where each diagram type signifies the nature of the changes [245, 152, 81]. For example, changes in UML diagrams can include the addition or deletion of class attributes, modifying relationships, and the addition or deletion of classes [152]. The architectural pattern can be changed, such as layered architecture pattern can be migrated to model-view-controller pattern. Addition, deletion and migration of design patterns are also architectural changes.

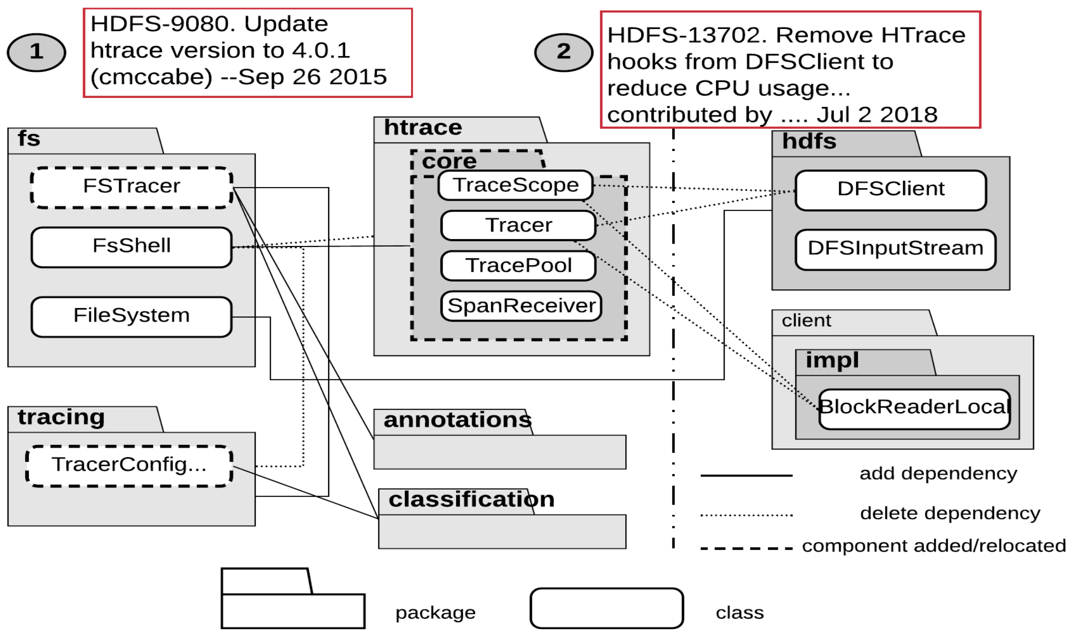


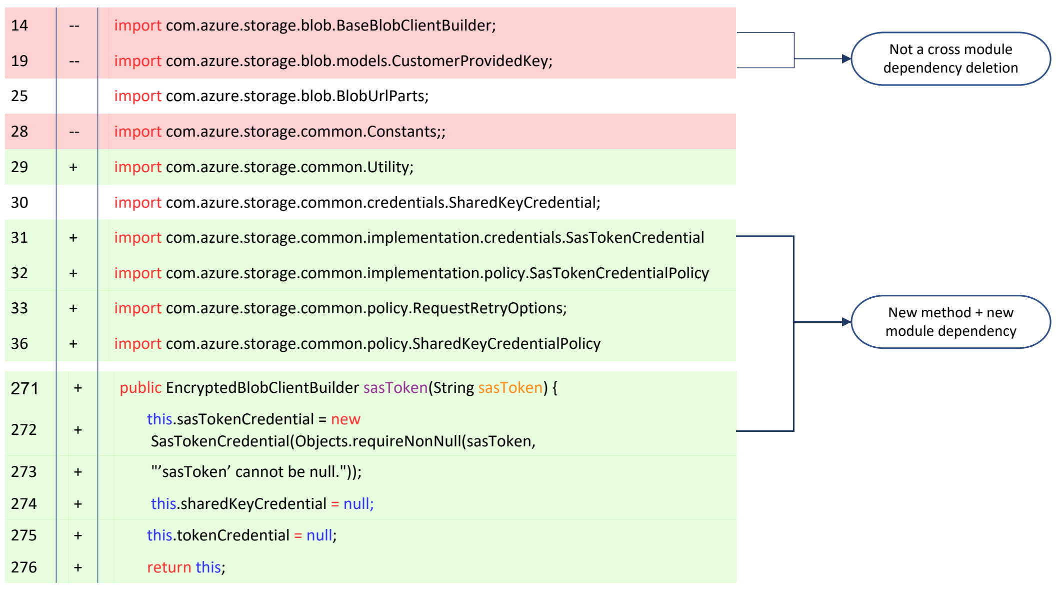
Figure 2.3: Class level architectural changes of Hadoop.

Architectural document change: Since architecture is documented with natural language in many projects, a change in the description of the document also indicates an architectural change [63].

Examples: A dummy change in the class-level architecture is discussed in Fig. 2.3. An example of a real-world module-level change within a commit of Microsoft AzureSDK is shown in Fig. 2.4. Here, the *azure-storage-blob-cryptography (ASBC)* module uses functions from *azure-storage-common (ASC)* module for improving a feature. To extract this information, the developers have to search the location of the code segments, functions, classes and modules in the codebase (perhaps with the IDE). However, the commit contains modifications of 32 classes with many variations. Hence, the manual process to extract the design impacted information requires tremendous effort.

2.2.3 Metrics as Architectural Changes

In this thesis, we explore the common change operations, configurations change, source-code layers, and natural text documents as the metrics for design impactful changes. We study the module/system-level changes and class and package level changes [174]. Literature has defined many metrics to indicate the architectural changes at various abstraction levels. srcTracer [93], ID-SD [151], RC [62], NHK [223], CSM [69], CC [167], CDI [117], DSM [24], MLC [242], and GKD [181] metrics are available for the intermediate level changes. MoJo [230], MoJoFM [245], HEAT [65], CB [40], C2C [81], A2A [130], and $A\Delta$ [110] focused on the high-level change detection (some of them also consider the intermediate-level change). Detail analysis of these metrics are discussed in our published article [174]. Many of these metrics considered add, delete, move, split, update and component renaming. Some leveraged fluctuations in various quantitative values



```
--  import com.azure.storage.blob.BaseBlobClientBuilder;
--  import com.azure.storage.blob.models.CustomerProvidedKey;
    import com.azure.storage.blob.BlobUrlParts;
--  import com.azure.storage.common.Constants;;
+   import com.azure.storage.common.Utility;
    import com.azure.storage.common.credentials.SharedKeyCredential;
+   import com.azure.storage.common.implementation.credentials.SasTokenCredential
+   import com.azure.storage.common.implementation.policy.SasTokenCredentialPolicy
+   import com.azure.storage.common.policy.RequestRetryOptions;
+   import com.azure.storage.common.policy.SharedKeyCredentialPolicy

+   public EncryptedBlobClientBuilder sasToken(String sasToken) {
+       this.sasTokenCredential = new
+        SasTokenCredential(Objects.requireNonNull(sasToken,
+        "'sasToken' cannot be null."));
+        this.sharedKeyCredential = null;
+       this.tokenCredential = null;
+       return this;
```

Not a cross module dependency deletion
New method + new module dependency

Figure 2.4: Code changes containing a module-level change. Red color represents deletion ($-$) and green color represents additions ($+$).

(such as fan-in and fan-out count) as indicative of an architectural change. Some considered updating an element or changing connection and dependency relation (connectors), and so on. Include-dependency and symbol-dependency (ID-SD) [151] metric considers the modification of classes and interfaces importing from different packages. These metrics are extracted by processing the directories, packages, code file inclusion, component size and numbers, class reference, and procedure call. However, proposed tool for these metrics are explicitly dependent on other techniques to extract architectural models and clusters (they are arbitrary and have no formal limit). Only expert intervention can ensure architectural change detection's accuracy of these metrics, and analysis of thousands of change versions is almost infeasible. In our research, we focus on the commits having module/system (higher) level and class level changes. Please note that, a module can be a sub-system, 3rd party library, and cluster of packages [41]. Overall, the change commits contain additions, removals, and moves of implementation-level entities from one module to another. This also includes additions and removals of modules themselves. We consider various rules from many of these metrics as architectural changes.

Change Operations Considered

From the literature survey, we have extracted the following operations in the design models (such as UML), architecture view, ADDL and relevant code elements considered to be architectural change: add, delete, move, split, update and component renaming. The summary of them are presented in Table 2.2. Existing techniques have considered various combinations (either all or a subset of them) of these operations for determining architectural changes. Fluctuations in various quantitative values (such as fan-in and fan-out

Table 2.2: Change operations and code properties considered for change detection.

Metrics	**Add**	**Delete**	**Move**	**Other**	**Value**	**Properties**
MoJo	Y		Y			Directory, code file
DSM					Y	Dependency
MoJoFM	Y		Y			Directory, code file
GKD					Y	Code files, methods
LILO					Y	Class
RC			Y	Y		Package, class, method
HEAT				Y		Code model
CDI					Y	Class, operations, ports
CB					Y	Module numbers, size
srcTracer	Y	Y		Y		Class, method, relations
NHK					Y	Class, dependency
CSM					Y	Package
GAR			Y	Y		Metamodels
A2A	Y	Y	Y			Directory, code file
C2C	Y	Y	Y			Directory, code file
ID-SD	Y	Y				Code file
DM					Y	Package
QC	Y	Y	Y			ADDL properties
MLC				Y		Directory, ..
CC	Y	Y				Class

count) are also indicative of an architectural change. DSM [24], EOC [152], LILO [233], GKD [181], CDI [117], CB [40], NHK [223], CSM [69], and DM [249] metrics are based on quantitative value. The rest of the metrics and techniques presented in Table 2.2 directly consider change operations. Quantitative metrics require fine-tuning of parameter values that may not be versatile. Metrics, their corresponding change operations, and which properties are used for measuring are presented in Table 2.2. *Other* column in the table represents updating an element or changing connection and dependency relation (connectors), and so on. Co-occurrence of various keywords (TD [172]) within the textual description also indicates architectural changes (to some extent). However, one surprising finding is that we have found one study (Mae [209]) that directly considered configuration and constraints for detecting architectural changes, which are the common elements of documenting software architecture.

From our change operation centric analysis, we see that defining the modification operations is essential before selecting a change detection technique. We also noticed that the HEAT [65], CDI [117], CB [40], GAR [29], and DM metrics [129] attempted to define (and a few of them validate) the systematic meanings of the change operations related to concerns and flaws in the codebase. For example, in HEAT, some change operations associated with a particular area of the systems are defined as the change in client-server relationships. A logical definition of the change operations such as this is more understandable by the development and maintenance team. Other studies have not experimented with or discussed how expressive or semantic so that the maintenance team can comprehend the two types of change detection metrics.

Software Properties Considered for the Change Detection

We have identified a list of concrete software documentation and codebase properties used for the change metrics: directory, package, code file inclusion, component size and numbers, class reference, and procedure call. Mapping of these properties with the associated studies is listed in the *Properties* column of Table 2.2. Many of the prescriptive architecture metrics leveraged UML properties, while a few of them utilized the ADDL properties (HEML and QC). From the *properties* column, we also notice that most of the detection techniques consider object-oriented properties. Surprisingly, no study (TD) explored textual properties within the developer's discussion for predicting a change task. Although the directory, code file, and method properties can be used to detect primary architectural changes, little focus has been given to detect meaningful architectural changes (in terms of modules, components, constraints, and connectors) considering various architectural views of the non-object-oriented systems. In terms of the descriptive architecture, we feel concerned in the recent consideration of architectural change instance [167]. In this concern, researchers should resolve the ambiguity that whether logical change coupling (two local segments from two components are frequently changing together without direct dependencies) would be appropriate for treating an architectural change instance or not.

Table 2.3: Software abstraction level supported.

Metrics	**High**	**Medium**	**Low**	**Type**	**Depends**	**Outcome**
MoJo	Y:cluster, module			Descriptive	Extract cluster	12.8% Q
DSM		Y:class, package, file	Y	Descriptive		NM
MoJoFM	Y:cluster, module			Descriptive	Extract cluster	Highest 65.8%
GKD		Y:class,file	Y	Descriptive		NM
LILO		Y:class		Descriptive		NM
RC		Y:package, class	Y	Descriptive		P,R >85%
HEAT	Y:component view			Descriptive	Extract model	NM
CDI		Y:class		Descriptive		NM
CB	Y:modules, views			Descriptive		0.80 SC
srcTracer		Y:class	Y	Descriptive	Define uml	Better than experts
NHK		Y:class		Descriptive		0.5 SC
CSM		Y:package		Descriptive		NM
GAR		Y:attribute		Descriptive		P 100, R 92
A2A	Y:cluster, module	Y:package		Descriptive	Define module view	NM
C2C	Y:cluster			Descriptive	Extract cluster	ARC 56 and ACDC 31
ID-SD		Y:file, class		Descriptive		NM
DM		Y:package		Descriptive		NA
QC	Y:ADDL			*Prescriptive, Descriptive*	Define and extract model	.50% efforts
MLC		Y:package, class	Y	Descriptive		100% F1 for DR
CC		Y:only class		Descriptive		NA

2.3 Categorising the Design-impactful Changes

In practice, software design decisions and changes are grouped from various perspectives. The grouping is done by focusing on the requirements, design decisions and concerns, design solutions, development and maintenance activities, design failings, and so on. Identification of these groups is essential for proper trade-off analysis to reduce current and future TDs. Lack of proper identification of them caused various TDs in numerous practical scenarios [234]. Software changes can be grouped [227, 127, 227, 125] based on - (i) requirements and design decisions – features, decision types, decision cross-relations, concerns, and tactics, (ii) development and maintenance tasks - change purposes, change impacts, and codebase resources, and (iii) design failure scenarios. This thesis specifically focuses on grouping architectural changes based on the development and maintenance activities. Design artifacts mostly document these changes.

2.3.1 Change types in the development and maintenance tasks

Change categories should be defined based on the development and maintenance tasks for better estimating, planning, and documenting software. Based on the change purposes in the development and maintenance tasks, change types are grouped as adaptive, preventive, corrective, and perfective. Mostly these are first defined by Swanson [226]. Some of the examples are shown in Fig. 8.2, where the *Breaking Changes* is referred to as the preventive change. The details of the change groups are as follows –

Adaptive – this change could be a reflection [246, 143] of system portability, adapting to a new environment or a new platform. Adaptive change might also occur for organisational and governmental policy changes.

Corrective – it is the reactive modification of a software product performed after deployment to correct discovered problems [55]. Specifically, this change refers to defect repair, and the errors in specification, design and implementation.

Preventive – this type of changes happen to improve file structure or to reduce dependencies between software modules and components may later impact quality attributes such as understandability, modifiability, and complexity [170, 246]. In other words, preventive changes happen to improve file structure or to reduce dependencies between software modules and components. Such changes may later impact quality attributes such asunderstandability, modifiability and complexity.

Perfective – these are the most common and inherent in development and maintenance activities. These changes mainly focus on adding new features or requirements changes [226, 246, 63]. Also, these changes areaimed at improving processing efficiency and enhancing the performance of the software. Thus, these changes can be both functional and non-functional optimizations.

2.4 Design Change Artifacts

Architecture (or design) description must be accurately and traceably linked to its implementation in order to handle the challenges of developing, maintaining, and evolving a critical software system. We refer to architectural design artifacts as the collection of one or more design change-related entities - change impacted components, modification operations, modified dependency relations, purposes of change, associated design decisions/ features, descriptive change summary, modified design document, design changelogs, design tactics, descriptive comments for the reviewers, and so on [187, 162, 115]. A few of the artifacts are discussed in the Motivation section. A design change artifact may contain (at least) - *"why the design change has happened," "what high-level program properties are impacted," and "what will be the probable descriptive summary/logs of that change"*. These artifacts are needed to be updated if changes happen [162]. These artifacts are valuable for reverse engineering as well because other software artifacts are linked and belong to them.

2.4.1 Steps of Generating Design Change Artifacts

Main steps (shown in Fig. 2.5) of generating design change artifacts and documents are described as follows.

1. Design Change Detection

During development phases, all change revisions or commits are not design impactful changes. Consequently, at first, the change revisions (or commits) that contain architectural design changes need to be identified to include their information in the change document.

2. Important Design Change Information Extraction

After detecting the design impactful change revisions, key change relations of the components (such as modules, libraries, packages, classes, etc.) are required to be extracted. In many scenarios, that information needs to be ranked for inclusion purposes because all the relations may not be considered when writing an architectural design document.

3. Determining and Categorising Change Purposes

Change purposes or change categories are the most fundamental information in the design change logs, release logs, and design decisions (requirements/issue). Therefore, after detecting the design revisions, they must be categorized into development and maintenance purposes (such as new feature addition, bug fixing, refactoring, etc.) for properly writing the design change document.

4. Description Generation

The final phase of the design change document writing is to produce a natural language description combining the above information of phases 1, 2, and 3. The description may also include a short form of design decisions

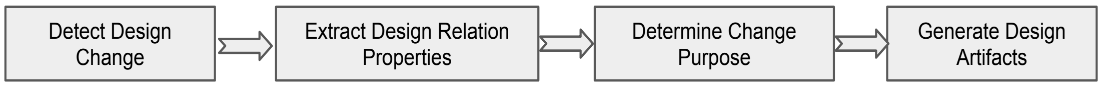


Figure 2.5: Operational phases of design artifacts generation.

with the modification operations.

2.5 Software Artifacts for Architectural Change Information Mining

2.5.1 Code Properties for Software Change Information Mining

Software codebase is the most reliable source of information as it is the original implemented resource. Following entities can be leveraged for the architectural change information extraction.

Change Operations

Abstract change operation types such as add, delete, modify, refactoring, etc., can be used for change information mining [242, 151]. Change in AST, method definition, body and even method location can represent specific change intentions. Identifiers in the changed code are also crucial for mapping and tracking requirement information change. The density of changed code can be used for commit classification.

DANS Properties

Following the initial conceptual design and architectural model a software codebase is structured into folders, sub-folders and files, and this structure is concrete [151]. These entities are the holders of the implemented software features. Code inside the files consists of identifiers, statements and inclusion of other files by the location/references. That said, the directory and naming structure (DANS) is a valuable information for architectural information mining; thus, change intention mining.

Meta Information

For change intention mining and code change summary generation in detail, source code properties such as project description, configuration files, deployment scripts, and so on can play an influential role [60, 254, 124].

Code Comments

Developers write comments in source code in natural language and briefly explain a function of code and what the code does [34]. Mining source code comments can be useful to analyze architecture-related information. In many cases, added, deleted, modified, or existing code comments may clarify the original intention to change a code segment or methods.

Semantic Change Relation

Structural semantic change relations (SSC) represent a meaningful description of relational information of the architectural change components. In the changed code, it might represent a more logical meaning of a change [47]. That said, an SSC is a self-explanatory property that at the same time indicates both the change operation type and direct change impact. For example, the SSC – *a new method with cross-module dependency* indicates that in a design change a new method is added that also impacts the dependencies of two modules. However, such relations are extracted from architectural component dependencies within the codebase [47]. Finally, they can be embedded with software meta information for mining change intentions.

2.5.2 Textual Features for Software Change Information Mining

Text analysis techniques have proven promising outcome to supporting automated software engineering tasks, for example, bug triage, program comprehension, traceability links recovery, and software documentation [34]. Some of the promising feature models for architectural change information mining are described here.

Sources of Textual Features

The main sources of textual features in the software development history are – design and requirement documents, commit messages, issue (features or tasks description) lists, developer discussion and messaging lists, review comments, code comments, release notes, and so on [13, 227, 195, 33].

Co-occurred Terms

Co-occurred terms are a set of words appearing in a sentence such as *"refactor organizational structure"* that might indicate a crucial semantic meaning. Term co-occurrence reflects subtle text organization that can be described concerning word interactions.

Discriminating Keywords Set

Selecting a set of keywords is promising in software repository mining [255]. Each keyword is treated as a feature in a text classification model. Discriminating feature/keywords selection (DFS) models can generate a specific set of keywords representing a specific type of topic [48, 255]. In this model, the vocabulary consists of the top N words ordered by descending values in the whole collection of topics (or messages). Various distribution models can select top N words [255]. A discriminating set of keywords can be used to mine software change intentions.

Word Strength

Words and their strengths to specific contexts are extensively used in sentiment mining [8]. Researchers [8] have been working on assigning numerical scores of a word/token/terms/synset to three opinion-related properties (such as the objective, positive, and negative) for sentiment detection from the natural language texts. These three scores are distributed from 0.0 to 1.0, and the totalling score is 1.0. This means that a term may have nonzero scores for all three categories. This indicates that the corresponding terms have a certain degree of the three opinion-related properties. A graded score to a term for a certain opinion may have three interpretations [8]: *(i) the terms in the synset are Positive only to a certain degree; (ii) the terms in the synset are sometimes used in a Positive sense and sometimes not, depending on the context of use; (iii) a combination of (i) and (ii) is the case.* A similar way of score grading and interpretation for certain words might be possible for the four architectural change intentions (perfective, preventive, adaptive, and corrective). Word strength mechanism can improve the discriminating keywords models for change grouping. It prompts experimentation with real-world architectural change data.

Concept Tokens

Concept tokens [200] are the contextual occurrence of words that dominate a sentence to clarify a meaning, such as {*update; API; versioning*} indicate adaptive change more confidently within a topic. Similarly, if we consider a description – *"add support for services down"*; the concept tokens {*add, support, down*} within this sentence may indicate flaw fixing. However, all the words in this sentence are general tokens. Formally, a concept is a triple *K=(G,M,I)* called a formal context. This triple is represented in a cross table consisting of a set of rows G (called objects such as adaptive), columns M (called attributes such as update, API) and crosses representing incidence relation I (such as three words co-occurred together). Concept tokens with a strength assigning strategy to them can be promising for change intention mining.

2.6 Applications of Architectural Change Detection and Categorization

There are many implicit implications of architecture change detection and categorization (ACDC) in software evolution support, maintenance support, and fault detection and change propagation [246, 205, 108]). Instead, we discuss some explicit contexts (and mostly regular supportive development and maintenance activities) where ACDC are essentials.

- ***Design Review***: A typical software design review procedure consists of extraction of requirements, extraction of design and its change, construction of causal relationships, the discovery of potential design issues, and so on [227]. Proper change information extraction helps to execute these procedures. However, 45% rejected pull requests in OSS projects contain design inconsistency [217]. Moreover, software projects need to avoid degradation as erosion, drifts as well as architecture pendency [108]. To help eradicate these challenges, architectural change detection is required to check the differences between the proposed and implemented design and whether it complies with the design guidelines [227].

- ***Design Document Generation:*** More than 60% developers and 85% project managers are likely to use architecture/design documents [45, 22]. Furthermore, more than 40% major release notes and 15% minor release notes contains design changes [33]. In such software documents, change category (i.e., new feature addition, restructuring, etc.) is the fundamental information. There are also pieces of evidence that design change logs (even with the releases) are also maintained for the development and maintenance teams by both the industrial and OSS projects[1]. That said, change detection and categorization are mandatory for various software change document generation.

- ***Architecture and Design Decision Recovery***: Many projects do not document architecture and design decision associativity with the components in the initial phases. But later detection and categorizing of the change revisions are essential to recovering and documenting the software architecture, and design decisions [109, 83, 214].

- ***Change Tracing and Change Impact Analysis***: Change detection is required for change impact analysis [19, 246]. Design decisions (i.e., flaw fixing and new features) and corresponding changes can only be traced through detection and categorizing techniques [214, 32, 93]. Moreover, architecture change tracing would facilitate on-demand artifacts extraction for code comprehension and other decision-making purposes because architecture is considered the primary artifact to trace other artifacts [34].

- ***development and maintenance tasks and Release Planning***: Design changes and their categorization is specially used for release and milestone planning (along with workforce assignment) for the development and maintenance teams [57]. For example, the component that implemented a new feature in this release may require design review and restructuring in the near future. At the same time, components that have gone through restructuring may not be planned for refactoring in the near future. Design change partitions of the detected change instances and their categories are also helpful for timely and orderly backporting them [139, 54].

- ***Developer's Profile Buildup***: Balanced team development and proper workforce utilization are crucial for a project [145]. Moreover, an organization may require searching for relevant experts to employ for resolving project design challenges [176]. Finally, mapping categorical architectural change tasks (i.e., design refactoring) with the involved developers is crucial for these purposes [30].

[1]https://tinyurl.com/jby4evdh

Chapter 3

Architectural Change Detection from Textual Document

In this chapter, we explore textual properties in the developer discussion (for design decisions) and commit messages for detecting architectural changes. Because the codebase size of the software systems varies project wise. As a result, the mentioned challenges in Chapter 1 also increase proportionately. In contrast, the range of the textual description for each change task is almost constant and does not vary significantly. In order to develop a textual model, we first followed the study of Ding et al. [63] where we investigated the source code and commit messages of three projects (Galaxy, iPlant, and ImageJ) and mailing threads of two projects (Hibernate and ArgoUML). After that, we review a number of studies related to architecture documentation and change scheme descriptions [246, 115, 11, 194]. Then we compiled four steps: (i) keywords extraction, (ii) keywords refinement, (iii) co-occurred terms extraction, and (iv) detection model development. Through the first three steps, we manually extracted around 130 co-occurred terms that may represent architectural change related activities. Finally, deploying the TF-IDF and term graph, we proposed an architectural change commits detection technique with those co-occurred terms.

In the subsequent sections, we describe our experimental methodologies and insights. Section 3.1 presents a brief overview and motivation of our study. Section 3.2 presents the dataset collection process. Section 3.3 reports architectural information contained in textual descriptions. Section 3.4 discusses co-occurred terms extraction. Section 3.5 presents our proposed techniques for architectural change detection. Section 3.6 presents performance results. Section 3.8 concludes our study with future work.

3.1 Introduction

Software architecture is important both in the short and long-term, and it can support fast customization to help satisfy various stakeholder needs [158]. However, a rigid, static architecture does not address these challenges effectively [210], and so continuous architecture [103] has become a recently adopted practice to cope with the evolving nature of software. With the advancement in cloud-cluster processing for Big Data, continuous architecture[1] has become more important [158, 193, 31] than ever to support the analysis,

[1] Architecture enhancement through refactoring and redesigning to align with the Agile development practices.

development, and evolution of software architecture that can accommodate domain specific Big Data technology [210]. Architecture continuity refers [193] to the ability of a system to change its architecture and maintain the validity of the goals that determine the architecture. With technological evolution and availability, architecture continuity directly impacts economic sustainability [193]. Researchers and practitioners investigate [31, 193] change models and change rules as key tasks for managing change to support architecture continuity. Extracting reusable models [210], patterns and designs, and understanding the causes and impacts of change helps practitioners conduct various aspects of continuous architecture such as repairing and redesigning the software [31], detecting anti-patterns, architecture verification, and training new developers on a project. Determining the causes and categories of the changes is essential to help practitioners with the aforementioned design decisions; it also allows researchers to create models to prevent architecture knowledge vaporization and degeneration [63].

Researchers have been developing architecture change characterization scheme [246, 247] since *"it allows engineers to group changes based on different criteria, e.g. the cause of the change, the type of change, the location of the change, the size of the code modification or the potential impact of the change"*. One of the benefits of change classification [246] is that engineers can adopt a common solution to address similar changes hence reducing overall costs compared with addressing each change individually. Architectural change classification provides extra information about defects which is vital for many tasks such as prioritizing architecture defects, improvement of the defect prediction, assignment of defects to developers, architecture defect resolution, and identifying the quality of components. Empirical studies [247, 19] with real-world projects have shown the effectiveness of this approach. However, most of the available analysis methods are manual and costly. Consequently, more automated techniques are warranted to speed up the practice.

Architectural message detection and classification techniques are useful for automatically generating an architectural change summary after each release which is important particularly for open-source-projects. While changes are easily detected from source code revisions, commits and discussions must be analyzed to identify the intention and category of changes. Kazman et al. [115] found that the development and contributing activities of a Big Data framework, HDFS (Hadoop Distributed File System) increased after adding software architecture documentation. As architecture documentation is increasing more than ever before, automated approaches can be useful for generating architectural documents efficiently. In addition, a number of studies are available for the study of architectural evolution, change pattern and knowledge discovery from source code. For instance, Ahmad et al. [14] proposed methodologies for identifying architectural change patterns from source code changes using a sub-graph mining technique. To apply or to reuse these patterns in real-world scenarios, it is necessary to identify the underlying features/behaviours, contexts, and reasons behind the architectural changes (and their categories). That information can also be extracted efficiently from developers' discussions in relevant change activities (such as commits) using an automated technique.

Code-reviewers and developers use natural language descriptions in discussion, commit, and communication messages when performing architectural change and to support developer awareness of that change [194].

Therefore, automatic architectural change analysis techniques would be cost-effective for both knowledge seekers and knowledge providers, and would help automate architecture level code-review comment generation [194] by parsing the past change pattern and message history.

In this study, we conducted an exploratory study to investigate whether free-form natural language text (e.g., communication, commit messages, and so on) can be utilized for automated architectural change analysis. In doing so, we primarily focused on four research questions to develop the automated approach described as follows:

RQ1. What types of architectural information can be extracted from the commit and communication messages that are valuable for software improvement, maintenance, and evolution?

RQ2. What are the most crucial natural language properties for describing the architectural changes that can be used as distinguishing characteristics to the software architects, reviewers and developers in the written document?

RQ3. How can we automatically detect architectural change related messages to support a software team to extract useful knowledge and its application in product development from a large collection of documents?

To answer RQ1 and RQ2, we first manually analyzed a number of published studies [246, 63], commit messages and developers' mailing lists rich in architecture change information for five well-known open-source projects (Galaxy, iPlant DE, Hibernate, ArgoUML, and ImageJ) and then annotated the architectural change related information. Hence, we handcrafted important keywords or terms (such as restructure, modularize and so on) adopting the information presented in SACCS [246] and IEEE 1471-2000 [11], and developed a natural language model to represent architectural details from manually labelled documents. To answer RQ3, we experimented with TF-IDF and term-graph [36] based techniques for the detection of documents relating to architectural changes. Term co-occurrence (e.g., a set of words appearing in a sentence such as *"refactor organizational structure"*) reflects subtle text organization that can be described concerning word interactions. A network of connections formed in a term-graph approximates human capability in the process of discourse understanding [36]. We used this approach to identify architectural change description from discussions.

While our detailed experimental results are discussed in Sections 3.5 and 4.4 and shown in Tables 3.4, 3.5 and 4.3, in summary, our automated approach obtained approximately 74% precision on detection with Term-Graph and 61% in F1 score for TF-IDF. Overall, our study reveals that automated architectural change analysis tool would be fruitful only if the developers provide considerable technical details in the commits messages or so as we have observed within the jvm-core dataset (96% architectural samples are detected). In summary, this study focused on the following directions:

- A revision study of what pieces of information are contained within the commit messages and discussions about architectural aspects.

- A natural language model containing 166 keywords and 130 co-occurred terms expressing architectural changes in commit and discussion messages.

Table 3.1: Data samples of the candidate projects for our study (we consider a portion; since manual inspection of all messages takes significant time). GIT = github.com.

Project	**Version Tag**	**Total**	**Samples**	**Data source**
Galaxy	16.04	1392	530	GIT/galaxyproject
Hibernate(ORM)	4.0.0,4.3.0, 4.3.1	2000	310	GIT/hibernate
ImageJ	2.0.0beta	1155	300	GIT/imagej
iPlant (DE)	1.9.0,1.9.5, 2.6.0	513	250	GIT/cyverse-archive
ArgoUML	Mailing list	∼	37	argouml.tigris.org

- A TF-IDF based technique to detect probable architectural changes.

- A term-graph based versatile method for detecting architectural change messages.

3.2 Dataset Collection and Study Design

We have collected the codebase, commit history, e-mailing lists, and releases of the five popular open-source projects. Two of the projects, ArgoUML and Hibernate were used in an earlier study [63]. Another three, Galaxy, iPlant DE, and ImageJ have been widely used for several years for Plant Phenotyping and Genotyping analysis. Galaxy and iPlant DE are large cloud systems integrating many advanced modules for handling massive and various types of data, and architecture change management is critical for them. Each of the projects contains thousands of commits and many releases. The description of the collected project's artefacts is presented in Table 6.1.

In this study, we mainly focused on commits and selected overall 1350 commit messages containing more than two words. We randomly picked the samples from the versions (which have gone through the highest number of packages/modules changes) of the Galaxy, Hibernate (ORM), iPlant Collaborative, and ImageJ projects respectively. Moreover, since the closest work [63] experimented with mailing threads, we added 40 mailing threads from Hibernate and 37 from ArgoUML project (we collected ArgoUML data from Ding et al. [63] since during our data collection and experiment time the history server was down) in our experiment. We manually annotated these samples (took around 60 working hours per person) which have gone through different architectural change operations mentioned in Section 4.2 by analyzing the changes in the source code revisions. Please note that we did not consider a commit as architectural change if the component changes are only in the web-page (HTML, CSS, JavaScipt) design and test-code modules. We prepared 927 training samples and 500 test samples primarily. In the subsequent sections, we discuss our methods and findings in detail.

3.3 Architectural Information Contained in the Development History

Knowing the possible architectural information contained within the discussion messages is important before developing the automatic technique. Therefore, we explored what benefits developers and practitioners might get if an automated tool is developed for architectural change mining. We have done the following steps to find the answer to research question **RQ1** (What types of architectural information can be extracted from the commit and communication messages ....?). We first followed the study of Ding et al. [63] where we investigated the source code and commit messages of three projects (Galaxy, iPlant, and ImageJ) and mailing threads of two projects (Hibernate and ArgoUML). After that, we review a number of studies [246, 115, 11, 194] related to architecture documentation and change scheme descriptions. In our analysis, we found that along with the categories for changes presented in the studies [246, 63], there are much more other valuable architectural related information can be extracted that are briefly described as follows:

- Change Operations: Information of different kinds of change operations can be contained as discussed earlier [14, 65].
- Contexts and category of changes: Researchers [63] extracted premier reason of changes from messages. Likewise, context of change request [246] are also discussed in them.
- Micro-architectural change of sub-components: For example, structure of *pulsar* sub-component with Galaxy has been changed due to issues.
- Run-time behaviours that might trigger changes: Information that HDFS design changed because of inconsistencies in run-time behavior is discussed in the mailing lists.
- Integration of API for architectural enhancement: For example, iPlant structure is changed for *gin* API addition in the commits.
- Possible impact of technological changes [103].
- Stability of the architecture of a system [23]. Complexity estimation [246] of a system's structure are also discussed in the document.
- Developers knowledge-base (such as awareness and how to deal with change) and complexity with the architecture [194, 103, 51].
- reusable solution, design concept, source code change pattern, and system model [15, 65].
- Various emerging architectural practices [16, 158] such as collaborative analysis for re-architecting for Quality-of-Service model.

The above information provided us the answer of RQ1. This step also helped us defining the co-occurred terms, e.g., if a commit message in Galaxy contains API integration and if our source code analysis of Galaxy reveals that corresponding commit performs modification of the API, we define a co-occurred term "*API integration*".

3.4 Natural Language Model Development

Language models which can accurately place distributions over sentences not only determine complexities of language such as grammatical structure, but also refine a fair amount of information about the knowledge that a corpora may contain. Intending to answering question **RQ2**, in our baseline natural language model development process, we compiled three steps: (i) Keywords extraction, (ii) Keywords refinement, and (iii) co-occurred terms extraction. These steps are discussed as follows.

Extracting the Primary List of Keywords: Keywords play an important role in mining contextual information from natural language texts. During reviewing the published works as mentioned in Sections 4.2 and 3.3, we listed the keywords used to or related to the architectural description. After that, we frequently checked which distinguishing words might have been used to describe architectural changes. We created a primary list of words that are used and recommended implicitly by recent studies [246, 11]. Finally, we extended the keywords list by manually scanning the annotated commit messages and mailing threads which represent actual architectural changes at the codebase. Hence we listed around 166 possible distinguished keywords. However, these keywords may occur in the other commits and mailing threads which do not represent architectural changes. Therefore, we need to consider the significance and refine the terms for which consequence is negative.

Significance Calculation and Refinement: Strength or significance calculation is really important to verify how keywords influence a certain lexical context; for instance, researchers have been developing a database, SentiWordNet [8, 21] containing the strength of almost all the English words towards emotions for sentiment analysis. Motivated by SentiWordNet, to figure out the significance of the listed terms we calculate the probability of each term being architectural change commit. The probability of a keyterm T,

$$P(T) = \frac{AC(T)}{C(T)}; \qquad \text{Significance } \theta = P(T) - \frac{N(T)}{AC(T)}$$

Where $AC(T)$ is the number of architectural change commits that contain term T, and $C(T)$ is the number of all commits where T is present. Where $N(T)$ is the number of non-architectural change commits that contain key-term T. To calculate significance (θ) of a key-term we subtract penalty with the probability $P(T)$. The terms with the positive significance and corresponding $P(T)$ are presented in Fig. 3.1. From the term significance analysis, we observed many key-terms show negative values mean they are significantly present in non-architectural commits as well. The full list of our refined key-terms will be available upon request. Among the 166 extracted terms we found 37 terms of positive significance within the dataset. However, many terms if they co-occurred in a sentence might have distinguishable significance comparing to non-architectural commits. In the next section, we will figure out the co-occurred terms representing architectural commits.

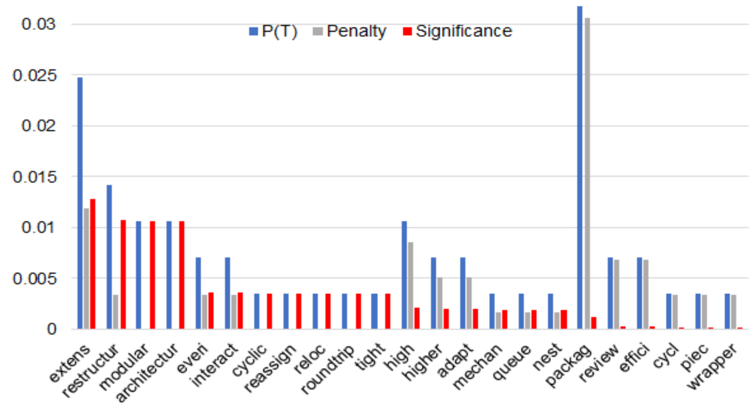


Figure 3.1: Top 24 stemmed terms distribution and significance.

Table 3.2: Sample commits that contain the intention of architectural changes

#	Text	Key-terms
1	**Migrate** imagej.ext **classes** to imagej base **package** (We leave only ...to avoid an issue with old versions of the ImageJ updater.) This is part of an effort to **make** the ImageJ package **structure simpler** and easier to understand	migrate classes package
2	**Add interface** for **objects** housed by a **UI** This UIComponent interface is shared between InputPanel and InputWidget,and could potentially be useful for other composition-style UI.. in the future.	add interface objects UI (not architectural change)
3	Tweaks to the recent Tool Shed **API enhancements**, making more RESTful.	API enhancements

Co-occurred Terms Extraction: The extracted key-terms can be used with the *TF-IDF* (Term Frequency-Inverse Document Frequency) [190] based binary classifier to detect the architectural messages. We experimented with our significant terms (negative terms as well) for the classification of the messages. However, the classification threshold varies according to the contexts. Besides, *TF-IDF* based technique also have adverse effects on noisy text data. Therefore we look for another unique technique which has the relative lower bias on the context and the noisy data that affects the accuracy. Key-terms (according to the context) are being used in many techniques for mining significant information from software artefacts. In natural language processing, n-gram model [42], the contiguous sequence of n words from a given sample of text is used to extract special types of representational characteristics of a collection of documents. However, in this context, contiguous words of n size does not represent (as can be seen in Table 3.2) architectural information in the most of the cases. After careful analysis, we found some interesting natural language patterns consisting of two to four words (not necessarily contiguous) in a sentence (many of them are present in SACCS-Software Architecture Change Characterization Scheme attributes) that express the intention of the developers about architectural changes; we call them as co-occurred terms. All co-occurred terms should be present in a sentence to express the architectural change. For example, the terms: make, structure, and simpler (contained in the sample #1 in Table 3.2) express that the structure of a module has been changed to

Table 3.3: Representational co-occurred terms of architectural change.

Co-occurred-terms	**Weak terms**	**Neutral terms**
design improvement	visualizations decompose into	distributed object free
decouple function	repository dependency hierarchy	component review approval
enhance process	enhance installing dependency	processing state runner
infrastructure improvement	enhancement displaying	adjust logic module
api add	merge changes	merge job changes
new applications module	ui design improvement	context dependency resolution

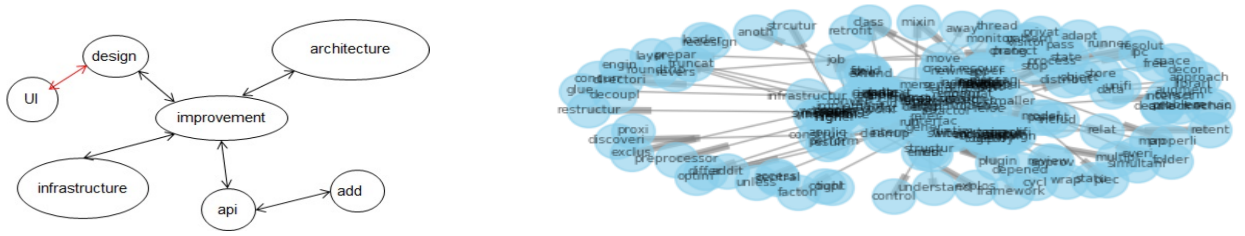
*We have identified 130 co-occurred terms. Presence of these terms in a sentence most likely to express architectural commits.

make it more simple. Thus, we found ~ 200 commits express architectural changes explicitly in their messages. Some of the samples of the commit messages are presented in Table 3.2. We also observe that some sentences do not mean architectural change even though the co-occurred terms exist along with some other terms (i.e., UI, visualization, display). Some co-occurred terms are ambiguous about explicit architectural change. We classified these co-occurred terms into three types: (i) explicitly represent architectural changes, (ii) weak terms − most of the cases they do not represent architectural changes but still co-occurred with the keywords, and (iii) neutral terms − might be contained in both types of messages. Some of these terms are presented in Table 3.3.

Initially, we considered 927 samples only to prepare a ground truth for the unseen test data of 500 samples. After a thorough investigation of those samples, we found that two to four terms are enough to represent architectural change activities in the commit messages for most of the cases. So far we manually extracted around 130 co-occurred terms that may represent architectural change related activities. Furthermore, with the key-terms, it is possible to generate a subset of combinations of key-terms into 2-terms, 3-terms, 4-terms, and so on. However, it is a costly operation to generate and test for the best combinations (in future, we will work in this direction). In the next phase, we will discuss our experimentation with the extracted co-occurred terms to auto-detect the architectural commits.

3.5 Architectural Change Message Detection

We explore a few popular text retrieval techniques for architectural change detection leveraging the extracted co-occurred terms. We develop two detection models and compare their outcomes - (i) Term frequency-inverse document frequency (TF-IDF), and (ii) Term-graph. Existing studies leverage TF-IDF to categorize bug reports [221]. Whereas a term graph might provide more control logic for information retrieval within a



(a) Sample graph

(b) Snapshot generated by *matplotlib* and *networkx* Python libraries

Figure 3.2: Key-term Graph related to architectural changes (red edge means not an architectural change), and a snapshot of the Term graph containing 130 terms (stemmed).

textual description than TF-IDF.

3.5.1 Change Detection using TF-IDF

Term frequency-inverse document frequency (TF-IDF) represents a statistical model of keyterms within a document reflecting the significance of an individual word in a collection or corpus [190]. The basic equations of TF-IDF calculation is presented in Eqn (3.1), (3.2), and (3.3). In this model, the Term-frequency (tf) is calculated from the relative frequency of a term in document d. The inverse document frequency (idf) is the measure of how much information a word provides. It is the logarithmically scaled inverse fraction of the documents that contain the word.

$$tf - idf = tf * idf \tag{3.1}$$

$$tf(t,d) = 0.5 + 0.5 * \frac{f(t,d)}{max\{f(t,d) : t\epsilon d\}} \tag{3.2}$$

$$idf = \log(\frac{N}{n_t}) + 1 \tag{3.3}$$

$$Here,\ N = total\ number\ of\ documents\ d\ in\ the\ corpus$$

$$n_t = total\ number\ of\ documents\ where\ the\ term\ t\ appears$$

$$f(t,d) = frequency\ of\ term\ t\ in\ all\ documents$$

The common TF-IDF model has a bias towards longer documents. To handle this, we have employed the enhanced version of TF-IDF. Here, we used augmented frequency for tf calculation as shown in Eqn (3.2); and employed smoothing for idf calculation in Eqn (3.3). We consider the total value of TF-IDF of all the co-occurred terms > 1.50 within a textual description as an architectural change sample. A value greater than 1.50 means there is the possibility of being more than one term within the texts.

3.5.2 Change Detection using Term Graph

A text graph [36] is effective for numerous applications involving information extraction from natural language text documents. In our work, we represent the key-terms as a graph (i.e., a term-graph), where vertices

Data: CT - Co-occured terms, TD - textual description, E - edge of a graph, N - node of an edge

Result: AC - Architectural Change True or False

```
begin
    AC⇐ False
    refTG ⇐ generateTermGraph(CT+NT)
    for all sentences ($S_i$) in a TD do
        sTG⇐ generateTermGraph(S)
        for all edges in refTG do
            Search Ns of a E within the shortest path ($P_N$) of sTG
            if $E_{weight}$ is 2 then
                Check NT and consider the existence of Ns within the $P_N$, set AC=True
            end
            if $E_{weight}$ is 3 then
                Check any connected predecessor or successor of $E$ within the $P_N$, set
                 AC=True
            end
            if $E_{weight}$ is 4 then
                Check any connected predecessor of predecessors or successor of
                 successors of $E$ within the $P_N$, set AC=True
            end
        end
    end
    If any $E$ of refTG is found in any sTG of TD then AC is True
end
```

Algorithm 1: Architectural change detection with term graph.

correspond to terms, and edges correspond to co-occurrence between the two terms. Specifically, edges are drawn between vertices if the vertices co-occur within a sentence of a commit message. In this study, we first generated a base or reference termgraph with all the collected co-occurred terms. A sample term-graph is shown in Fig. 3.2(a). During scanning all the commit messages, a temporary term graph of each sentence of a commit message is generated. After that, we searched a path in the temporary graph from each edge contained in the base graph. If the temporary graph contains a path then that sentence is the candidate sentence representing an architectural change commit. However, we see many cases where key-terms do not exactly represent architectural changes such as #2 sample containing UI related terms as shown in Table 3.2. Thus, we did not consider those sentences as decision-making sentences in the commit message. We set the weight of each edge of a path corresponding to the number of the co-occurred terms that represent the architectural changes. First, an edge containing two terms is matched in the base graph. Then we check the weight of the edge and consider the sentence as a candidate if it contains the similar number of terms corresponding to the weight in the adjacent edges in the base graph. This technique is similar to mining a sub-graph from the base graph. We construct the term graph with Python NetworkX[1] library. We have a different approach than the traditional usage of the term graph [36]. The algorithm for change detection with the term graph is shown in Algorithm 1. The *generateTermGraph*() function generate a term graph by assigning an edge (E) weight with the length of the words containing in a co-occurred terms (CT) pattern. The length of the shortest term pattern will replace the weight of an existing edge.

3.6 Experimental result

3.6.1 Experiment with Change Commits

At first, we measure performance with the change commits. A snapshot of the generated term-graph with our proposed model is presented in Fig. 3.2(b). Here, weight is significant to track the number of nodes of a path for filtering non-representative commits. The experimental result of the model with our prepared samples is presented in Table 3.4. Since the F1 score represents both the precision and recall rate, we will highlight it in most of the discussions [89]. Let's consider a test set contains a total of 100 samples where 50 are from class A and 50 are from class B. If a detection model predicts 60 samples as class A (where 20 are from class B), then the precision rate would be *50/60*. Now, if the relevant samples from class A among the predicted samples are 40 (60-20), then the recall rate is *40/50*. Hence, the classification accuracy (a special type of precision) of the term-graph with our 500 test samples is 64%, and the F1 score is around 60% for the **ImageJ** project (recall rate is 59%); whereas these rates are 61% for TF-IDF classifier [190]. Therefore, term-graph produces a more promising outcome in case of accuracy, and the F1 score is more promising for TF-IDF. For all samples, precision (P) and F1 are 56.35 % and 56.73% respectively for the best case. Overall, our language model

[1]https://networkx.org/

Table 3.4: Performance of the classifiers using the language model (F1 is calculated from precision (P) and recall).

Classifier	**Metric**	**Galaxy**	**iPlant(DE)**	**ImageJ**	**Hibernate**	**Total**
TF-IDF	P	50.63%	50%	61.1%	48.1%	54.69%
	F1	48.11%	46.83%	**60.96**%	48%	56.73%
TGraph	P	62.23%	73.3%	**64**%	26%	56.35%
	F1	54.76%	38%	60%	26%	54%

is promising as some commit messages are falsely identified due to the cross natural language effect. The number of false positive can be reduced, but in that case, essential commits may be skipped.

3.6.2 Experiment with Change Review Documents

Paixao et al. [194] collected a code review dataset of architecturally significant changes in four popular OSS projects from the Gerrit code review platform. Therefore, we experimented with our proposed techniques with this dataset to observe how accurately architecturally significant change samples can be retrieved. Our experimental results for reviewed change samples ($\sim$ 1441) and architectural awareness discussion samples ($\sim$275) are summarized in Table 3.5. Although the domains are different, the model can detect architectural change review documents up to 74% (for **java-client**) recall rate. Likewise, our architectural change model can detect up to 96% (for **jvm-core**) architectural awareness discussion from overall review documents. In all the cases, TF-IDF shows better performance, but TF-IDF requires threshold adjustment that varies contextually. Although Term-Graph performance is lower than TF-IDF, it is a versatile solution, and term-graph performs well where details are presented (i.e., F1 for Term-Graph for **Galaxy** in Table 3.4 is better than TF-IDF). The detection rate for awareness document in a different context proves the potential of our proposed language model, and we believe including some contextual properties would increase the detection rate in a significant way. However, many architectural-change commits that do not contain explicit natural language about the change may not be detected with this technique. As we have seen with the experimental result of **jvm-core** project, when intention of the commits are expressed in a bit of details almost all of them are detected (96%). Furthermore, once these commits are identified from thousands of them, project manager and software architects can conduct further analysis in accordance with source-code change patterns and design architectures for solution modeling, applying or re-using in real world scenarios.

3.7 Related Work

In this section we discuss related work based on different kinds of automated analysis approaches in software engineering which are discussed in the following.

Table 3.5: Experimental outcome of the Paixao et al. [194] dataset (partial) annotated as architecture aware (documents are detected from the defined samples containing architectural information, and we cannot calculate the F1 score as the dataset defines only a portion that are architecture aware).

Type	**Classifier**	**java-client**	**egit**	**linuxtools**	**jvm-core**
#Annotated		184	484	474	299
RReview	TF-IDF	**73.6%**	72%	60%	52%
	TGraph	**23.07%**	15.02%	17.34%	13%
RAware	TF-IDF	92.5%	89%	78.48%	**96%**
	TGraph	40%	**47.2%**	32.55%	28.57%

Here, RReview is the percentage detected architectural change reviews. RAware is the percentage detected where discussion is related to architectural awareness by the developers (details can be found in [194]).

Software artefacts analysis: Existing studies have focused on architectural evolution and issues during software development phases in various domains (business, industry, development tools, and so on). An empirical study [57] with more than 217 developers reports that 58% of them are interested in investigating architectural evolution but found it difficult to analyze as fully automatic technique is not yet developed. Numerous techniques and methodologies have been studied for analyzing bug reports [57], code-review comments [194, 256], usual software change from messages [255, 80], release notes [119, 177, 57, 185], and API documents [199]. In this study, we focused on to architectural change messages analysis. In the recent time, continuous architecting practice is being used for dynamic and changing system development [103, 31, 193]. For such a practice, previous activities should require [103] ongoing analysis of a system under development and re-architecting. More specifically, architectural knowledge gathering is essential [194] for the whole development team as architectural erosion and changes [74] have a more significant impact on development costs and efforts. Having said that, before any changes and implementation of a new feature of the system it is fruitful to extract experience and knowledge-base about changes, impact, and the reason behind the changes. In order to do that, existing studies [115, 194, 63] focused on manual analysis. In this study, our objective is to devise an automatic technique for analyzing architectural change related documents.

Search space reduction for software artefacts analysis: Architecture team [186] requires automated tools support for various aspects such as software framework to aid change summarizing and reviewing. For developing these tools, an efficient automatic technique is essential. During development and maintenance life-cycle, a project has thousands of artefacts, and a small subset of them is the point of interest to the experts for the relevant analysis. Most of the existing analysis studies [194, 63] selected a partial subset, even from a single project due to painstaking efforts of manual analysis. Therefore reducing the irrelevant information is essential for speeding up the analysis task. With a view to this, a few studies [28, 186] proposed automatic techniques to reduce search space for software artefacts analysis. A few studies focused on to the automatic software change and impact analysis from development artefacts. Behnamghader et al. [28] analyzed the impact of each commit (their dataset contains $\sim$ 20,000 commits, the size of the whole data of a project is multiple times of this amount) to the main module of a system as error-proneness. They describe it as quality evolution analysis of a system. However, an automated technique is essential to cover a wide area and reduce search space. In another study, Nejati et al. [186] proposed a technique for reducing search space to analyze change impact from requirement statement and design elements; where requirements and design information is essential as input into their technique. Unfortunately, no automatic technique is available for extracting only architectural messages from thousands of datasets. In our study, we proposed a method to reduce search space for analyzing architectural changes from the commit and communication messages.

Automatic software change analysis: The automation of software engineering problems utilizing machine learning techniques is increasingly being studied. With source code change, researchers also analyze developer discussion and communication messages to understand various aspects of software architecture (e.g., developers awareness of architecture change [194]). A number of studies [64, 194, 63] manually analyzed commits, discussion messages, and release notes to mine significant information about architecture. Unfortunately, manual analysis of thousands of documents is time-consuming and not-feasible to replicate frequently. Nazar et al. [185] stated different techniques for the automatic summarization of software artefacts from development history; however, they report no method for summarizing architecture. Moreno et al. [177] proposed an automatic technique for release note generation from change history. Likewise, Klepper et al. [119] propose a semi-automatic release note generation for different viewers (customers, project managers, testers, developers). However, both of these works did not include architectural change summary due to lack of automatic technique. Our proposed architectural messages detection and classification technique are also useful for generating automatic architectural change summary. Natural language document is an excellent source of architectural information [64], and an automatic technique would have a significant contribution in this domain. A proper natural language model can leverage the properties of the texts to automate the analysis. The architectural information described by the researchers [246, 11, 115, 63] can be exploited to define architectural change description within the discussion topics. Our study explicitly follows the guidelines and insight about architectural change characteristics directed by these works. In this study, we identified a collection of keywords and co-occurred terms describing architectural changes. Term-graph [36] based techniques are being used for information-retrieval in various domains including software engineering by defining a textual model. In this study, we also proposed a term-graph [36] based technique for detecting architectural change documents.

3.8 Conclusion

In this study, we explored the feasibility of performing architectural change analysis automatically to reduce manual analysis efforts. We developed approaches both for architectural change detection using various kinds of natural language text including review comments. Our experimentation with the TF-IDF and term-graphs showed that our developed model can detect change related samples moderately (61% F1). Overall, our empirical study with various projects concludes that an automated architectural change analysis tool would be fruitful only if the developers provide considerable technical detail in the commits messages and other text as we observed in the jvm-core dataset (96% architectural samples were detected).

In future, we will extend our study by including more software projects and more sources of information (e.g., issue tracking systems), which will allow us to enrich our model with a wide range of key-terms and to develop a reusable reference textual database (like the prominent 'SentiWordNet' database for sentiment extraction from text) for automated architecture change analysis with higher precision and accuracy which would be helpful for the practitioners in various ways (e.g., for efficiently generating architectural documentation and artifacts) which in turn support architecture continuity.

Our experiments reveal that the textual description should contain considerable details for detecting the architectural changes accurately. Moreover, the co-occurred terms should be present in the description. Therefore, we will explore the lightweight code properties of the software systems in a future study (Chapter 6). However, as soon as the architectural changes are detected, they must be categorized for real-world usage (as we described in Chapter 1). That said, our next study (i.e., Chapter 4) focuses on change categorization.

Chapter 4
Architectural Change Categorization using Discrimination Feature Model

As soon as the architectural change revision is detected, it is also required to determine the cause or purpose of the change to better represent the change knowledge and generate various software documents (such as release notes [178]) along with other things discussed in Section 2.6 in Chapter 2. In this chapter, we present our study on architectural change classification. We have observed that the developers write change category information in almost all the release logs (presented in Chapter 1). Therefore, at least for descriptive summary generation, it is mandatory to categorize the architectural changes. To that end, we first explore the existing techniques for software change classification. In this study, we extracted four discriminating sets of keywords for four types of architectural changes (perfective, corrective, preventive and correct) from experimental training data using DPLSA [255]. Then these keywords sets are used for generating prediction models using LLDA [149], SemiLDA [80], and DPLSA [255]. These models are referred to as discriminating feature selection models (DFS). However, this is the first study that specifically focused on architectural change categorization.

This chapter continues as follows. Section 4.1 presents a brief overview of our study. In Section 4.2 we discuss the background and motivation of architectural change classification. Section 4.3 describes our dataset preparation. Our explored classification models are presented in Section 4.4. Section 4.5 reports our experimental outcome and Section 4.7 discusses threats to validity. Section 4.6 discusses related studies and Section 4.8 concludes the chapter with future direction.

4.1 Introduction

Software architecture can be changed during regular development and maintenance activities – new feature addition, bug fixing, refactoring, and so on. Williams and Carver linked the causes of architectural changes to four Lehman's law of software changes [132] – continuing software change, increasing complexity, continuing growth, and declining quality. A typical software change may contain local code change or architectural change or both. However, compared to local code changes, design impactful (or architectural) changes are involved in the wider spectrum of code components, and dependency among multiple modules/components [213] despite focusing on a single issue in such changes. As a result, comprehending their scopes and impacts are more complex to the reviewer [227, 232], and elevate the change and maintenance cost and effort across a system's lifecycle [247]. Thus, understanding and updating a system's architecture in elegant ways is crucial for development and maintenance [82]. In this regard, architectural change management process helps predict what must be changed, facilitates context for reasoning about, specifying, and implementing change, and preserves consistency between system design and adaptive as well as evolutionary changes [189, 175].

However, for architectural change management, development teams categorize the changes based on different criteria, such as the cause of the change, the concept of concerns/features, the location of the change, the size of the code modification, or the potential impact of the change [75, 101, 67]. For example, causes of architectural changes are [246, 63]: *perfective* – indicates new requirements and improved functionality, *corrective* – addresses flaws, *adaptive* – occurs for new environment or for imposing new policies, and *preventative* – indicates restructuring or redesigning the system. Different categories trigger different strategies for change management. Overall, architectural change categorization is important for change characterisation, design change document generation, design decision recovery, automatic tagging of the change revisions, development and maintenance tasks planning, and so on.

Textual descriptions written within the commit messages are an excellent source of understanding the developer's intention for a change task. Keywords or topic identification within the message is the most widely used approach for software development history (or commits, issues and bug reports) [57]. Latent semantic analysis and topic modelling methods are being widely employed for texts and document classification in various software engineering fields. A number of studies attempted to classify general software changes into three categories (perfective, adaptive and corrective) [80, 255]. In this study, we investigate how traditional text classification (TC) algorithms employed in usual commits are feasible for classifying four types of architectural changes.

Latent semantic analysis and topic modeling methods are being widely employed for texts and document classification in various software engineering fields. However, a few studies attempted to classify general software changes into three categories. Fu et al. [80] developed a semi-supervised LDA model from predefined keywords representative of those changes. Yan et al. [255] proposed a probabilistic semantic model based technique called DPLSA using discriminative keywords of the three categorical changes. On the other hand, architectural changes are divided into four categories [63]. Moreover, defined keywords for the three usual change classes are not enough to categorize the four architectural change classes. While architectural change is a subset of general changes, no automatic techniques have been studied to categorize the architectural changes. In our study, we identified a bag-of-words for four architectural change categories and employed both the topic model-based techniques [37, 202, 80] and latent-semantic analysis based techniques [255] for the automatic categorization. Thus, we explore the following research questions

RQ1. What is the feasibility of the existing TR-based (text retrieval) approach for the classification of architectural changes?

RQ2. How new feature models of TR-based approaches are promising for predicting the types of architectural changes?

To answer RQ1 and RQ2, we manually labeled the change documents with one of four types of changes, as defined in previous studies [63]: perfective, corrective, adaptive, and preventative. Finally, we experimented with various latent semantic analysis and topic modeling techniques [255, 202, 80] to automatically classify the documents into one of the four groups using our extracted natural language model. Although precision is around 72% for only one case for architectural change categorization, the outcome for the random dataset is insignificant (around 45% F1 score for the best model). Notably, despite the unpromising outcome in general, our approach performs better for the architectural changes than the automated general software change analysis technique proposed by Yan et al. [255], which, unlike our approach, did not identify preventative architectural changes; they only identified three types of changes. Identification of preventive change further supports practitioners in improving software maintenance. However, to understand the unexpected outcome, we utilized the subset of annotated data by Ding et al. [63] as a standard to follow. What we have found is that there are many samples where natural language contains *Tacit* expression (only deeply rooted in owner's head) and does not directly represent the categorical architectural changes as described by previous works [255, 80] for the general changes. In summary, this study focused on the following directions:

- A benchmark dataset by manually annotating the architectural changes into four categories.
- A list of categorical keywords which have a probabilistic relationship with the four architectural changes.
- A feasibility analysis of a number of text-categorization and classification techniques for architectural change classification.

4.2 Background

Software architecture is defined as *the fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution* [107]. Here, elements are program entities (e.g., modules, packages, classes, objects) at the abstract level. Recent studies defined many architectural properties as quality attributes through empirical studies. Those are: Coupling [49] - interconnectivity between two components, Cohesion [49] - intraconnectivity among sub-components within a group, Dependency [191] - how components are tight together in terms of calling actions/services, Independence [191] - module-wise independent searching and replacement freely , Design Hierarchy [251] - design decisions on hierarchical layers of a software, Design Pattern [78] - relationships and interactions between classes or objects, and other object-oriented properties. These are the important metrics or information that the developers and the experts specify in their commit and discussion messages.

Architectural change operations: Change classification schemes are used to measure the impact and risks (qualitatively) inter-related with making specific types of changes. Williams and Carver [246] present an empirical study for extracting an architectural change schema by analyzing 130 research works. Some other studies [14, 74] also provide information about architectural change activities. In summary, architectural change represents the following actions.

Adding components: adding a package, module, inheritance with the codebase which changes one of the properties mentioned in the earlier section among internal components.

Removing components: deleting package, module, inheritance with affecting interdependencies.

Splitting components: splitting means divide or partition a package or module into one or more packages or modules.

Merging components: aggregating packages, modules, classes into a single one with the changing of one of the previously discussed properties.

Relocating: altering dependency and hierarchy due to change in position.

Cyclic Dependency: introducing new cross-dependencies among existing components such as importing a class or inheriting a class.

Changing runtime connection and configuration: updating run-time architecture [246].

The architectural changes may happen for various reasons, and existing studies categorize the changes [246, 63] into four groups: (1) *Perfective* changes occur to adjust new behaviour or requirements changes. Also, these changes are aimed at improving processing efficiency and enhancing the performance of the software. (2) *Corrective* changes fix defects (e.g., processing failures) in the system. (3) *Adaptive* changes direct towards a new environment or platform or accommodating new standards. (4) *Preventative* changes mainly focus on future maintenance by restructuring or re-engineering the system. In our work, we adopted these four categories of the architectural changes.

4.2.1 Documenting Software Architecture:

According to Kazman et al. [115], documenting software architecture systematically has emergent benefits for both the members of the project and the outside practitioners. It is valuable to understand how software architecture is described among practitioners. Ding et al. [64] conducted an empirical study on how to describe software architecture based on the suggestions of IEEE 1471-2000 [11] standard including ten main architecture elements: system, mission, environment, stakeholder, concern, model, rationale, view, viewpoint, and library viewpoint. According to their study, a software architecture is mostly presented in the following ways, *Natural Language*: Information of architectural document elements can be expressed with natural language which may use specific technical or domain terms to interpret various aspects of the architectural elements, *Diagrams*, *UML (Unified modeling language)*, *ADL (Architecture description language)*. Among the ways mentioned above, natural language is the most significant [64, 115] medium to represent a system architecture (at least for open source projects). Researchers [64] observe that among 108 open source projects, 88.9% describe the system architecture using natural language. Information of architectural document elements can be expressed with natural language which may use specific technical or domain terms to interpret various aspects of the architectural details. IEEE [11] has set up a standard for describing various aspects of software architecture.

Table 4.1: Data samples of the candidate projects for our study (we consider a portion; since manual inspection of all messages takes significant time). GIT = github.com.

Project	**Version Tag**	**Total**	**Samples**	**Data source**
Galaxy	16.04	1392	530	GIT/galaxyproject
Hibernate(ORM)	4.0.0,4.3.0, 4.3.1	2000	310	GIT/hibernate
ImageJ	2.0.0beta	1155	300	GIT/imagej
iPlant (DE)	1.9.0,1.9.5, 2.6.0	513	250	GIT/cyverse-archive
ArgoUML	Mailing list	∼	37	argouml.tigris.org

For example, some of the texts used in the recommendation for describing viewpoints of an architecture are *"...An architectural style, then, defines a family of such systems regarding a pattern of structural organization"*, *"..physical communications interconnects and their layering among system components"*, and so on.

Information Mining from Message Description: A commit message, discussion, or review comment is a short description written by the developers or experts, and it is unstructured free format text in most of the cases. Despite this, there are many salient words which are used by the writers to describe their intentions relevant to various contexts. The words and mapping their relationship with the contexts is the primary task for mining significant information. This concept has been explored by the researchers with the text-retrieval and topic modelling methodologies [36, 37] to bug-report detection, code review generation, release note generation, and software change analysis. An architectural change message [63] reveals fixing or updating the architecture due to defect or adding new behaviors. Mapping architectural words with this concept facilitates to employ text mining techniques for the change analysis. In summary, textual documents are an excellent source of extracting architectural information. In our study, we attempt to utilize this knowledge-base to develop a natural language model.

4.3 Dataset Collection and Study Design

We have collected the codebase, commit history, e-mailing lists, and releases of the five popular open-source projects. Two of the projects, ArgoUML and Hibernate were used in an earlier study [63]. Another three, Galaxy, iPlant DE, and ImageJ have been widely used for several years for Plant Phenotyping and Genotyping analysis. Galaxy and iPlant DE are large cloud systems integrating many advanced modules for handling massive and various types of data, and architecture change management is critical for them. Each of the projects contains thousands of commits and many releases. The description of the collected project's artefacts is presented in Table 6.1.

In this study, we mainly focused on commits and selected overall 1350 commit messages containing more than two words. We randomly picked the samples from the versions (which have gone through the highest number of packages/modules changes) of the Galaxy, Hibernate (ORM), iPlant Collaborative, and ImageJ projects respectively. Moreover, since the closest work [63] experimented with mailing threads, we added 40 mailing threads from Hibernate and 37 from ArgoUML project (we collected ArgoUML data from Ding et al. [63] since during our data collection and experiment time the history server was down) in our experiment. We manually annotated these samples (took around 60 working hours per person) which have gone through different architectural change operations mentioned in Section 4.2 by analyzing the changes in the source code revisions. Please note that we did not consider a commit as architectural change if the component changes are only in the web-page (HTML, CSS, JavaScipt) design and test-code modules. We prepared 927 training samples and 500 test samples primarily. In the subsequent sections, we discuss our methods and findings in detail.

4.4 Classification of Architectural Change Messages

In this section, we discuss in detail about the feasibility of the classification tasks into four groups with the state-of-the-art natural language models. For measuring the impact and risks associated with making certain types of changes, experts [247] have been using change classification schemes. We manually analyze and annotate the architectural change messages and discussion based on the 37 provided discussion threads and knowledge-base by Ding et al. We are able to categorize 362 samples among 423 architectural changes (within the 1427 set). Among them, *preventive* samples are 156, *adaptive* 26, *corrective* 69, and *perfective* are 111. These also confirm the findings that preventive is the most common change type (with different datasets). We could not categorize other samples because of little and ambiguous texts (e.g., *Creating test for desktop notifications*). We prepare the 241 (from the 927 set as discussed in Section 4.3) samples as the training set and the rest of the 121 (from the 500 set) samples as the test set. Although various text classification techniques have been experimented, among them, adaptive classifiers, topic model and probabilistic methods [141, 203] out-perform other machine learning techniques. Therefore, we adopted the following widely used techniques: DPLSA [255], L-LDA [202], and semi-LDA [80] to experiment with the auto-classification of the architectural changes. As no specific features are defined for the four categories beforehand, these classifiers are well suited for our empirical study, and we expect close proximity to the outcome reported by Yan et al. [255] for the general change classification.

4.4.1 Labeled LDA (L-LDA):

LDA [37] is an unsupervised generative model that classifies the terms contained in the corpus of documents into groups which are defined as topics. LDA is being used in software repository mining extensively from various perspectives. In LDA, prior values α and β are used to compute topic distribution θ. To mitigate the inference problem of LDA [37], Ramage et al. [202] proposed L-LDA restricting the topics into predefined keywords. Labeled LDA [202] is the combination of supervised LDA and Multinomial Naive Bayes. Unique label assignment is essential during the training process of L-LDA. Inference for new messages is made from the precomputed prior probability ϕ.

4.4.2 Semi-supervised LDA (Semi-LDA):

In the semi-supervised LDA model [80], the topic predicting parameters are calculated from the training samples of the documents, hence changing the unsupervised training process into a semi-supervised fashion. The training samples influence the generation process of the topic words since they increase the co-occurrence frequency of the keywords associated with the same category. The inference is made by calculating cosine similarity of a new message from the pre-computed topic-words assignment. Semi-LDA has the unique capability to infer ambiguous category into an anonymous one using multiple iterations thus separating from the labelled class. L-LDA and semi-LDA are almost same except semi-LDA uses multiple iterations to increase the classification accuracy.

4.4.3 Discriminative Probabilistic latent semantic analysis (DPLSA):

Probabilistic latent semantic analysis (PLSA) [149] is used for topic modelling by generating a probabilistic model. In PLSA, word-topic distribution ϕ and topic-document distribution θ is initialized by the random values. Next, these probabilistic parameters are maximized through Expectation-Maximization (EM) algorithm. However, PLSA has the limitation on making inference about new unseen documents. Yan et al.[255] upgraded the PLSA [149] in order to overcome the limitation on making inference using discriminative keywords (hence labelling into classes). They develop a bag of keywords for categorizing software changes into three categories mentioned earlier. DPLSA used supervised learning with pre-computed values from the training samples, whereas PLSA was unsupervised for generating K top topics. A new text message is inferred for the intended class from the θ values computed during the EM step. Although the previous two methods defined the keywords for three software change types, these keywords should be for four architectural changes, and many of the keywords for general changes might not represent architectural changes. Therefore, we determined the top distinguishing words for the four categories of change from the labelled samples (training set). We use the probability calculation defined by Yan et al.[255]. The probability of the words are calculated as follows:

$$\phi_w^{(j)} = \frac{c(w, D^{(j)})}{c(D^{(j)})}, \quad \theta_j^{(d)} = \frac{\sum\limits_{w \epsilon T} c(w,d) P(T_{d,w} = j)}{\sum\limits_{j'} \sum\limits_{w \epsilon T} c(w,d) P(T_{d,w} = j')}$$

where $c(w, D^{(j)})$ is the number of times word w occurs in the collection $D^{(j)}$, $c(D^{(j)})$ is the number of times all words in the key-terms T occur in the collection $D^{(j)}$. These keywords are also used for the other two techniques. The top 20 keywords in the training samples are presented in Table 4.2. We observed that many keywords defined by previous studies are not present in the top 20 lists for architectural change categories.

4.5 Experimental Outcome

We implemented the models with various open source Python libraries. The artifacts are available in [9]. LLDA implementation is based on *nltk* based *lda* [3] and parameters are, α = 0.1, β=0.1; SemiLDA is based

Table 4.2: Top keywords for various categories of architecture changes for 241 samples. Words are stemmed with PorterStemmer.

Perfective	Corrective	Adaptive	Preventive
remove move include template rename clean unused amend properti applic request contribut multipl schema easi resourc select popul explicitli easier	association treatment properli preliminary evolve motiv complic mediat revers engen unabl nearli strongli specifi issue bug reassign inform fix	invers trivial increasingli util priorit mode anchor explor altern quickli redefinit visitor live appreci locat achiev temporari isol postpon exactli	heavili intertwin renam separ command split clean exist restructur mani simpli embed refactor elimin automat dialect remov intern attempt represent

Table 4.3: Experimental outcome of various classification techniques compared to change model in [255].

Classifier	Metric	DPLSA[255]	LLDA[202]	SemiLDA[80]
KGC (3C)	P	**33%**	33%	20%
	F1	**42%**	37%	19%
Our method(4C)	P	34%	33%	**34%**
	F1	30%	**45%**	34%
KGC is 'keywords for general changes'; and C is categories. ∗Prior values and iterations have a minor influence on the outcome.				

on [7] with parameters α =0.5, β =0.1, and burn in iteration is 20; DPLSA is based on [4]. We tuned the hyper-parameters with different values but the outcome does not impact much. Our experimental outcome for all the test samples with DPLSA [255], LLDA [202] and semiLDA [80] is presented in Table 4.3. F1 score [89], calculated from precision and recall rate, presents the feasibility of a text classification model. At first, we experimented with the extracted keywords by Yan et al. [255], and the produced F1 score with that approach for DPLSA, LLDA, and semiLDA are respectively 42%, 37%, and 19%. For four categories of architectural changes, with our defined keywords, these rates are 30%, 45%, and 34%. The outcome is unexpected as the most promising technique by Yan et al. [255] reported the lowest F1 score as 73%. The best recall rate among the experimented algorithms with our proposed model is 70% for LLDA. However, as shown in Table 4.4, for preventive class, the precision rate is more than 70% with DPLSA (recall is 40%) for our method (while for perfective class, with keywords by Yan et al., the best precision is 48% and recall is 48%). With both of the approaches, the classification rate of corrective and adaptive changes are poor. However, DPLSA performs the best in precision due to: (i) iterations for inference, and (ii) consideration of expectation maximization during the training step. The experimental outcome reveals that with our extracted keywords for architectural changes, the precision result (DPLSA) is promising only with the preventive class (overall 45% F1) than with the previously extracted keywords. Here, the precision rate is important, as low false positive reduces efforts. However, both the precision and recall rate is inconsistent for all the models which undermine the feasibility of those models.

Nonetheless, to figure out the reasons behind the unexpected performance of popular text-classifications

Table 4.4: Precision for individual categories with our approach using the best model. Recall rate is up to 50%. Change model in [255] only does better for Perfective (48%).

Perfective	**Corrective**	**Adaptive**	**Preventive**
36% (LLDA)	30% (SemiLDA)	17% (DPLSA)	72% (DPLSA)

methods, we further investigated the labelled samples. We found that there are a few distinguishable keywords among them which perform poorly if we apply the text-classification techniques on the test set. Also, the keywords for general change description are not sharply distinguishable for three (excluding preventive) categories of architectural changes. The Adaptive change category shows the worst performance (less than 20% accuracy) due to a lack of distinguishable text properties. Therefore, from Tables 4.3 and 4.4, it is deducible that no single algorithm can be selected as a more feasible one (it answers our research question **RQ4**). However, we further analyzed the classified samples to determine natural language properties of interest. More specifically we tried to find an answer to the question: do co-occurrence patterns exist in the samples? We utilized the subset of annotated data by Ding et al. [63] as a standard to follow. What we have found is that there are many samples where natural language contains *Tacit* expression (only in one's mind) and does not directly represent the categorical architectural changes as described by previous works [255, 80] for general changes. Furthermore, we were unable to extract significant common co-occurrence patterns of the words of the four type changes; which one represents which category is not straightforward and mostly subjective to human perception. But, as is noticed during the analysis, corrective change messages have some sort of negativity. A large collection of training samples might improve the classifications significantly.

4.6 Related Work

Textual descriptions are major sources of various architectural information. Most of the change classification methods focused on natural language processing, text retrieval, and machine learning techniques leveraging textual properties. The followings are the popular classification methods.

One of the pioneering studies for classifying the causes of code change is conducted by Mockus and Votta [168] employing direct lexical analysis (word presence and count) into *adaptive* (A_m), *perfective* (P_m), *corrective* (C), *inspective* (I), and not sure categories. However, adaptive and perfective concepts are different than Swanson [226] classes (as explained in Section 2.3.1). The authors first extracted discriminative keywords from the description texts of 20 modification requests for each of the classes. Then, this word list is employed for classifying where the presence of a keyword is the indicator of a type, but if more than one type of keyword is present, the type with the most number of keywords in the description is considered. However, if that measure also has the same value, then it prioritizes perfective and then corrective types. One interesting finding of this study is that the more restructuring tasks (they call them the perfective tasks) are linked to the longevity of the commercial products.

Hassan et al. [94] followed the work of Mockus and Votta. Their model classifies bug fix (BF), feature introduction (FI), or general maintenance (GM) types based on the keywords similar to Mockus and Votta [168]. During the classification process, it first prioritizes the BF class because the keywords for this class are less ambiguous than those of other classes. Again, the category which can not be classified is separated into not sure (NS) types; thus, the performance is not indicative of the actual outcome. One key finding of this study is that there are ambiguities among the senior and less-experienced developers on how they determine a change type.

Hindle et al. [101] explored the classification of large change commits that impact more than one code file and other major concerns such as licensing documents. These classes are as follows: *implementation* – new requirements, *maintenance* – maintenance activities, *module management* – the way the files are named and organized into modules, *legal* – related to the license or authorship of the system, *quality (non-functional) source changes* – source code that did not affect the functionality of the software, *source control system change* – the manner the Source Control System is used by the software project such as branching and *meta-program* – files required by the software that are not source code. Many of these classes are the extension of Swanson [226] maintenance categories. However, since their process is manual, it is difficult to employ this process for large commits and other projects. Following the manual classification, Hindle et al. propose an auto classifier [100] using machine learning techniques to categorize the seven large commits. They combined three types of features: word distribution, author information, and module or file type. Word distribution is calculated using Bayesian type learning on word frequency. Machine learning techniques employed in their studies are J48 (decision tree), Naive Bayes, and so on. However, the best model achieved more than 50% accuracy. As the authors report, the manual annotation for creating a ground truth remains problematic. Overall, the most valuable outcome of their study is a set of keywords that are later experimented with, modified, and extended by various studies. Finally, many of the large commits are architectural.

Hattori and Lanza [95] proposed a keyword-based technique for classifying commits into *forward engineering, reengineering, corrective engineering, and management activities.* Here, *forward engineering* activities are related to the integration of new features and implementation of new requirements. *Reengineering* activities are related to refactoring, redesign, and other actions to enhance the quality of the code without adding a new function. *Corrective engineering* indicates defects, errors, and bugs in the software. *Management activities* are unrelated to codification, such as formatting code and cleaning up and updating documentation. In this study, first, they group the commits into four based on the number of files affected: tiny, small, medium, and large. For that, classification is experimented with these groups in separate cases. After that, the authors curated a list of keywords for each of the categories, and then they predicted a commit based on the keyword appearance. However, the first keyword in the commit description indicates the relevant change category.

Mauczka et al. [160] presented a word-dictionary based technique to categorize the commit message into Swanson [226] maintenance categories. They adopted the classifiers presented by Hassan et al. [94] as bug fixing, feature introduction, or general maintenance changes based on keywords. For that purpose, the authors extracted a set of words for each of the categories and developed a plugin called *Subcat*. Furthermore, they created a benchmark dataset annotated by the real-world developers [159]. Additionally, they merged keywords from the early study of Hindle et al. [101]. However, the most important finding of this study is that there were ambiguities among developers regarding the explanation of the adaptive and perfective categories.

Ding et al. [63] explored the four causes (adaptive, perfective, corrective, and preventive) of architectural changes in open-source software. They analyze the communication and discussion messages of the developers based on the explanation of Williams and Craver [246]. However, their categorization technique is manual.

Fu et al. [80] proposed a semiLDA (semi-supervised) technique by extending LLDA [202] for classifying the Swanson [226] maintenance categories. The main difference between LDA and semi-supervised LDA is the generation of the topic. In the semi-supervised LDA model, they added signifier documents to change the unsupervised training process into a semi-supervised fashion. The signifier documents can influence the generation process of the words because they can increase the co-occurrence frequency of the keywords which belong to the same category used by human supervision. However, they used keywords extracted by Mauczka et al. [160]. Their test set was 50% samples, and they did not employ cross-fold validation. Among the explored models, SemiLDA achieved 70% $F1$ score and can determine 80% change message, but the proposed method can not decide 20% of the messages. Moreover, too brief messages are pruned during the dataset creation (but did not mention the criteria of being too brief).

Yan et al. [255] presented a Discriminative Probability Latent Semantic Analysis (DPLSA) technique to classify the Swanson classes. This model initializes the word distributions for different topics using labelled samples to categorize the change messages. Their proposed technique creates a one-to-one mapping between the extracted topics and the change classes. However, authors utilized the keywords extracted by Mauczka et al., and claimed that the multi-category (tangled) classification is improved compared to LLDA, SemiLDA and Naive Bayes. Moreover, the study also included the not sure (NS) category like Hassan et al. Yet, they have not tested the DPLSA classifier with the standard cross-fold validation to reduce the over-fitting problem; only the model was tested with a separate set. Thus, DPLSA's performance might not be conclusive enough.

4.7 Threats to validity

We identified two kinds of threats (internal and external) to validity for our study discussed as follows.

Internal validity: Natural language analysis is subjected to human bias. However, during manual analysis, we carefully followed the knowledge-base of previously published studies to annotate our dataset (along with codebase changes). To reduce the human bias, we cross-verified the annotated samples in between first two authors separately and resolved some disagreements by a discussion based on provided 37 samples by Ding et al. [63], and information in Williams and Carver [246]. Additionally, we experimented with the curated dataset of Paixao et al. [194] for the detection, and partial dataset provided by Ding et al. [63] for the categorization. The architectural message detection result is better for the dataset of Paixao et al. as well (recall rate is up to 96% despite the projects are from different and unseen contexts). Moreover, the discussion documents contain a lot of noisy data which are not pure natural language texts (e.g., $< xyz@mail.com >$, $Job.cancel()$, $Signed - off - by$:, and so on), and to mitigate the threat we remove the words if they do not contain the pure ASCII letter. This filtering increases the precision rate by around 1.5% on an average. To check over-fitting problem of the change classification techniques, we employed 4-fold cross-validation. Thus, we found the better outcome in a tiny portion (as key-words extraction is essential from the training samples, it is irrelevant here).

External validity: In a study such as this, the generalization of the experimental results to various domains may pose an external threat. However, the projects we selected cover diverse domains and technologies (C/C++, Java, Python, Go) and the experimental outcome shows consistent trends for most of the cases.

4.8 Conclusion

In this section, we have presented an exploratory study on the architectural change categorization using various kinds of natural language messages including commits, emails and review comments of various popular open-source projects. Our study is the first to specifically categorize the architectural changes into four predefined types (perfective, preventive, corrective, and adaptive). Here we experimented with the popular text-classification techniques for the classification of architectural changes and found about 62% precision for the best case, which is 14% better than the existing software change analysis approach [255]. In general, for the classification of architectural changes, the outcome for the random dataset is unpromising (around 45% F1 score for the best model). This is due to overlapping keywords in the descriptions of different types of commit messages and tacit information. In future, we will extend our study by including more software projects and source code properties, which will allow us to consider more intuitive textual properties.

Our experiments show that the traditional discriminating sets of keywords are not promising for architectural change categorization. Therefore, more intuitive properties should be explored for efficient change classification. In the subsequent study (i.e., Chapter 5), we will explore the concept theme of a text and a more promising classifier leveraging that theme.

Chapter 5

Architectural Change Classification using Concept Tokens

In the previous study (i.e., Chapter 4), we observe that many keywords are overlapped among the change categories. Additionally, tacit variation of intention cannot be captured with the discriminating sets of keywords. To eradicate this limitation, we explore intuitive text properties for architectural change classification. In this study, we define and extract concepts [200] from the commit messages of all the annotated samples that express the corresponding intention of a task. Even the top words (such as support) among the defined concepts contain many overlapping words. However, we have found some patterns in many samples for expressing different concepts when these terms co-occurred with other tokens, which are stop words, code elements, and API, library, or framework name. To handle this, we train a model by assigning weights to the concept tokens using a specialized normalized frequency model from a set of pre-classified commits into four change categories. This is motivated by the core idea of how the model for a word's sentiment is generated [21]. These weights represent the strengths while present within the concepts of the categories. Finally, the trained model produces a collection of unique concept tokens, which are then used to predict the change message to an expected category.

The chapter continues as follows. Section 5.1 presents a brief overview of our study. In Section 8.2.1 we discuss the background of architectural change detection and classification. Section 5.3 describes our dataset creation process. Section 5.4 explains the challenges of change classification. Our proposed classifier is presented in Section 5.5. Section 5.6 reports our experimental outcome and Section 5.7 discusses threats to validity. Section 5.8 discusses related studies and Section 5.9 concludes the chapter with future direction.

5.1 Introduction

Software architecture is concerned with the partitioning of a software system into parts, with a specific set of relations among the parts [84]. A meaningful architectural document helps reduce the cognitive load and maintenance activities of the software development team [107]. Moreover, appropriate architectural formulation is becoming more critical to circumvent software bloat, scalability, and security backdoors [87]. However, elements of architecture can be changed [130] continuously as code components of a software system changes to support continuous development and maintenance [172] such as adding new features, restructuring the design models, and fixing flaws. Architectural change can affect many aspects of a software system and, for this, change analysis is a crucial task. Development team can group architectural changes leveraging change classification process based on the cause of the change, type of change, location of the change, the size of the code modification, and impact of change [246, 101]. For example, four major causes of architectural changes have been defined explicitly in the literature [246, 63, 172]: (i) *perfective* – adjusting new behaviour, (ii) *preventive* – prevent bad design, (iii) *corrective* – correct discovered problems, and (iv) *adaptive* – adapting to new platform.

Grouping causes of change is beneficial for post-release analyses, where design change activities are not explicitly annotated [63]. Change classification is also required for composing a developer's profile, building a balanced team, and handling anomalies in the development process [136]. Furthermore, code review process involving architectural change is complex than local or atomic change [240], which is dependent on determining change type. Moreover, an automated technique can be employed to produce design documentation for every release recording types of structural changes happened and associated components [115]. Automated architectural change classification technique [246, 194] can be used to develop strategies for implementing a system change, support continuous architecture, augment DevOps and Model-Driven Engineering tools [34, 91, 87]. Existing active software projects (even if we consider a tiny portion of the 100 million repositories in GitHub [2]) could immediately benefit if a structural change classification technique is available to help develop an architectural versioning schema.

However, while architectural change can be identified from source code change, identifying the design decision, reason, and categories of changes requires analyzing the development team's intention. The intention can be extracted from textual description of the developer's tasks and discussions [34, 172]. Literature has focused on classifying typical software changes, architectural design concerns and design solutions [254, 166, 100]. Yet, supporting architectural change classification is still in its infancy [172, 87], perhaps, due to lack of benchmark data and requirements of laborious human analysis. Nevertheless, a few of the studies explored for both manual [63, 194] and semi-automated [172] techniques for classifying architectural changes. In these studies, a small collection of samples is being experimented where challenges are not identified properly, which leads to developing infeasible models. Besides, the traditional text classification techniques [37, 255] might not handle the scenario when keywords are present among multiple concepts within the description of a task.

To address the shortcomings, we design a benchmark data and propose a text classifier called ArchiNet for architectural change classification. In particular, we focus on the two research questions:

RQ1: How can source code properties that are independent of the description of project activities classify the rationale of architectural changes?, and

RQ2: How can we improve text classification to predict the rationale of architectural changes leveraging commit descriptions?

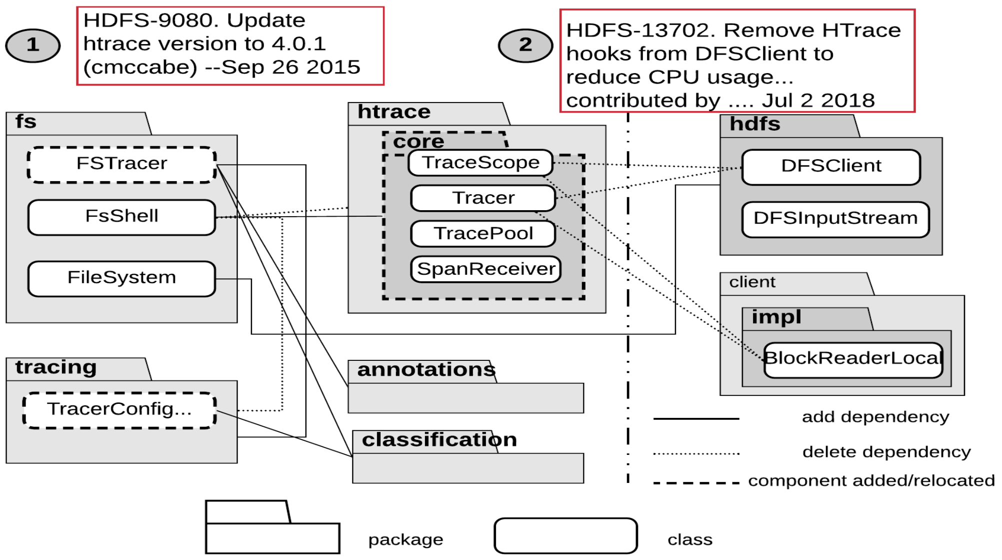


Figure 5.1: Two commits of *Hadoop* where new components are added, dependency added and deleted.

To answer RQ1 and RQ2, we collect around 1,133 architectural change instances from 5K commits of five popular projects (shown in Table 5.1). After extensive analysis of the created dataset, we have successfully identified the challenges of categorizing the architectural changes both from the source code and the texts. One of the challenges is that typical operations in the source code do not have a significant number of distinguishing patterns in various changes, and classification performance is not promising (F1 score is 33%). A major challenge in the commit description is that multiple concepts are presented, whereas only one or two concepts indicate the intention. Furthermore, many words are common for expressing the reasons for changes, such as keyword *update* is used to describe both *perfective* and *adaptive* changes. Such a phenomenon is not acute in many other text classification tasks [128]. All things considered, we propose a new technique for classifying the changes from the text where trained keywords from concept analysis of different changes play a crucial role. The training process of our proposed technique is different from the traditional NLP training process. For training, we first define the relevant concepts (contextual occurrence of words and tokens such as $\{update, API, version\}$ indicate *adaptive* change more confidently) within each sample. Next, all tokens' weights appeared within all the concepts for a relevant change class are calculated. These weights are distributed among all of the classes leveraging a statistical model. Thus, our technique does not consider all the words within a description. Finally, a given commit is predicted to one of the four classes using a probability model from the trained database. Experimental outcome of our classifier with different datasets shows that the F1 score is around 70% and promising compared to the competing techniques (including deep learning).

5.2 Background

Architectural Change Instance: Studying typical changes from version control systems does not require a change detection strategy as it provides differences. However, architectural change detection [130, 150], even from the version control system (*diff*), is challenging. Some of the widely used change metrics are DSM [24], MoJo [245], MoJoFM [245], graph kernel structure [181], A2A [130], C2C [150] and include-symbol dependencies [150]. These metrics are calculated based on the following operations: *adding components, removing components, replacing components, splitting components, merging components, relocating, module dependency graph, and usage dependency.* We focus on intermediate-level architecture for collecting change samples and employ $A2A$ and $include + symbol$ dependency metrics for change detection. A2A considers component addition, removal and moves; $include + symbol$ dependency considers including/removing header file, program file, importing class, and importing interface. Causes for architectural changes are grouped as follows.

Adaptive (A) change: This change would be a reflection [246, 143, 226] of system portability, adapting to a new platform such as commit① in Fig. 5.1. Adaptive change also happen for imposing new organisational and governmental policies.

Corrective (C) change: A corrective change is the reactive modification of a software product performed after deployment to correct discovered problems [246]. Specifically, this change refers to defect repair, and the errors in specification, design and implementation.

Preventive (PV) change: Preventive change [226, 246] refers to actionable means to prevent, retard, or remediate code decay. In other meanings, preventive changes happen to improve file structure or to reduce dependencies between software modules and components may later impact quality attributes such as understandability, modifiability, and complexity.

Perfective (PF) change: Perfective changes are the most common and inherent in development activities. This change mainly focuses on adjusting new behaviour or requirements changes [226]. Also, these changes are aimed at improving processing efficiency and enhancing the performance of the software (such as commit② in Fig. 5.1) that is both functional and non-functional optimizations.

This classification is essential to deal with various challenges (discussed in the Introduction) since different types of change influence them in different ways. Among the change categories, preventive and corrective changes are directly related to major design debt management. A few of the change types in the two commits in Hadoop is shown in Fig. 5.1. Commit descriptions simply express their intentions. Commit① is an adaptive change in 2015 and commit② is a perfective change in 2018. It is noticeable from commit② that a dependency change between two components ($htrace$ and $hdfs$) increases performance by reducing CPU usage, which is also an architectural change. Both of the changes happen almost a decade later of the first release of Hadoop. Components of the $htrace$ module are at the center of these two changes (commit① and commit②) although the second change occurred after three years of the occurrence of the first change.

Table 5.1: Candidate projects for our study (in inspection time).

Project	**All**	**Archi.**	**Domain**	**Source**
Hadoop	22631	266	Distributed Computing	gt/apache/hadoop
HibernateORM	9811	261	Object/Relational Mapping	gt/hiber../hibernate-orm
LinuxTools	10630	265	C&C++ IDE for Linux	gt/eclipse/linuxtools
JavaClient	1477	136	Java bind for Appium Tests	gt/appium/java-client
JVMcouchbase	914	205	JVM core for Couchbase Server	gt/couchbase/couchbase- jvm-core
Total	45463	1133		

Archi: architectural changes in selected 1K commits; gt: github.com

5.3 Dataset Preparation

A few of the studies [63, 172] created datasets for architectural change classification from the development history. The recent dataset created by Mondal et al. [172] consists of 362 samples of four projects (26 of them are adaptive). This dataset might be insufficient for detecting some of the text classification challenges such as various concept tokens including code elements and framework name. Created dataset by Pixao et al. [194] contains architectural change only for new features and other categories are not annotated (recently they updated their dataset with fixing issues but not specifically annotated to four groups discussed widely in the literature [246, 63, 172]). Another dataset is constructed by Ding et al. [63] which is not publicly available (thanks to the authors for providing us 37 samples). Therefore, we prepare a new dataset containing a large collection of commits (shown in Table 5.1).

5.3.1 Architectural Change Commits Filtering

We selected five open source projects that are widely experimented in literature for software change and architectural analysis [166, 130, 115, 194], ensuring a diversity of domains. We also ensure that the projects are in active development for at least several years. The selected projects are: Hadoop, Hibernate ORM, Linux Tools, Java Client, and Couchbase JVM Core have 45,463 commits which are infeasible to analyze manually. Since determining and categorising architectural change instances require huge human efforts, in our dataset creation process, we restrict primary selection of commit samples into 5K. We randomly choose 1K commits from each of the projects containing more than two words in the messages excluding stop words, non-alpha words (that contains non-letters such as *issue-110*) along with the words having *Change-Id:* or *Signed-off-by:* and so on as shown in Fig. 5.1.

We separate the architectural change samples from the primary collection (around 5K) if *A2A* and

Table 5.2: Training and test samples in the golden set.

Split	Perfective	Corrective	Preventive	Adaptive	Total
Train	425	122	185	68	800
Test	173	49	73	39	333
Total	598	171	258	107	1133

include + *symbol* dependency metrics are changed. However, as suggested by the literature [156], we do not consider system library usage from native (Java, Python) framework for dependency change. In this way, we get around 1133 samples (distribution of them is shown in Table 5.1) as architecturally changed commits.

5.3.2 Architectural Change Category Annotation

In the next step of our study, we manually label those samples by two authors independently into one of the four categories described in the existing studies [172, 63]. There are ambiguities in some of the descriptions of four types of changes. We review most of the relevant papers referred by [246, 63, 172] for more explanation to resolve the ambiguity (details are discussed in Section 8.2.1).

Our manual annotation process has two iterations. In the first iteration, two of the authors having three years of average software industry experience, categorized the samples separately. In this step, we get many samples mismatched in annotation. In the second iteration, we recheck the mismatch samples and resolve the disagreements by discussion. Total number of samples in each of the annotated categories from the candidate projects is shown in Table 5.2. The finalization of our dataset took one month of two person-hours, indicating that manual change analysis is expensive. In the next section, we investigate the automatic change classification challenges.

5.4 Change Classification Challenges

For examining the challenges of classification, we divide the samples into two parts: training and test sets. As empirical study [121] suggests that 30% test samples are ideal for real data, we split around 70% of the architectural commits for training purposes and around 30% for testing purposes with random sampling. However, we could not extract meaningful concepts (Section 5.4.2) for some of the samples due to lack of information, and skip those during the training and testing phases. Distribution of change types in the train and test sets are shown in Table 5.2. Both the train and test sets contain the conflicted samples accordingly.

5.4.1 Classification from Source Code

First, we explore classification options leveraging source code operations. Yamauchi et al. [254] cluster the change commits based on source code modifications: identifiers, method name, and class name into as many groups dependent on component-requirement relations. Their technique cannot be used for a fixed number of classes. The clustering basically groups the commits into related components attached to an implemented functional requirement, not the reason for changes. Therefore, we explore a technique utilizing the distribution of change operations of the architectural components (static). We examine the abstract operations (O) occurred in the source code of a commit: *import added* or *deleted*, *class file added* or *deleted*, *file* or *package rename*, and *function added* or *deleted* as properties of change classification since they are universal and independent of project context.

Considering these properties, we design a classifier using $C_i(w_O)$ in (5.1) as described in Section 5.5 to evaluate how significant the classification is using these operations as metrics and has the following outcome with 10 fold cross-validation. The best F1 score (among different combinations of the operation types) for perfective, preventive, corrective, adaptive, and all combined are 0.33, 0.53, 0.08, 0.13, and 0.33 respectively. F1 for the corrective and adaptive classes are negligible. In summary, source code properties are not promising for architectural change classification; this answers our RQ1. In the next section, we explore existing change classification techniques from commits messages.

5.4.2 Change Classification from Text

Explored Models

Next, we examine the explored models of Mondal et al. [172], where the best model produces 39% F1 scores with our dataset. Following these approaches, we also develop a discriminating feature selection (DFS) model from the distribution of words in our training dataset. Similar to Mondal et al., our DFS model has many common keywords in the top list. Considering such overlapping of keywords, existing techniques based on the DFS model discussed in DPLSA [255], LLDA [202], and SemiLDA [80] predict more false positives since such a model also considers the words that might be irrelevant to the original intentions. With our new collection, the best DFS model produces 46% F1 score with precision 45.6% which is similar to the outcome of the best method in [172]. Our DFS model for the dataset in [172] produces an F1 score of 20% that is significantly lower than the previous model. In summary, the DFS models are not promising and possibly biased to the project contexts. Therefore, we focus on a more advanced classifier identifying the challenges within the textual descriptions. We discuss classifiers from traditional machine learning and neural word embedding models in Section 5.6.

Concept Analysis

As we have a large number of samples, we are able to identify the specific challenges within the message description. One of the significant challenges present in many commit messages is developers express more than one concept (*contextual occurrence of words*) for a single intention. An N-gram model might capture continuous sequences of n-words involved in such concepts within a sentence [220, 224]. However, in multiple iterations of our inspection, we find that concept words are scattered among multiple sentences in many commit descriptions. We also prioritize such scattered words while categorizing the commits. Traditional text classification techniques do not address this particular scenario (including the n-gram model). We also attempt to determine the dominating concepts from multiple concept tokens. Lets consider the corrective change message *"adding more support for services **down..**"*; here *adding support* and *down* keywords will influence to predict a category by $TF - IDF$ [190], LLDA, SemiLDA, and DPLSA techniques. Unfortunately, *adding* and *support* keywords will measure more weight to other categories because they are present among the list of the top keywords. However, if we prioritize the *down* keyword as the dominating concept, it is more likely to be a corrective category. We annotated such keywords for the dominating categories.

In the list from Section 5.4.2, some dominating words (such as issue and leak) for this corrective category are hardly used for others. But, many samples contain negative words which are not meant faults, such as - *"...This changeset moves the responsibility of sending into the locators, which has two benefits:- **No** Node[] allocations since nothing needs to be signalled back.- The code **doesn't** need to iterate through the list again ..."* is more likely to be a preventive change despite too many negative words. This is significantly an opposite concept in the sentiment analysis [21], which would treat this as negative for such words. The existing techniques falsely classify such a description as a corrective one. However, in many samples, when the word ***not*** co-locates with the word *working*, it indicates flaws in the system. Therefore, we should not skip such keywords during concept extraction. Furthermore, some code elements are used for assuming a corrective concept such as NullPointerException and LinkedError. For the adaptive category, the dominating concept is indicated by mostly multiple words. From the example in Fig. 5.1, we notice that *update*, and $version$ form a dominating concept together where domain specific terms (such as htrace, API, library, and so on) need to be included. Again, these words are present in other categories. We have manually re-analyzed all the training samples to find such concepts containing the minimal number of words. Overall, there are ambiguities of concepts (and top keywords) among all the categories. The most ambiguities are found in the perfective category to define the related concept uniquely with the minimum number of words. While manual annotation is easier for the perfective categories, defining the dominating concept, as discussed previously, is the most difficult.

We have seen that a word might indicate different concepts with co-occurring different terms as shown in Table 5.3. In the next section, we describe our proposed solution based on this finding.

Table 5.3: Ambiguity of concepts appeared in description.

Base words	Not failure	Faults
Not	complex	
Doesn't	need	work, release
Error	message	−network, −fix
Can't		change

Symbol '−' indicates located before the base word.

5.5 Our Proposed Classifier: ArchiNet

From the empirical observations, it is evident that handling overlapping words among the descriptions is the key to develop a promising solution. We conjecture that no word should be in the distinguishing list to a single category. Instead of the logic of the previous techniques, we assign a strength of a word for each of the categories. For example, for the strength of the words presented in Table 5.4 for different concepts, if the words *add*, *support*, *down* appear within a text description, then the total value for the category C_1 is $0.52 + 0.38 + 0 = 0.90$, and the total value for the category C_2 is $0.03 + 0 + 1 = 1.03$. As $1.03 > 0.90$, the sample would be for the category C_2. For simplicity, we explain with weight addition; more complex situations (with various token strengths) are handled with a probabilistic prediction technique described in Section 5.5.3. Therefore, this gives more importance to the co-occurrence of the words *add* and *down*, and such a solution might handle the described challenges in a promising way. In our solution, the crucial point is to get the concept tokens and their weights distribution efficiently, and then predict a class confidently. We describe our proposed method in three steps.

Table 5.4: Strength of words within the concepts C_1 and C_2.

Word	Strength in C_1	Strength in C_2
add	0.52	0.03
support	0.38	0
down	0	1

5.5.1 Concepts Extraction

In this stage, we define and extract concepts from the commit messages of all the annotated samples that express the corresponding intention of a task (as discussed in Section 5.4.2). Even, the top words (such as *support*) among the defined concepts contain many overlapping words. However, we have found some patterns in many samples for expressing different concepts when these terms are co-occurred with other tokens which are stop words, code elements, and API, library or framework name. Some of the examples are discussed in earlier sections. Before training, natural words are stemmed with PorterStemmer. In the next section, we discuss our training and weight distribution process from the extracted concept tokens.

5.5.2 Training Model Generation

In this phase, we train a model by assigning weights to the concept tokens using (5.1) from a set of preclassified commits into four change categories. This is motivated by the core idea of how the model for a word's sentiment is generated [8]. These weights represent the strengths while present within the concepts of the categories. The trained model produces a collection of unique concept tokens denoted by S having weights w_i to the classes C_i:

$$C_i(w_S) \Rightarrow \bigcup_{t \epsilon S} \frac{w_i(t)}{\sum_{1}^{i} w_i(t)} \cup C_i(w_O),\ \ S = \{t_w, t_p, t_s, t_a\} \tag{5.1}$$

$$w_i(t_w) = \frac{f(t_w)}{N_i}, w_i(t_p) = \frac{\boldsymbol{F}(t_p)}{N_i}, w_i(t_s) = \frac{\boldsymbol{F}(t_s)}{N_i}, \tag{5.2}$$

$$w_i(t_a) = \frac{\boldsymbol{F}(t_a)}{N_i}, w_i(t_o) = \frac{\boldsymbol{F}(t_o)}{N_i} \quad where \quad O = \{t_o\} \tag{5.3}$$

Here, $f(t)$ frequency of a token t within the concepts S of all samples in a category N_i, and $C_i(w_O)$ is the weight of the tokens defined with source code change operation types (O) associated with S, $C_i \epsilon \{PF, PV, C, A\}$. $w_i(t)$ can be calculated by adopting various metrics such as frequency value or $tfidf$. A concept S consists of various types of tokens such as t_w is natural words without stop words, t_p is some specific stop words such as negation words, t_s is some special code elements such as *NullPointerException* and *LinkedError*, t_a is api, library or framework name, and t_o is code operation types treated as tokens; each of these types has a collection of tokens. We utilized frequency normalized sum for calculating the probability value. We calculated the frequency values differently represented by bold **F** in (5.2) and (5.3): $\boldsymbol{F}(t_p)$ considers only inclusive stop words, $\boldsymbol{F}(t_s)$ consider the issue related token parts (such as Exception and Error from the mentioned tokens) extracted from a code element using camel case parsing, $\boldsymbol{F}(t_a)$ is calculated by converting all the api names into a unique token ($AABB$ in our experiment), and $\boldsymbol{F}(t_o)$ consider one or more instances of each token in t_o as value 1 within a commit. All the values in (5.1) and (5.2) are adjusted when new concepts are defined with new trained samples. Then, we employ a classifier from these trained weights.

5.5.3 Classification

During the classification phase, the generated models (M) from the training phase (in (5.1)) are used to evaluate the probability ($P_m(C)$) that a given class C is associated with the commit m. Only the tokens identified in the concepts (S) from phase one and tokens as the source code operations (O) in Section 5.4.1 are considered from commit message and code change. The classification score is then defined as follows.

$$P_m(C) = \frac{\sum\limits_{t\epsilon(S \cup O) \cap C} M(w(t))}{\sum\limits_{t\epsilon(S \cup O), C_i} M(w(t))} \tag{5.4}$$

where the numerator is computed as the sum of the token weights $(w(t))$ of all types that are contained in C, and the denominator is the sum of the token weights for all types for all classes (C_i). The probabilistic classifier for a given commit m will assign a higher score $P_m(C)$ to class C that contains several strong tokens for concept S and operation O. However, if the probabilities $P_m(C_i)$ are same for more than one class, ArchiNet considers the class which contains the highest weighted word.

5.6 Performance Evaluation

Our created dataset contains both an architectural change set three times larger than that of [172] and a list of concept-words with strength. Our proposed classifier ArchiNet is designed to handle the overlapping words and includes various tokens discussed in Section 5.4.2 within the change description. We compare the performance of ArchiNet based on recall (R) – quantitative correctness of retrieving relevant categories; precision (P) – the rate of accuracy among the predicted samples, and the F1 score – $2PR/(P + R)$ calculated from precision and recall. We have also compared with the published dataset and classifiers [172, 255]. The performance is also compared with other promising techniques in literature [100, 166, 224, 146] for text classification. These techniques include RCNN-LSTM the state-of-the-art Deep Neural Learning (DNL), Naive Bayes (NB), Bag-of-words (BoW) model, Decision Tree (DT), Random Forest (RF), DPLSA, LLDA, and SemiLDA. Our training model is significantly faster than RF and DNL, but we will not discuss time complexity since it is less critical if a model is built once for application. We evaluated the performance of ArchiNet in the following four phases.

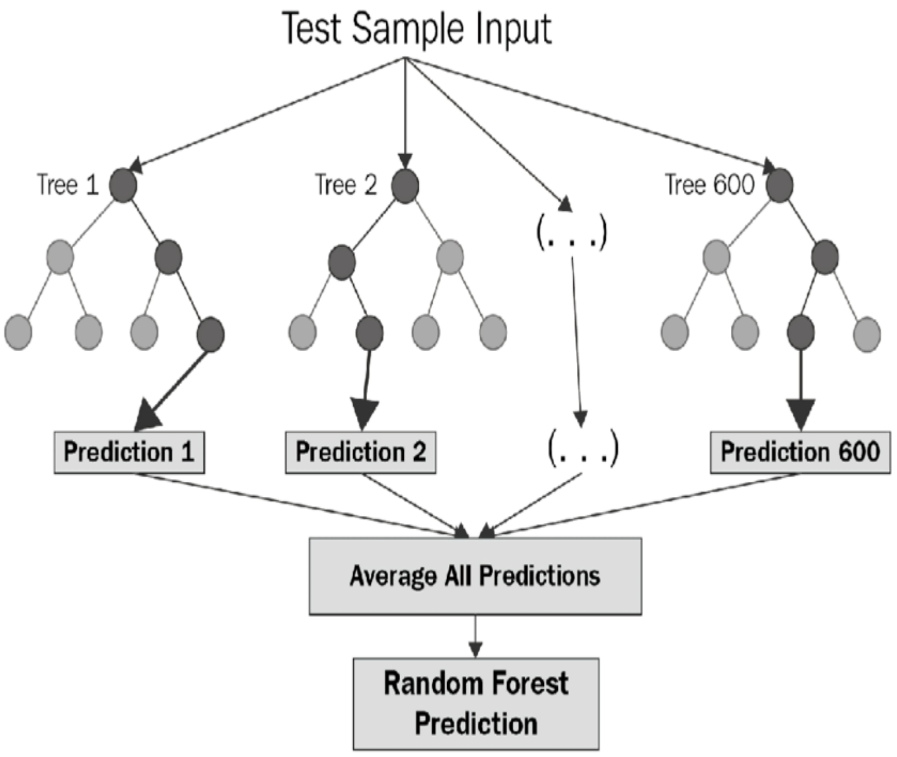


Figure 5.2: Basic steps of a Random Forest model (adapted from [212]).

5.6.1 Random Forest

A decision tree (DT) is a promising classification technique that forms a conditional predictive model by recursively clustering the training (labeled) data samples into smaller subdivisions satisfying some criteria from the feature sets at each branch (or node) in the tree [79]. The formed tree has a root node, a set of internal nodes, and a set of leaf nodes. However, DTs have several advantages over traditional supervised classification approaches. In particular, DTs are strictly nonparametric and do not assume the distributions of the features of the input data [79]. In addition, they handle nonlinear relations between features and classes, missing values, and are capable of handling both numeric and categorical inputs. Finally, decision trees are widely applied because the classification structure is explicit and easily interpretable. However, enhancement of the limitations of DT is possible. For example, the Random Forest (RF) is a classification algorithm consisting of many decision trees [146]. It is an enhancement of the Decision Tree to remove the limitation of training data sensitivity and accuracy. In the decision tree, small changes to the training samples may produce quite different tree structures as a prediction model. In sharp contrast to DT, RF is formed based on bagging and feature randomness when building each individual tree to eradicate the limitation. Bagging is the combination of bootstrapping and aggregating the votes (as shown in Fig. 5.2). It creates an uncorrelated forest of trees whose prediction by the committee is more accurate than that of any individual tree. During the training model generation, it does follow bootstrapping for dataset embedding and then random feature selection for each tree. Bootstrapping method creates a dataset for a tree with the same sample numbers but randomly selects samples from the original set, meaning that the same sample may appear more than once. Finally, the bad predictions may balance out with good predictions by some trees in the Random Forest. However, random forest is memory-intensive and computation-intensive for large datasets. For splitting the data into tress, entropy, **Gini impurity** or information gain can be used.

5.6.2 RNN-LSTM

One of the crucial challenges in text classification is feature extraction. A common textual feature is the bag-of-words (BoW) model, where unigrams, bigrams, n-grams, or manually crafted patterns are typically extracted as features (as we have done in the previous study). Several other feature selection methods are also applied to select more discriminative features, such as frequency, pLSA, LDA, and so on. Yet, these methods mostly ignore the contextual information or word order in texts and can not properly capture the semantics of the words. Lai et al. [128] reported that – *Although high-order n-grams and more complex features were designed to capture more contextual information and word orders, they still have the data sparsity problem.* This issue heavily affects text classification accuracy. Recently, the development of pre-trained word embedding and deep neural networks has been promising for texts processing by solving data sparsity problem [164, 165, 102]. Here, word embedding is a distributed representation of words and significantly eradicates the data sparsity problem. In another way, the neural transformation of a token/word is called word embedding and is a real-valued vector [128]. That said, word embedding can be leveraged to measure word relatedness by using the distance between two embedding vectors. Several researchers reported that pre-trained word embedding could capture meaningful syntactic and semantic regularities [219]. Among them, the Recurrent Neural Network (RNN) model processes and transforms a text token by token and stores the semantics of all the previous text in a fixed-sized hidden layer [224]. The advantage of RNN is the ability to better capture contextual information. This could be beneficial to capture the semantics of texts in commit messages and developer discussions. RNN is a subsequent enhancement of neural networks.

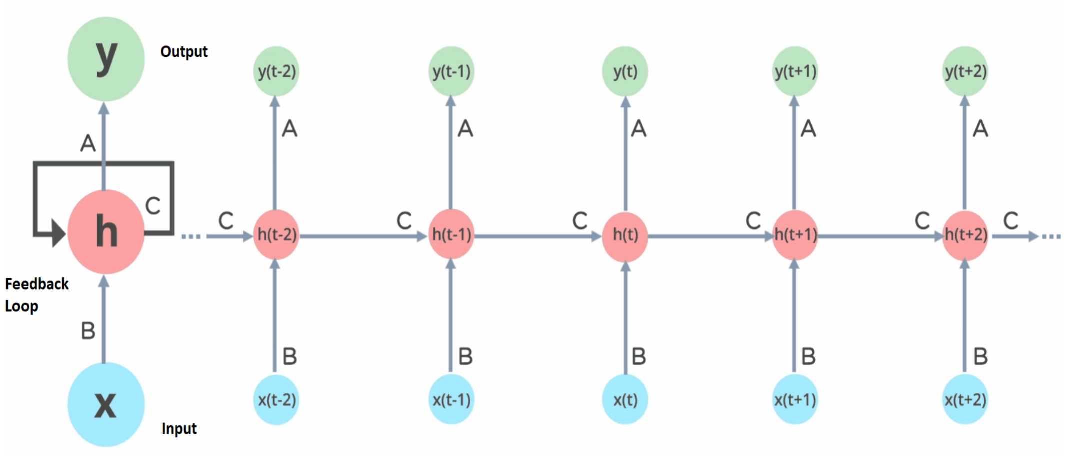


Figure 5.3: Basic structure of a Recurrent Neural Network (adapted from [6]).

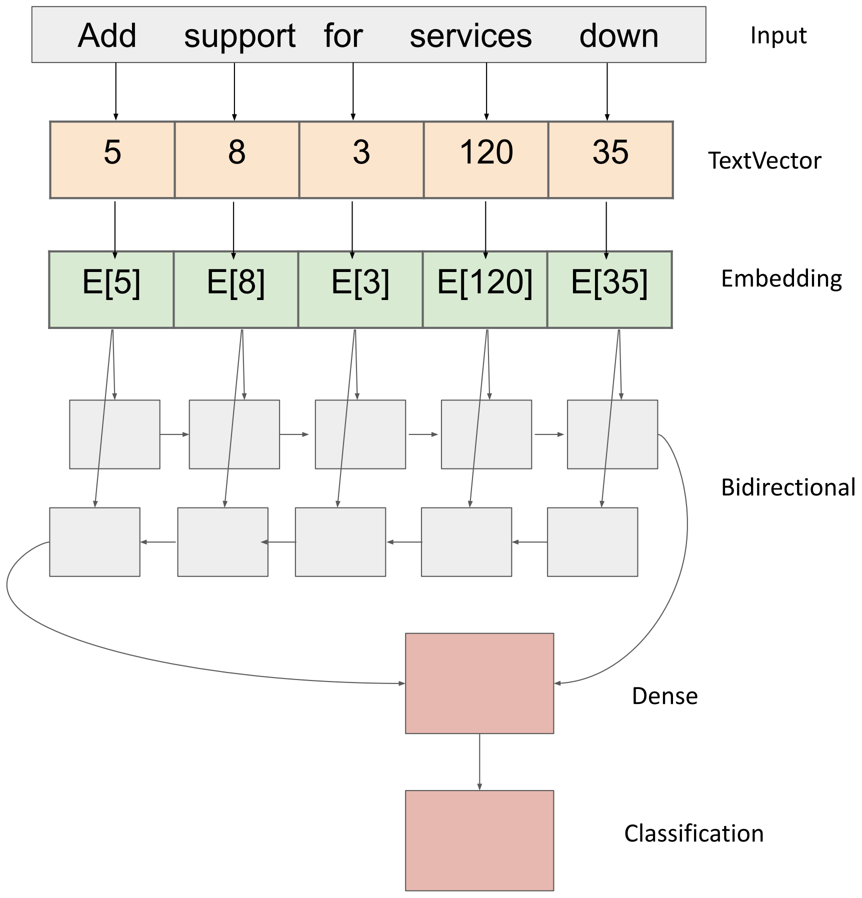


Figure 5.4: Text processing for classification by a DNL model (adapted from [1]).

Neural networks, a widely used classification technique, try to mimic the way the human brain develops classification rules [68]. A neural network can have multiple layers of nodes, with each layer (nodes) receiving inputs from previous layers and passing outputs to further layers (nodes). The way each layer output becomes the input for the next layer depends on the calculated weight to that specific link, which depends on the cost function, and the optimizer focusing on the targeted output for classification. The neural net iterates for a predetermined number of iterations. After each iteration, the cost function is analyzed to see where the model could be improved. The optimizing function then alters the internal mechanics of the network, such as the weights, and the biases, based on the information provided by the cost function until the cost function is minimized for prediction. When a new sample comes, it considers the features of the sample and produces predictive values to the classes based on the previously adjusted weights (corresponding to the features probably) from training samples. However, a recurrent neural networks (RNN), an advancement of the neural networks, work better for textual data. Because the textual description is a sequence of words that express an intention [128, 224]. A feed forward neural network with hidden layers of nodes (each node has some sort of calculated weight that is later used for prediction) can form a decision region from training data for classification [105]. In contrast to the basic neural network, RNN has feedback loops in the network layers and hidden states to update already calculated weights while moving forward with the next sequence of inputs [38, 248]. RNN maintain a hidden state vector to capture a digested representation of the current context as they scan forward, token by token (as shown in Fig. 5.3). Learned parameters both read out this vector to predict the score of a token and update this vector upon seeing the next token. These models are quite effective when trained with sufficient data. However, it has a vanishing/exploding gradient problem in training. To handle these problems, mechanisms such as ReLU and Gradient clipping techniques are employed. RNNs are further extended with Long short-term memory networks (LSTM) that can be trained to selectively *”forget”* information from the hidden state [224]. This strategy is essential to consider more important information during the learning process. Moreover, LSTM can learn the presence of semantic phrases and sentences that are susceptible to word order. Another crucial property of the LSTM is that it learns to map an input sentence of variable length into a fixed-dimensional vector. However, we employed the most advanced bidirectional RNN (in 2020). It reproduces the input forward and backward through the RNN layers and then aggregates and sum-ups the final output. In addition, the crucial benefit of a bidirectional RNN is that the information from the beginning of the input may not require to consider for every timestep in the processing path to affect the output. The basic steps of text processing by this mechanism are summarized in Fig. 5.4. However, Gradient calculation is crucial for optimizing the weights of a network model. For gradient calculation, we use cross-entropy (softmax) as the loss function. The purpose of the cross-Entropy is to take the output probabilities (P) and measure the distance from the expected values.

5.6.3 Testing with the Golden Set

We train our proposed method (ArchiNet) and other methods with the training set. Train and test set partitioning is described in Section 5.4. Then, the classification performance is tested with the test set (from Table 5.2); comparison of the outcome is presented in Fig. 5.5. Please note that only methods having close performance are shown here. The most promising method in the baseline work by Mondal et al. is DPLSA, where discriminating keywords for the individual classes are used as features in a probabilistic model. The difference in the percentage of F1 score between ArchiNet and DPLSA for all classes is 24 points higher, while this difference is 35 points higher for the adaptive category. The F1 score of our model for the test data in [172] is 63% (shown in Table 5.6), which is 18 points higher compared to their best model (45% gain in performance). We have employed a DNL based text classifier [128, 224] with Google Tensorflow [10]. The DNL network where encoded words are embedded with the RCNN-LSTM strategy shows a 61% F1 score, which is 2 points lower than ArchiNet. The configuration of our DNL model has 64 layers, 64 units, epoch size 10, *relu* activation function, and cross-entropy as loss function [224].

Furthermore, we adopted the best algorithms suggested by Hindle et al. [100] to classify large change commits into five categories, and Soliman et al. [220] to classify architectural discussions. We also explore Naive Bayes (NB), Decision Trees (DT), and Random Forest (RF) [146, 100, 220] for our dataset with the WEKA [92] tool utilizing word-to-vector features [164]. Among them, the most promising classifiers such as NB, and DT have less than 55% F1. However, Random Forest (RF), which forms a group of DTs, produces around 58% F1 score for our dataset. The F1 score produced by our technique for the adaptive category is much higher than the competing methods. The ranges of precision and recall rate of ArchiNet among the individual categories are 42.4–77.8% and 64–73.7% respectively, which are more consistent than other classifiers. Notably, from the graph, we can see that F1 scores of RF and DNL for the perfective category is higher than ArchiNet, while significantly lower in the adaptive category because many samples from adaptive might be falsely predicted (high recall rate) into the perfective category (due to lack of handling mechanism of the overlapped concepts). We also see this pattern in the 10-fold validation phase. In this evaluation phase, the distribution of P, R, and F1 scores to all the classes with the test sets indicates a better and stable outcome of ArchiNet with the concept-words.

10-folds Validation

In this phase, we show how our classifier is performing with cross-fold validation since it provides a more accurate evaluation against the over-fitting problem [100, 166]. However, we experiment with the promising methods proven in the first phase. We compare the performance of ArchiNet with DNL and RF by 10-fold cross-validation technique. In 10 iterations, we take 90% samples as the training set, and 10% as the test set exclusively for each of the iterations [166]. The performance comparison is presented in Table 5.5. The F1 score of ArchiNet is around 69%, which is 7 points better than the two classifiers. Deep learning with

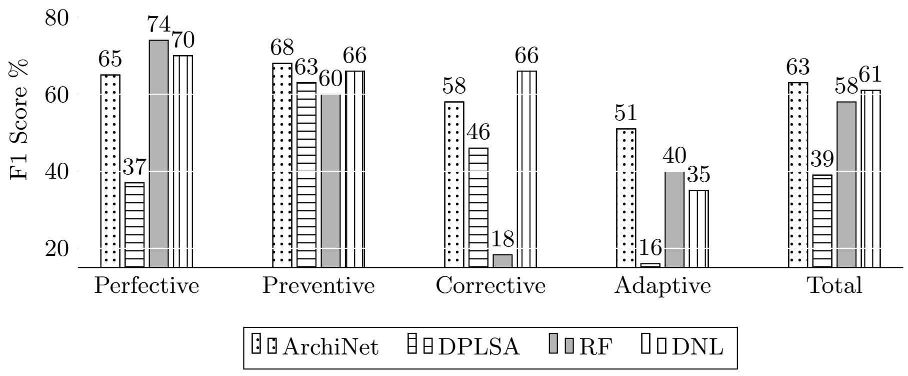


Figure 5.5: F1 score comparison of ArchiNet with the most promising classifiers DPLSA [172], DNL [224], RF [100].

Table 5.5: Performance (%) comparison of ArchiNet (A), Random Forest (RF), and Deep Neural Learning (DNL).

Metric	**Perfect**			**Correct**			**Prevent**			**Adapt**			**Combined**		
	$ANet$	RF	DNL	A	RF	DNL	$ANet$	RF	DNL	$ANet$	RF	DNL	$ANet$	RF	DNL
P	77.5	62.7	77.6	63.4	88	50.3	77.8	91	66.5	42.4	96	40.1	69.1	76	62
R	73.7	99	69	66.25	19	62	63.8	43	62.1	64	25	28	69	67	62.1
F1	**76**	77	73	**63**	30	50	**70**	58	64	**51**	40	33	**69**	62.2	62

RCNN-LSTM [128, 224] shows 62% F1 score; RF has a similar outcome as of DNL. F1 scores for some other classifiers are between 50 to 60% with the word-to-vector [164] features. From the median and range values in box plots in Fig. 7.3, it is observed that the precision and recall rate of ArchiNet (in the ranges 40.1–77.8% and 63.8–73.7% represented by A_p and A_r) are consistent with all the classes (recall is highly consistent than others indicated by DNL_r and RF_r). For the adaptive and corrective categories, the outcome of ArchiNet is significantly higher. Poor recall rate of RF and DNL (marked with circles in Table 5.5) for the adaptive category and high recall rate for the perfective category indicates that many samples from adaptive are falsely retrieved into perfective by both of the classifiers. A similar trend is observed with the corrective category except for lower precision for DNL. Since RF and DNL do not distinguish and select words based on concepts/semantics, they produce more unstable outcomes. In summary, our proposed classifier has better performance for all metrics (F1, P, and R scores) compared to other classifiers because concept-words handle various influential tokens from a commit message efficiently. This exploration answers the research question RQ2.

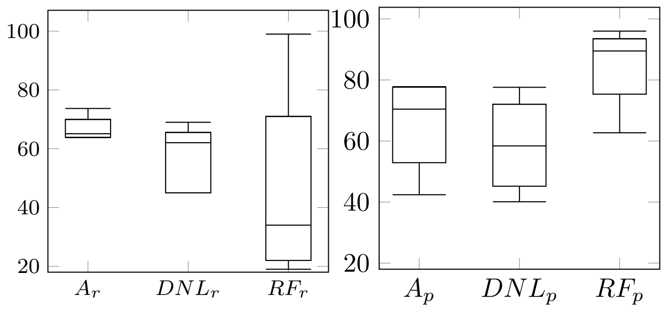


Figure 5.6: Range of Recall (r) and precision (p) rate of all classes

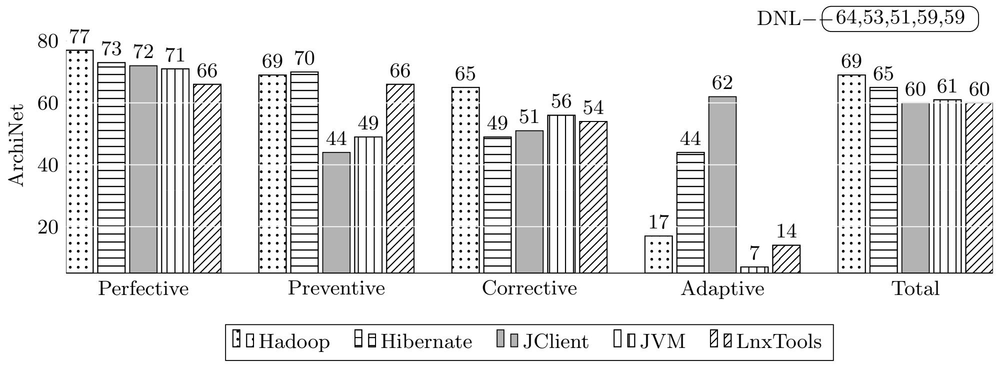


Figure 5.7: F1 score for individual projects.

Project-wise Validation

We also conduct cross-projects validation of our proposed approach. We train the classifier with four projects and test with the remaining project in each iteration for the five projects. The project-wise outcomes for both ArchiNet and DNL are presented in Fig 5.7. Combined F1 scores of each of the projects produced by ArchiNet are better than that of DNL. None of the project's F1 scores is below 60% for ArchiNet, while the highest is 69%. The highest precision is 85%, and the recall is 80% (for the perfective and preventive category) for our method. However, the precision and recall can be low for the adaptive category as can be seen in the Fig 5.7. On the other hand, the adaptive category's F1 score reaches 62 for the ArchiNet (whereas 23 for DNL). Performance of some of the projects is lower than 10-fold validation because of insufficient training data. Overall, Hadoop's outcome for both ArchiNet and DNL is the most promising because the commit messages in Hadoop might contain a less ambiguous explanation compared to other projects.

Sensitivity of Tokens

The performance sensitivity of ArchiNet (for 10-folds) for various token-weights ($w(t)$) combination (in (5.1) and (5.2)) is shown in Fig. 5.8. The best performance is shown for the combination $\{t_w, t_p, t_s, t_a\}$ which is three points (69% F1) better than only considering natural terms (t_w) (66% F1). Including API, library, and framework name (t_a) increments the performance by two points as there is more likely to be an adaptive category for those compared to others. As can be observed from precision and recall in Fig. 5.8, the adaptive

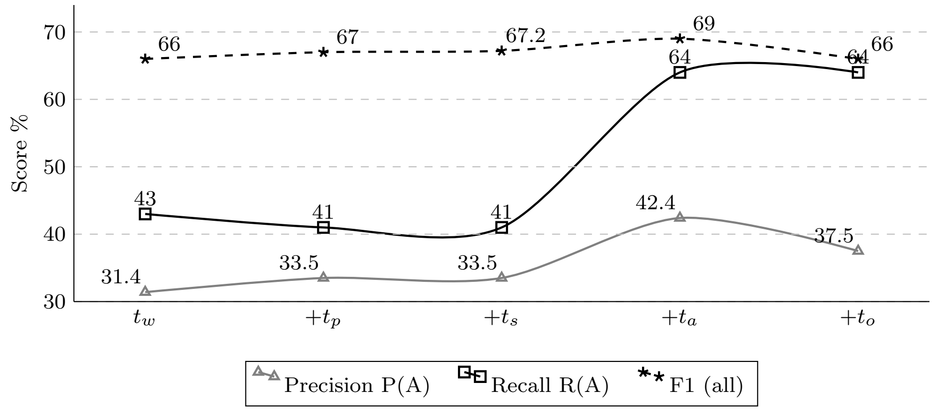


Figure 5.8: Performance sensitivity of terms. '+' means including all others terms from left. *P*,*R* are from the Adaptive(A) class.

Table 5.6: F1 of ArchiNet with our data and data in [172].

Dataset	**Perfective**	**Corrective**	**Preventive**	**Adaptive**	**All**
Our data	65	58	**68**	**51**	**63**
Data [172]	55	**61**	**80**	16	**63**

category is the most sensitive. However, we notice that combining source-code operations (t_o) affects the performance slightly negatively (66% F1, whereas it is 65% with t_w); therefore, source code operations are not promising features for classifying the architectural change.

5.7 Threats to Validity

One of the greatest threats to the validity of our result is that annotating the intention of change is subject to human bias. To reduce this threat, two of the authors independently annotated, and then conflicts are resolved by discussion. Any classifier may suffer an over-fitting problem. To overcome this, we experimented with our classifier with a tenfold cross-validation technique and found a promising result. Another concern of our classification model is how general it predicts change from different programming languages and cross-projects. One of our test sets is collected from Mondal et al. [172] that also contains projects of Python language, and have similar outcome as shown in Table 5.6. A few of the projects such as Hadoop has substantial industrial participation [115]. Therefore, our study also mitigates generalizability threat to some extent.

Our model can be trained with different metrics. Therefore, for (5.2) in Section 5.5, we also have trained our model with the *TF-IDF* metrics. However, the result is not as promising as the direct probability value, but still shows a better result than DPLSA, LLDA, and SemiLDA. With this metric, the best F1 scores for the data in [172] are 47% and 51% for our benchmark data. Yan et al. [255] utilized DPLSA for predicting multiple categories of usual changes (three types). We found only a few of the samples in our data have multiple intentions when architectural changes happened. ArchiNet can handle such scenarios to some extent as we experiment on that mode; when the predicted sample is in Hit@2 [220, 201] (within the top 2 ranks), the F1 score is 83.5%. Notably, our proposed classifier is versatile and does not require parameter tuning, unlike others. Our dataset and trained models are available in *github.com/akm523/archinet* for further investigation.

5.8 Related Work

Architectural Design Issues and Solutions Classification

Yamauchi et al. [254] proposed a technique considering program identifiers to group the large commits into related components having relations with the functional requirements. An early approach of committed code classification was studied for architectural tactics (design solutions such as resource pooling, secure session management, and so on) [166] based on code identifiers (such as heartbeat) mapped with text description (heartbeat emitter and receiver) from a set of trained samples, and commits are predicted using a term-frequency based classifier. Solaiman et al. [220] reported Bayesian Network and Naive Bayes as the best algorithms to classify architectural discussions related to six ontology classes (such as technology) into three design steps focusing ambiguous concepts (such as *server* has different meanings for different cases), concepts expressing reasons of architectural changes are different than those. However, although these classes were either subset or irrelevant to architectural changes, they were not specialized in four architectural changes. In our work, we explore both source code features and concept-token properties to predict the reasons for architecture changes.

Architectural Change Classification

We are aware of only one study by Mondal et al. [172] to categorize four architectural changes from the text. Their model was generated by popular discriminating feature selection techniques DPLSA [255], SemiLDA [80], and LLDA [202] originally proposed [255] for classifying all software changes into three groups, and none of the techniques could handle the twists and challenges of architectural change classification properly. Consequently, the outcome of their proposed technique is poor. Another study by Hindel et al. [100] close to ours explored various machine learning techniques for classifying large commits (commits with many files changed) into five groups. We also explore the promising classifiers reported by them: Naive Bayes, Decision Trees, and so on. However, Random Forest (RF), an advanced version of Decision Trees, produces promising outcomes with our dataset (but 7 points lower F1 than ArchiNet). Recently, word embedding technique that captures contextual and semantic information with deep learning is being successfully used for software artifacts analysis and classification [128, 224, 163]. However, due to the overlapping of concept words, deep learning might not produce the best outcome, which is mostly inexplicable when multiple intentions are required to extract from a single message. Our proposed classifier ArchiNet handles these concerns considering other tokens and gains in performance.

5.9 Conclusion

In this study, we present a dataset collected from five popular projects and a promising classifier for architectural change categorization from texts. Our study identifies the challenges of classifying changes from both source code properties and textual properties. We address those challenges with a concept analysis approach that indicate the developers' intentions. Both 10-fold cross-validation and cross-projects validation show that our technique is promising in all aspects compared to traditional methods (F1 score is 70%). We also explore the sensitivity of the performance of our classifier for various tokens. Besides, we extract around 237 keywords (with trained weights for each change category) from the training set. Given the success, many of the text analysis approaches to support the ten activities of software architecture discussed by Bi et al. [34] might be enhanced by adopting our proposed technique. In future, we will explore automatic design documentation generation and architectural versioning schema applying our change classification technique.

Although the concept tokens are promising for change classification from commit messages, in many cases, it is not possible to capture the concept tokens. In that scenario, code component change relation properties can be leveraged. Moreover, the message triad (empty, meaningless and tangled) should be handled. In the subsequent studies, we will explore lightweight techniques to detect and slice such code properties and enhance change classification, utilizing them.

Chapter 6

Architectural Change Instance Detection and Extraction from Code Properties

In a previous chapter (i.e., Chapter 3), we found that architectural change detection leveraging the textual properties is promising, but it has limitations. Directory and naming structure patterns and *diff* tool change information can complement the challenges. In this chapter, we explore code properties for detecting architectural change and extracting relations considering the challenges of the previous studies. During the manual analysis of the changed code of 3,647 commits from 10 OSS projects, we noticed that DANS (directory and naming structures) properties play a crucial role in determining the high-level architectural or design impactful (DIS) change instances. In this process, we observe various DANS properties and their variations across the change revision commits of the projects. The list of these properties (and types) is later discussed in Chapter 6. However, implemented architectural change instance can be detected in two ways: (i) comparing ASTs from byte code of two consecutive commits and (ii) processing the provided information by the commit *diff* tool of the VCS [88, 5]. In the first technique, a complete codebase for each committed version needs to be compiled into byte code, which mostly requires manual intervention to resolve 3rd party library dependencies. Moreover, the intensive computation could be a bottleneck for a normal purpose machine for analyzing changes at the statement levels for large systems. Usually, each release of a project may contain hundreds of commits. Thus, AST-based techniques might far exceed the manual analysis's time and efforts for architectural change detection. Therefore, we propose to detect high-level architectural change revisions by processing the DANS properties, codebase, and *diff* tool information using the *String* processing techniques.

The rest of the chapter is organized as follows. Section 6.1 presents a brief overview of our study. Section 6.2 presents the background and motivating example. Section 6.3 presents the dataset collection process. Section 6.4 discusses the change detection process. Section 6.5 reports the semantic slice generation process. Section 6.6 presents performance results. Section 6.7 discusses related work. Section 6.8 concludes our study with future work.

6.1 Introduction

Software maintenance activities induce the most efforts and costs in a software system's lifecycle [131]. Understanding and updating a system's architecture is crucial for software maintenance [82]. As parts of the maintenance activities, development teams review changes [228, 53], and design structure after completing a certain milestone or release (or in the late-lifecycle) for various reasons such as paying technical debts, fixing flaws or correcting the design structure [17, 87, 116]. In this review process, requirements and associated design information extraction, semantic design recovery (concerned with the production of meaning, and how logic and language are used in designing a software [98]), design summary generation, untangling changes, and so on are essential tasks [228, 213, 61]. Collectively, we refer to them as design review tasks [228].

For these tasks, architectural change instances are required to be detected and decomposed by the automated tools to reduce human efforts [131, 14]. In this study, to support the design review tasks, we propose a lightweight tool for detecting module-level architectural changes and generating semantic slices of the change instances from source code. We rely on the directory and naming structures (DANS) properties for developing our tool. Please note that our extracted semantic slices are based on the structural relations that are mostly different than the slices generated by the existing tools [61, 140, 240, 138, 238]. A structural semantic slice (SSC) consists of a description of relational information of involved modules, their classes, methods and connected modules in a change instance, which is easy to understand to a reviewer [140]. With these SSCs, various layers of abstract design summary generation are possible [113]. For example, a high-level abstract summary can be automatically generated from such a slice as – *The ASBC module defines a sensitive STC method with new runtime dependency on ASC module for authorizing sensitive service access* (underline texts represent an architectural relation, acronyms are discussed in Section 6.2). Besides, this information is crucial to feed into other DevOps tools such as design decision recovery [214] and design note generation [172].

In a release or a milestone, the changes that contain architectural instances require special attention from the development team due to the far-reaching consequences [195, 87]. However, software architectural changes involve in more than one module or component and are complex to analyze compared to local code changes. Architectural aspects (design) of a change task are reviewed considering crucial scenarios such as access rules and restrictions on usage of program entities across the modules. Moreover, design review is essential when proper architectural formulations are paramount for developing and deploying a system [153, 116, 87]. Researchers are working for supportive tools and techniques for reviewing change commits focusing on atomic or local changes [61, 240]. However, little or no effort is given to support reviewing architectural change instances of those projects by slicing the structural relations.

As a preliminary study, here we investigate the SSC generation for Java Platform Module System (JPMS) based projects (in Java and Kotlin) [35, 153] (where the architectural organization is predominant). JPMS is introduced to handle coupling and dependency among modules to reduce bloated software, performance issue, security backdoors, and higher maintenance costs [87, 116] by well-defined modules and access restrictions among them. However, despite built-in supports for handling JPMS, development and maintenance teams require tracking a complex knowledge-base of static and run-time architecture and are still error-prone. That is why supportive review tools are urgently needed.

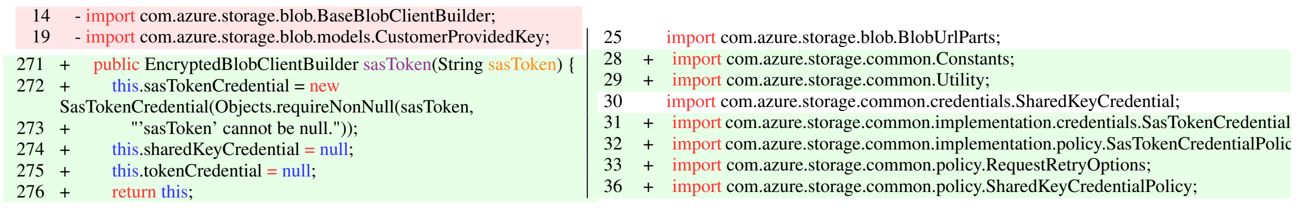
For our tool, we define an M2M (module to module) metric for high-level architectural change [131, 81] of JPMS based projects following the existing metrics. We identify several observations (Table 6.2) on the DANS properties of the program entities for detecting architectural change instances based on this metric and decomposing the SSCs. We extract the DANS properties (such as addition, deletion, moving or shrinking multiple class imports) based on regular expressions of string matching. Since our tool does not process the Abstract Syntax Trees (AST) properties, it does not require compiling each version (that requires human intervention to fix 3rd party library dependencies). Thus, our approach is easy to deploy with the version control system (VCS) APIs. Preliminary evaluation of our tool with ten open-source projects indicates that the DANS properties produce highly reliable precision and recall rate (93-100%) for detecting and decomposing architectural change slices. In summary, the key contributions of this study are:

- We construct a benchmark dataset containing module-level architectural change (M2M) commits and semantic slices of the changed code for advancing research in this domain.
- We manually explore change commits and identify 16 types of directory and naming structure (DANS) properties necessary for automatic processing (for any tool) of the architectural change instances from the source code.
- We develop a lightweight tool using the DANS properties for module-level architectural change detection and semantic slice generation of the changes based on the special structural relations in the source code.

6.2 Motivation and Background

Background: This section aims to provide an idea of how architecture can be a centrepiece of modern-day software development and why reviewing architectural relations is crucial. From Java 9, we can define a concrete module compared to a conceptual module than we could before JPMS. Concrete modules accelerate containerization of cloud-service based systems more efficiently and securely [116]. JPMS is built for supporting the following core principals [153]: (i) prevents unwanted coupling between modules, (ii) only exposes well-defined and stable interfaces to other modules, (iii) provides a reliable configuration of the dependent module, and (iv) controls reflective access to sensitive internal classes. The Java module system will have a profound impact on software development. In JPMS, a *module* is a uniquely named collection of reusable packages which is defined by a descriptor file called *module-info.java* having meta-data, including the declaration of named module [87, 35]. A named module specifies (1) its dependencies of classes and interfaces (entities) on other modules and should specify (2) which of its entities are exposed to other modules for usage. Some of the operations provided by JPMS [153] for handling these specifications are:

- *requires (R)* - express its dependency on the other module.
- *provides (P)* - provides an implementation of an interface with another class as an implementation class.
- *opens/open (O)* - gives run-time access and open for use it with reflections. It is used to expose the whole module.



```
14    - import com.azure.storage.blob.BaseBlobClientBuilder;
19    - import com.azure.storage.blob.models.CustomerProvidedKey;
271  +    public EncryptedBlobClientBuilder sasToken(String sasToken) {
272  +        this.sasTokenCredential = new
SasTokenCredential(Objects.requireNonNull(sasToken,
273  +            "'sasToken' cannot be null."));
274  +        this.sharedKeyCredential = null;
275  +        this.tokenCredential = null;
276  +        return this;
```

```
25       import com.azure.storage.blob.BlobUrlParts;
28   +   import com.azure.storage.common.Constants;
29   +   import com.azure.storage.common.Utility;
30       import com.azure.storage.common.credentials.SharedKeyCredential;
31   +   import com.azure.storage.common.implementation.credentials.SasTokenCredential
32   +   import com.azure.storage.common.implementation.policy.SasTokenCredentialPolic
33   +   import com.azure.storage.common.policy.RequestRetryOptions;
36   +   import com.azure.storage.common.policy.SharedKeyCredentialPolicy;
```

Figure 6.1: A change that sets the SAS token used to authorize requests sent to an Azure service. Here, the plus (+) sign with green background indicates addition and the minus (-) sign with a red background indicates deletions.

- *uses (U)* - instructs run time loading of services.
- *transitive (T)* - expresses implied dependency for API. It ensures that any module which requires second module also implicitly requires third module (linked to the second module).

Each of these operations are architectural and have greater implication in terms of both static and run-time behaviors of a project. We call these operations as module operations (MO).

Motivating Example: In this section, we explain a subset of modifications of *EncryptedBlobClientBuilder (EBCB)* class of *azure-storage-blob-cryptography (ASBC)* module from a commit of the Azure Java SDK project [58]. The modifications are shown in Fig. 6.1 following the same GUI representation in github.com. In JPMS projects, usage of module entities has purposeful rules and restrictions due to runtime access of sensitive parts, dynamic containerization and separation of concerns. If the reviewers want to review architectural changes to revisit those, they must first identify whether the commit contains such changes. They also need to extract various other code related information [140, 240].

For example, in Fig. 6.1, a new method called *sasToken()* is added in lines 271-276. The reviewers have to figure out which classes and modules are involved with this method and with which it has new dependencies. Variables in lines 272, 274 and 275 need to be searched in multiple places and compared for references to find the dependencies. Some of the candidates of those variations are discussed later in Section 6.5. However, statement 272 uses a newly imported class *SasTokenCredential (STC)* in line 31. Next, they have to find which module it belongs to. But, the reviewers cannot determine the module with this GUI interface (or with the provided information by the VCS API) perfectly; not even if it is from the same module of the *EBCB* class such as the *BaseBlobClientBuilder* class in line 14. For instance, despite being from different modules, both lines 14 and 29 contain a significant portion of the common structure (*com.azure.storage*). For these, they have to search the location of the class in the codebase (perhaps with the IDE). This manual search returns the directory from where they identify the module, which is the *azure-storage-common (ASC)* module in this case (a cross-module). Such an instance is a candidate for revisiting the cross-module rules and restrictions due to various concerns discussed in the Background section. However, the commit contains modifications of 32 classes with many variations. Thus, the manual process to extract the crucial code information is time-consuming, tedious, and error-prone (such as the same class name exists within multiple modules). Besides, extracting the described information is crucial for many other analytic techniques such as design decision recovery and multiple intentions detection. Consequently, our study contains two steps: (i) detecting commits containing architectural change instance, and (ii) then slicing those commits.

6.3 Dataset Preparation

We collect and prepare our experimental dataset in various phases. For collecting JPMS based projects, first, we search commits containing *module-info.java* in GitHub. Thus, we get almost 200 projects, many of them are large or small or toy projects. Among them, we selected ten projects of various domains having the highest number of commits (and multiple modules), excluding native JDK-related projects. We filter out the commits having structural code change [150, 172] or having modification in *module-info.java* from all the commits in the period of July 2017 to July 2020 because of the official release of JPMS in 2017. In this way, we collect a total of 3,647 commits (*Selected* column in Table 6.1). In the final phase, we manually determine the commits having an architectural change instance (M2M). It took around **240** working hours to complete the analysis of the commits. We found around 2,720 such commits, which are presented in the *M2M* column in Table 6.1. We use this dataset for investigating the automated tool development utilizing the DANS properties.

Table 6.1: M2M dataset and change detection performance.

Project	**Commit**	**Selected**	**M2M**	**P**	**R**
HibernateSearch[236]	9504	53	20	1.0	1.0
Aion[18]	4718	1064	863	1.0	0.98
Webfx[244]	3770	778	563	1.0	0.99
Speedment[222]	4483	243	222	0.97	1.0
AzureSDK[76]	15,180	276	244	0.99	1.0
Atrium(Kotlin)[20]	1988	379	210	0.98	1.0
Bach[43]	2114	365	145	0.99	1.0
Vooga[235]	1210	447	416	0.96	1.0
Imgui(Kotlin)[106]	1703	35	34	1.0	1.0
MvvmFX[180]	1100	7	3	NA	NA
	Total	3647	2720		

6.4 Architectural Change Detection

6.4.1 M2M Change Metric

Detecting architectural change is an ongoing research. The most popular metrics for detecting higher level changes are $A\Delta$ [110], A2A [131], MoJoFM [245], and C2C [81]. Adopting these metrics, we define a new metric for JPMS based projects, which is called the module-to-module ($M2M$) metric. Our metric is based on architectural changes at the module (M) level, arguably the higher level. Constraints of the M2M metric are defined based on A2A (or C2C) and ID-SD (include and symbol dependency) [150] metrics suited for JPMS. Deleting, adding and moving modules or their respective classes are considered A2A delta operations. In contrast, ID-SD considers the modification of classes and interfaces importing (for Java and Kotlin) from

Table 6.2: Observation of directory and naming structures

SL	Type	Description
1	Import	Code location change appears as deletion and addition
2	Import	Shrinking and elaborating multiple imports appears as deletion and addition
3	Import	VCS APIs only return the first and last lines as modification of the multiple imports commented with $\backslash * .. * \backslash$
4	Import	New Java allows importing static method and inner class
5	Import	Kotlin offers static method import without any syntax variation
6	Import	JPMS config includes both class name and package name
7	Dependent	Methods have subsequent relations and contain relative references
8	Directory	Both Kotlin and Java modules have uncommon directory structures
9	Directory	Directory name of one module could be similar to sub-directory name of another module
10	Directory	Some of the modules have only one root package/directory name
11	Directory	Module has submodules
12	Naming	Import like directory name such as *application.api*
13	Naming	Similar class name in multiple modules
14	Naming	Class and package renaming appears as addition and deletion
15	Naming	Class and method name in Kotlin do not have different patterns
16	Naming	Module name in module-info.java is different from directory name, and used in ambiguous ways in import

different modules (representing $M(IDSD)$), not from the same module. Module operations (MO) (described in Section 6.2) update within the *module-info.java* files changes at least the runtime architecture irrespective of changes within the class files. Consequently, we also consider the modification of them for the M2M metric. Hence, a module-level architectural change metric $M2M$ may contain any of $\Delta a2a$, $M(IDSD)$ and MO changes.

6.4.2 M2M Change Detection Process

During the manual analysis of the 3647 commits, we notice that DANS properties play a crucial role in determining the M2M instances. We observe various properties and their variations across the commits of the projects. The list of these properties (and types) are presented in Table 6.2. However, the M2M instance can be detected in two ways: (i) comparing ASTs from byte code of two consecutive commits, and (ii) processing the provided information by the APIs and libraries of the VCS [88, 5]. In the first technique, a complete codebase for each committed version needs to be compiled into byte code, which mostly requires manual intervention to resolve 3rd party library dependencies. Moreover, intensive computation could be a bottleneck for a normal purpose machine for analyzing changes at the statement levels for large systems. Usually, each release of a project may contain hundreds of commits. Thus, AST-based techniques might far exceed the manual analysis's time and efforts for architectural change detection. Therefore, we aim for a lightweight tool based on the second technique.

We have extracted code change information in between two consecutive commits by *git* APIs with the help of GitPython [88], and PyDriller [5]. It provides string/text of the modified code segments with line numbers, methods and classes. Among the 2,720 manually extracted M2M commits, we have selected 48 commits (5

for each project except mvvmFX) from 10 projects having multiple intentions in the commit description. These 48 samples are used as so called training samples in our experiment for automated tool development with the DANS properties. We have manually analyzed all types of changes of those commits and found that code information returned by the VCS APIs has some non-trivial challenges that might compromise the detection process's accuracy. These challenges are described in Table 6.2. Here, we discuss some of them. For example, for SL8 in the table, we have identified at least three types of directory structures of JPMS modules where the class files might reside in: *"module_name/**src**/class_dir"*, *"module_name/**main/java**/class_dir"*, and *"module_name/**src/main/***
***java**/class_dir"*; but, this information is not included within the class imports as can be seen in Fig. 6.1. One instance for SL2 is that *import aa.1, import aa.2, and import aa.3* can be shrink to *import aa.*;* and vice-versa. Another complex challenge is distinguishing between *import renaming* and new *import addition* in SL1. For example, renaming *import aa.**b**.1* to *aa.**c**.1* due to directory renaming is appeared as a deletion and an addition operation. One example for SL16 is that the similar directory structure *ch/tutteli/atrium/core/api* of the module name *ch.tutteli.atrium.core.api* does not exist up to that commit but used within other module as import. Comment within the change information also poses challenges, such as commenting as shown in SL3. We point out some other anomalies in the commits and handle all these concerns using regular expressions in string processing. We have developed a tool in the Python platform to handle these observations for the DANS properties.

6.5 Semantic Slice Generation

Based on the DANS properties in Table 6.2, we generate semantic summary (architectural) containing relational information (SSC) of the changed code snippets presented in Table 6.3. We extract 16 types of change relations (such as cross-module class used in a newly defined method) involved in architectural change instances. One of the slices for Fig. 6.1 would be, *ASBC:EBCB=>sasToken<- ASC:STC*. As discussed in Section 6.2, this slice represents that it is a M2M instance where *EBCB* class of *ASBC* module added *sasToken* method that is dependent on the *STC* class of *ASC* module. A complex change might contain many such slices of all the information in Table 6.3. Our extraction process depends on directory processing, *module-info.java* file processing, methods processing, and searching within the programs before and after modification with regular expressions for extracting the DANS properties. Initially, we thought the process would be straight-forward. The challenges described in the previous section significantly influence the performance of the automated technique. However, for including a method within a slice, we handle the SL7 in the table as follows (along with other common concerns such as removing the Java keywords). Let's consider that class A is involved in an M2M instance, then following would be the candidates of the search process for a method:

- B <T> objectB = new B<A>(), then objectB is used.
- B objectB = C.getObj(A), then objectB is used.

Table 6.3: Information in the semantic slices

#	Entity	Change Relation
1	JPMS>>Direct module/ MO add, delete, modify	connected jpms+API modules disconnected jpms+API modules
2	JPMS>>Added class	connected jpms+API modules and their classes, contextual new methods
3	JPMS>>Deleted class	disconnected jpms+API modules and their classes, contextual deleted methods
4	JPMS>>Modified class	Relation information in both # 2 and 3
5	JPMS>>Modified class	Not involved in M2M

- B getObj(A), then getObj is used.
- All the cases directly assigned in variables and used in methods.

A few of the challenges are compromised due to better performance since resolving those introduces other problems (mostly due to static method import) and worsen the outcome. The extracted information is saved into *yaml* template so that any tool can read the data for further purposes.

6.6 Performance evaluation

We compare the outcome of M2M detection and slice generation with the manual collection and measure recall (R) – quantitative correctness of retrieving the change instances, and precision (P) – the accuracy rate among the predicted change instances [89]. First, we run our tool on all 3,647 samples (except 48) (shown in Table 6.1) having structural changes; 2,720 of them contain M2M metric. We measure the precision (P) and recall (R) excluding the 48 training sets. The individual project's performance result is shown in Table 6.1 (P and R columns; MvvmFX has no test M2Ms left). In some cases, performance is compromised for the static import. In the worst case, our tool's precision rate is 97%, and the recall rate is 96%. The highest number of incorrect outcomes is for Vooga (12). For many projects, both the P and R are 100%. Therefore, the DANS property is highly reliable for detecting M2M instances.

For the preliminary investigation of the SSC generation of the M2M instances, we explored the 48 training samples. First, we manually extracted (and saved into YAML files) all the slices of those commits. Then, we evaluate the automated technique's outcome based on the involved entities of a slice with that ground truth. For instance, the discussed slice in Section 6.5 has five entities/instances. If a module itself modifies its dependency with other modules and appears within the other two modules, the instance count would be three; this is true for all other cases. The total number of such instances in each project is shown in Table 6.4; it also shows the classes that are not involved in M2M (nonM2M). The outcome of the automated tool is also presented in the *P* and *R* columns of Table 6.4. For each project, P is from 93 to 100%, and R is from 97 to 100%. Therefore, the DANS properties are also highly reliable for generating the SSCs and can be extracted without compiling each version's code. However, some of the instances are not properly extracted due to SL 3, 4, 5 and 7 in Table 6.2. The lowest precision is for Bach. We have investigated that the module directory structure is unusual for Bach (e.g., the sub-modules are within the *src* folder of the main module), and solving those actually decreases the overall performance significantly. The technique produces the most number of incorrect instances for the AzureSDK (57). The performance is quite general because the investigation is conducted for ten projects with two language frameworks. Our tool cannot process anonymous inner class methods since the *git* API does not provide separate (and structured) information about that.

Table 6.4: Semantic slice data and performance outcome.

Project	Commit	M2Ms	nonM2M	P	R
Hibernate	5	220	17	0.99	1.0
Aion	5	158	19	1.0	.97
Webfx	5	28	3	0.94	1.0
Speedment	5	278	22	1.0	0.98
AzureSDK	5	949	67	0.95	0.99
Atrium	5	135	36	0.99	0.98
Bach	5	48	2	0.93	1.0
Vooga	5	99	9	1.0	1.0
Imgui	5	52	10	1.0	0.97
MvvmFX	3	96	4	0.98	1.0

Bias Testing: To reduce bias in performance testing for the automated SSC generation, we measure the outcome of our proposed tool with 16 unseen commits (two samples from each of eight projects). Those samples are randomly selected, excluding the experimental set (48 commits), and the SSCs are first manually extracted. Then the detected slices are compared against the manual extracted set. The performance outcome of our tool is shown in Table 6.5. The lowest precision rate with this dataset is 91%, and the lowest recall rate is 96%. Therefore, the bias testing also confirms the performance of our tool for semantic change slice generation.

Table 6.5: Bias testing outcome.

Project	Commit	M2Ms	nonM2M	P	R
Aion	2	114	9	0.91	1.0
Webfx	2	651	39	0.96	0.98
Speedment	2	124	19	1.0	0.98
AzureSDK	2	79	9	1.0	1.0
Atrium	2	106	6	1.0	1.0
Bach	2	21	4	0.98	0.99
Vooga	2	50	6	1.0	0.96
Imgui	2	418	85	0.98	0.99

6.7 Related Work

Architectural Change Detection and Design Decision Recovery: Software architecture can be defined into three levels of abstraction according to the convenience of the development team: (i) high-level – where design models or modules are considered, (ii) intermediate level – package, and classes are considered, and (iii) low level – methods and functions are considered. MoJo [230], MoJoFM [245], $A\Delta$ [110], CB [40], C2C [81], and A2A [131] are focused on high-level change detection. In recent times, a few studies focus on recovering architectural design from the release history of software [81, 214, 213]. EVA [182] and ARCADE [213] are excellent tools for recovering a static architecture and detecting changes based on ACDC/MoJo [231] and ARC [81] techniques. However, these tools are explicitly dependent on other techniques to extract models and clusters (they are arbitrary and have no formal limit). Only expert intervention can ensure architectural change detection's accuracy of these metrics, and analysis of thousands of change versions is almost infeasible. Moreover, they cannot recover semantic design [98]. We propose a module-level architectural change detection tool based on the developer's defined modules and thus more concrete and reliable. Our tool also extract semantic change relations on the detected change instances. Hence, our tool and benchmark data can be used to further validate and enhance the existing metrics for architectural change detection (available at [173]).

Change Slicing for Code review: Here, we discuss a few of the most famous works among the existing studies [61, 140, 240] for slicing the committed code. Dias et al. [61] worked on tangled code change information slicing at the fine-grain statement level (i.e., one variable is associated with two lines of code, two files are changed together, the distance between two modified lines in a file, etc.) using AST properties to separate multiple intentions within a single commit. Later, they attempt to cluster the slices based on the pair relation, such as two methods are only refactored, two classes within the same package, etc. Li et al. [140] separate all types of atomic changes with the AST algorithm of a set of related commits in the version history for commit porting. Wang et al. [240] developed a more intelligent tool for decomposing changed code within a commit using AST parsing and machine learning. They cluster the code based on the class-level, method-level, field-level, and statement-level changes. Then they rank those changes considering the number of referenced variables for the code reviewers. Several studies have been enhancing atomic code change slicing works [138, 238]. However, these studies do not focus on architectural semantics and relations. Therefore, decomposing architectural change of a commit would enhance these techniques for reviewing more complex scenarios for architecture intensive systems. To reduce the gap, we attempt to generate architectural change slices for design review.

6.8 Conclusion and Future Work

In this study, we present our initial observation on the impact of DANS properties to develop a design review tool that detects and semantically slices the architectural change instances of a commit. Performance evaluation with ten open-source projects proves that this process produces reliable outcomes while the technique is lightweight. We will cover more concerns in the tool, such as extracting indirectly impacted methods that invoke methods in Table 6.3. We believe that the directory-based challenges that we have discussed in this study will persist in the AST-based approaches (since AST nodes are generated from the directory structure information). Furthermore, we will evaluate the performance of various types of slices presented in this table. Semantic change information presented in Table 6.3 can be utilized to generate more understandable code descriptions [113] and can be mapped with multiple intentions if they exist within a commit. Our tool would be useful for a number of empirical studies besides assisting design review, such as the effectiveness measures of the existing design decision recovery approaches, determining architectural change types, developers profile buildup based on design changes, design debt and change impact analysis, release note generation, design change versioning scheme, etc. Dataset and the script of the tool are available for further advancement.

Future Work: Our main objective is to assist in semantic design review. Semantic design recovery and semantic design summary (for each release or milestone) generations are the essential steps for that. For that purpose, we plan to investigate concept generation by mapping with the commit description and code identifiers associated with each of the DANS properties within the change instances with our proposed tool. Semantic software design is involved in the concept/meaning of software features/requirements associating design logics (including architecture) and implementation in the programming languages [98]. That is why semantic slicing is essential for semantic design recovery. The architectural change relations and concept generation would facilitate to advance of our planned empirical study on semantic design recovery and summary generation. We also explore separating tangled commits (having M2M) with DANS properties, which is also required for the efficient design review tool. Moreover, our proposed tool needs to be enhanced in string pattern matching as we have observed that in some cases, almost identical directory structures are falsely identified (completely ignoring them reduces the tool performance significantly). To handle this situation, we will experiment with the *Context Triggered Piecewise Hash* [122] mechanism.

The experiment shows that our proposed automated techniques are promising that will facilitate the acceleration of change classification and automated change summary generation. Therefore, in the next study in Chapter 7, we will utilize these techniques for enhancing change classification.

Chapter 7

Architectural Change Categorization leveraging Structural Change Properties of Source Code

In this chapter, we enhance the previous techniques (i.e., Chapters 5, and 6) for categorizing the architectural change revisions with code properties for handling the message triad (empty, meaningless, and tangled) reliably. However, the relationships between code and their architecture knowledge are hard to reveal [34]. The issue is a lack of overview of the structure of the system, linking to the source code and program file. To improve this scenario in message triad handling, through manual analysis of the source code of the high-level architectural change samples, we extract 17 semantic change relations (SSC) from code operations (leveraging the approach proposed in Chapter 6). We have explored various classification models with these SSCs (as are explored in Chapters 4 and 5). Then, we explored various classification models combining these SSCs with concept tokens. Finally, we have proposed approaches to handle the challenges of commit triad – tangled changes, ambiguous messages, and non-informative or empty message descriptions. Thus, our proposed models are promising to apply in real-world applications such as release change logs or release note generation.

The rest of the chapter is as follows. Section 7.1 presents a brief overview of our study. Section 7.2 discusses the background of our study. Section 7.3 presents our dataset of study; Section 7.4 presents SSC extraction; Section 7.5 discusses our proposed technique with SSC; Section 7.6 presents our combined models; Section 7.7 presents performance evaluation with message triad samples; Section 7.8 discusses related work,; and Section 7.9 conclude this study.

7.1 Introduction

Software development and maintenance teams conduct various tasks such as bug fixing and feature enhancement regularly [226]. Source code changes in these activities are integrated into the original codebase of the target software. In this process, a commit records changes to one or more files in a codebase revision, and a commit message briefly describes the changes [57]. However, during a commit merging, development and maintenance teams need to provide some more detailed information about changes in commit messages. This detailed information can help other developers to understand this change and support other software maintenance tasks, such as code review, bug triaging, traceability link recovery, and change impact analysis [246]. Hence, these commit messages, especially those accurate, complete, and labeled, are essential to change comprehension and software maintenance during software evolution. There are two aspects that development and maintenance teams are usually concerned with for a commit, i.e., *what* information and *why* information [206, 216]. The *what* information refers to the changes involved in an incremental change, while the *why* information describes the motivation and context behind the changes (such as performance improvement or refactoring). According to the literature, the quality attributes of commit messages can be divided into adequacy, conciseness, and expressiveness [45].

But in practice, although development and maintenance teams are eager to learn more information from commit messages regarding these three attributes, some original commit messages generated by developers are of less use for acquiring useful information from them [239]. Moreover, the development and maintenance team finds it difficult to capture the source code context effectively from the original commit messages since they do not meet the demand of adequacy both in *what* information and *why* information in most of the cases. Empirical studies found *why* information is invaluable to support maintenance work. Lack of it may result in difficulty in understanding the purpose or motivation of the changed code. To capture the *why* aspect properly, a commit change needs to be categorized based on the causes and purposes. Because many commits messages do not provide enough context to understand it behind a change. This is due to the fact that developers are reluctant to tag or label the commit. Empirical studies reported that most of the commits in open source software are not tagged or labeled. According to the literature, labelling change category is ambiguous even by the developers [168, 99, 80]. Herzig et al. [97] found that the developers misclassify 33.8% of the bug reports in the issue tracking system. Even the reports that 14% messages are empty and 10% messages are meaningless [148, 239] indicate that they do not write the messages properly. Moreover, categorical information is mandatory for writing release notes [178]. Therefore, efficient techniques to categorize the change commits are important for the development and maintenance team.

That said, automated change categorization can be applied in many tasks – tagging the commits in the version control systems (VCS), estimating the quality of changes, making decisions for backporting, characterizing architectural evolution, optimizing development process, forming developer and reviewer expertise, and so on [246, 172]. According to the existing studies, textual properties (extracted from commit messages) are the most fruitful for change grouping [171, 94, 101, 160, 104]. Some of the studies also explored fine-grained level code properties such as change densities to enhance the classification combined with the commit message. However, many commit messages are empty, non-informative, tangled, and ambiguous [112, 148, 73, 240]. Most of the studies skipped 20-28% commits from the collected dataset having inappropriate messages. In the existing approaches, the above-mentioned challenges (as discussed in Table 8.1) remain inexplicable. For proper DDARTS generation, these contexts should be handled reliably rather than adopting random guesses.

Source code properties are the most reliable source of information [50]. Meaningful code properties can fill the gap of natural text descriptions. As can be seen in Fig. 8.2, only the critical and major impactful information is included in the changelogs. Therefore, it is worthwhile to develop a model with code properties that are not too fine-grained but provide a meaningful way of understanding major perspectives (as described in the *Motivation* section). Overall, we explore 17 SSCs of the dataset of module-level architectural change instances from [173]. The meaning of those SSCs are shown in Table 6.3.

RQ1: *How can SSC predict module-level change types?* If SSC properties can predict change groups significantly, then SSC would enhance on-demand design artifacts generation approaches.

RQ2: *How SSC properties with the texts are performing for module level change classification?* This is important to know whether SSC has a positive or negative impact when combined with the textual properties. SSC can be used to generate descriptive change summaries efficiently embedding with texts if SSC enhance the change grouping.

RQ3: *How can SSC resolve issues in non-informative, ambiguous, and tangled description in change type classification?* Research found that around 14% of commit messages in open source projects is empty. Moreover, there is a significant presence of non-informative and ambiguous descriptions of software artifacts. Multiple intentions are also included in the messages. Answering this research question will help to employ code properties for handling them.

In this study, we have annotated and experimented with 2,697 DIS commits from eight open-source projects [173], which is comparatively a large collection in the context of architectural changes. We enhance an existing tool [173] and extract total 17 semantic change relations from code changes. We have also extended the concept tokens list of ArchiNet [171] with the new dataset. ① With these SSCs and concept tokens, we explore various change grouping models considering the challenges. Our experimental outcomes with multiple configurations prove that SSCs can mitigate the issues of ambiguous, non-informative, and tangled commit messages in a promising way. SSC properties alone can produce 52% F1 scores, while the combined model with the concept tokens can extract 86% change categories correctly within the Recall-Hit@2 rank [201]. Overall contribution of this study are:

- Construct a benchmark dataset by annotating into four categories that contains a substantial number of high-level architectural change samples.
- Experiment the categorization model with SSCs.
- Propose combined models for classification.
- Propose a technique to predict tangled commits.

7.2 Background

7.2.1 Architectural Change:

Software architecture/design may be modified intentionally or unintentionally during the development and maintenance life-cycles. Software architecture modification is considered as of [192] – configuration change

[72, 87], source-code layers (i.e., directories, package structures, and location of code files within the directories) changes [151], design model change (i.e., UML diagram) [245, 152, 81], architectural document in natural language [63, 115], and code component change operations (i.e., addition, deletion, moving, and merging components) [246, 213]. Software architecture is studied at three abstract levels: high level, intermediate level, and low level. In this study, we focus on the commits having module/system (higher) level changes. A module can be a sub-system, 3rd party library, and cluster of packages [41]. Overall, the change commits contain additions, removals, and moves of implementation-level entities from one module to another. This also includes additions and removals of modules themselves. Moreover, include and symbol dependency changes [151, 87] are also architectural changes. Our consideration of these metrics as architectural changes are based on a number of existing studies [47, 81, 130, 213, 151, 245].

7.2.2 Architectural Change Categories:

Architectural changes can be grouped on focusing various perspectives [246, 63, 169]. In this study, we consider change grouping based on the development and maintenance activities [246]. *Adaptive (A):* This change is a reflection [246, 143] of system portability, adapting to a new environment or a new platform. *Corrective (Cr):* This change refers to defect repair, and the errors in specification, design and implementation. *Preventive (PV):* Preventive change [170, 246] refers to actionable means to prevent, retard, or remediate code decay. This is related to inappropriate architecture that does not support the changes or abstractions required for the system. *Perfective (PF):* Perfective changes are the most common and inherent in development activities. These changes mainly focus on adding new features or requirements changes [226, 246, 63] including improving processing efficiency and enhancing the performance of the software.

7.2.3 Important Definitions:

This section presents some definitions related to change commit messages.

- *Non-informative (NI):* A message is non-informative if no particular information is presented about the specific reason for the change. Sample 3 in Table 8.1 is an NI message.
- *Ambiguous (AM):* A message is ambiguous if a long discussion is provided and can have multiple meanings or code change does not strongly reflect the description. Sample 2 in Table 8.1 is an AM description.
- *Message with multiple concepts (Tangled):* A message which has multiple unrelated change information is called a tangled message. We call a tangled commit as archTangled if it contains multiple intentions having architectural involvement. Sample 3 in Table 8.1 is an archTangled commit.
- *Concept:* A set of words that represent a specific meaning collectively. Concept words *"add" and "support"* in a sentence indicate new feature, whereas *"add", "support", and "down"* indicate flaw fixing.

Table 7.1: Total number of samples in each type (nontangled)

	Perfective	**Preventive**	**Corrective**	**Adaptive**
Commits	996	1155	168	102

7.3 Dataset Preparation

We collect the module-level architectural change commits of Mondal et al. [173]. This dataset are formed from 10 open source projects, and contain 2,720 architectural change commits. The commits are within the period from 2017 to until December 2019. However, we exclude two projects not having enough samples to balanced split (for all four types). Most of the projects are commercially important in various domains. Thus, our experimental dataset contains 2,697 DIS from eight projects. The Dataset is shown in Table 7.2.

7.3.1 Golden Set Construction for Experiment:

We divide our collected dataset into four parts: (i) samples having tangled commits, (ii) training set with non-tangled commits, (iii) natural test set, and (iv) test set with NI and AM samples. Unbiased test sample creation is critical to reducing the biasness. As can be seen from Fig. 7.1, the dataset contains an unbalanced number of classes. This is problematic with the multi-class classification experiment. Therefore, we also prepared a dataset with the Stratified random sampling (downsample the majority classes to an almost equal number of samples from each group) [215]. We consider 168 (size of the corrective class) as the minimum number of samples as the base size for the preventive and perfective.

7.3.2 Change Type Annotation:

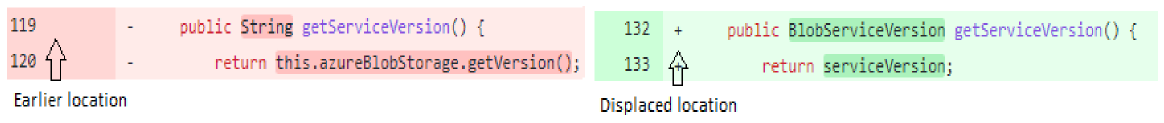
Our collection of change samples is large, considering the manual analysis perspective. This is the most critical phase of our study since it is subject to human bias and inconsistent description. We annotate the commits into four groups based on the existing studies [63, 172, 171] as well adopting the knowledge from more than 150 categorical change descriptions of the AzureSDK project (as shown in Fig. 8.2). Some other specific challenges of annotation are: (i) insufficient, and ambiguous words, (ii) implicit and tacit intention in the explanation, (iii) different meaning of glosses of terms in software context and natural language, (iv) multiple intentions in a single message but a few of them are architectural, and (v) noisy, irrelevant and non-separable text. Two authors independently annotated the samples then resolved the conflicted samples with rigorous analysis and discussion. We analyze the commit messages, comments related to changed code, issue/feature tracker, bug tracker, and source code to mitigate the challenges for determining a change intention. For some samples, we also contacted the developers for clarification (a tiny portion). The dataset creation, annotation and conflict resolving process take three months per person.

Table 7.2: Selected projects and DIS commits (until 2020).

Project	**All**	**2017-2020**	**DIS**	**Domain**
Aion[18]	4718	1064	863	A multi-tier efficient blockchain network
Webfx[244]	3770	778	563	Providing a web port of JavaFx to JavaScript
Speedment[222]	4483	243	222	Creates a Java representation of the data model from SQL
AzureSDK[76]	15,180	276	244	Azure SDK for java
Atrium(Kotlin)[20]	1988	379	210	Assertion library for Kotlin with support for JVM, JS and Android
Bach[43]	2114	365	145	Java Shell Builder
Vooga[235]	1210	447	416	Game development engine
Imgui(Kotlin)[106]	1703	35	34	Game engine and 3D application framework
	Total	3587	2697	

7.4 SSC Extraction

Meaningful properties extraction focusing on the design impacts is a heavyweight process (either requires human intervention or building each commit). It can even take a few days to process the AST for a single version of a medium or large-scale project [150], and thus not deployable frequently (a release may contain hundreds or thousands of commits). Considering the feasibility, we explore the semantic change relations (SSC) with a lightweight tool proposed by Mondal et al. [173]. It is lightweight because it can extract the properties by processing directory and naming structure along with the code change information provided by the VCS APIs without compiling and AST processing. However, the tool has lacked in extracting information on method and class moving that are important in terms of design impact. Therefore, we extend the tool with the clone detection technique. We consider the 14 SSCs from the study of Mondal et al [173]. We have enhanced the tool to detect method moving (*MVM*), constructors (*CMD*) and classes (*MVC*) from one place to another. For that purpose, we employed the heuristic clone detection technique of CloneWorks [225] tool. These technique has more than 90% P and R for Type 1 Type 2 clone and thus reliable. However, we get significantly better outcome with the threshold 0.50 compared to 0.70 for clone detection. For example, 0.70 threshold miss the method move like Fig. 7.1. We found around 99% accuracy within the training set of Mondal et al. [173]. Git [88] and PyDriller [5] provide moving method as addition and deletion operation. Those methods can be moved with change or renamed (which is sometimes appear as method overloading).



```
119  -      public String getServiceVersion() {
120  -          return this.azureBlobStorage.getVersion();
```

```
132  +      public BlobServiceVersion getServiceVersion() {
133  +          return serviceVersion;
```

Figure 7.1: Method displaced with content changes

For example, such a moving is shown in Fig. 7.1 of a commit [1]. Therefore, we have additional three SSCs than Mondal et al.

We consider method moving and class moving in the following ways –

- Method location changed in the same class (as shown in Fig. 7.1 of BlobClientBase.java class).
- Method moved from one class to another.
- Class moving–methods of deleted class and added class are same.
- Merging–one class is added from the contents of 2 deleted classes.
- Splitting–2 classes are added from the contents of a deleted class.
- Partial separation of code–methods from a few classes are deleted but mostly appeared in the new classes.

We consider a new class as the moved class if 30% of the methods are clones (better than 50% and 70%). Because, some constructors having signature of that class can be added in the new class.

7.4.1 Relation Between Change Purposes and SSCs

The presence of the 17 SSCs among the four change groups is shown in Table 7.3. The left side in each column represents all commits, and the right side represents balanced commits in the training and test sets. Gray colored cell values represent the highest rank (in some cases, all the closest presence) based on the percentage of presence. From the table, we observe that most of the highest distributions of delete and disconnect (dSSCs) relations are for the preventive (PV). Perfective (PF) and adaptive (A) have approximate distribution patterns (except in some dSSCs). Practitioners can treat these two groups as a single one since fewer adaptive samples are found in practice as a separate group. Corrective (CR) has mostly MCC and MCAC SSCs. However, all 17 SSCs are present in the PV group. In contrast, the module config remove (*MMD*) is not present in the PF, module config add (*MA*), delete (*MD*) and *MMD* are not present in the CR group. The *MA* and *MVM* are not present in the A category. Surprisingly, class move (*MVC*) exists for all the groups. However, *MVC* and *MVM* are unexpected operations in the PF group. We investigated the *MVC* in the PF category and found some irregularities. For example, VCS APIs return the renaming of *EventProcessorBlobPartitionManagerSample* class in commit[2] as class add and delete, and our technique

[1]https://tinyurl.com/2p9sebb2
[2]https://tinyurl.com/mwjmknve

Table 7.3: Semantic operations, their presence (% of commits) in different groups of changes and classification impacts.

#	SSC	Meaning	Perfect	Prevent	Correct	Adapt	Grouping F1 (Excluding)	Inclusion Impact(base 42)
1	MA	Module file addition	6 5	4 3	∼ ∼	4 4	40	(+)
2	MD	Module file deletion	∼ ∼	3 3	∼ ∼	∼ ∼	39	(+)
3	CA	New class addition	37 33	23 21	12 12	47 47	46	(-)
4	CD	Class deletion	3 4	16 15	2 1	12	39	(+)
5	MVC	Move class	3 4	14 16	1 1	11 10	38	(+)
6	MCNM	Modify class+new method+cross module dependency	18 17	10 9	12 12	25 26	42	(∼)
7	MCDM	Modify class+delete method+remove module dependency	2 1	6 5	1 1	7 7	39	(+)
8	MCNMA	Modify class+new method+new lib connection	21 25	11 18	14 14	22 21	36	(+)
9	MCDMA	Modify class+delete method+removing lib connection	2 1	6 5	4 1	5 7	39	(+)
10	MVM	Move method	2 1	4 3	2 2	∼ ∼	39	(+)
11	CMD	Modified constructor	4 4	5 5	3 3	5 5	40	(+)
12	MCC	Modified class with cross module connection	58 57	50 45	48 48	56 56	37	(+)
13	MCD	Modified class, cross module disconnection	22 18	51 50	14 14	30 31	32	(+)
14	MCAC	Modified class with lib connection	57 68	40 49	54 54	72 71	44.2	(-)
15	MCAD	Modified class with lib disconnection	19 22	43 49	22 22	28 29	33	(+)
16	MMC	Modified config with cross module connection	11 10	14 13	4 4	20 20	41	(∼)
17	MMD	Modified config, cross module disconnection	3 6	11 11	∼ ∼	5 5	37	(+)

Table 7.4: Rank of SSCs based on Pearson correlation analysis w.r.t categories in Weka.

#	SSC	Worth value
1	MODIFY_DISCONNECT	0.2433
2	MODIFY_API_DISCONNECT	0.2276
3	CLASS_MOVE	0.1764
4	CLASS_DELETE	0.1742
5	MODIFY_API_CONNECT	0.1381
6	CLASS_ADD	0.1201
7	MO_DISCONNECT	0.1189
8	MODIFY_NEW_METHOD	0.1069
9	MODIFY_NEW_API_METHOD	0.1064
10	DELETE_MO	0.0927
11	MODIFY_DELETE_METHOD	0.0914
12	MODIFY_DELETE_API_METHOD	0.0835
13	METHOD_MOVE	0.0716
14	MODIFY_CONNECT	0.0517
15	MO_CONNECT	0.0517
16	CONSTRUCTOR_MODIFY	0.049

Table 7.5: Correlation $p-$values considering the perfective and corrective categories

SSC	$p-$**value**
CONSTRUCTOR_MODIFY	0.7059
DELETE_MO	NA
MODIFY_DELETE_API_METHOD	0.9746
MODIFY_API_DISCONNECT	0.676
MODIFY_DISCONNECT	0.3363
MODIFY_DELETE_METHOD	0.3246
CLASS_DELETE	0.2914
MODIFY_NEW_METHOD	0.2596
NON_M2M	0.248
CLASS_MOVE	0.1673
MODIFY_CONNECT	0.1662
METHOD_MOVE	0.1622
MO_CONNECT	0.1088
MODIFY_API_CONNECT	0.0461
MODIFY_NEW_API_METHOD	0.0284
NEW_MO	0.022
MO_DISCONNECT	0.0018
CLASS_ADD	0.0003

Table 7.6: Correlation $p-$values considering the perfective and preventive categories

SSC	$p-$**value**
MO_CONNECT	0.4895
MO_DISCONNECT	0.4431
NEW_MO	0.43
CONSTRUCTOR_MODIFY	0.3172
MODIFY_DELETE_API_METHOD	0.2277
MODIFY_CONNECT	0.1858
NON_M2M	0.1689
DELETE_MO	0.09
CLASS_ADD	0.0875
MODIFY_NEW_API_METHOD	0.0759
MODIFY_NEW_METHOD	0.0483
METHOD_MOVE	0.0276
CLASS_DELETE	0.0088
MODIFY_DELETE_METHOD	0.0087
MODIFY_API_CONNECT	0.0043
CLASS_MOVE	0.0002
MODIFY_API_DISCONNECT	2.52E-05
MODIFY_DISCONNECT	5.23E-07

Table 7.7: Correlation $p-$values considering the perfective and adaptive categories

SSC	$p-$**value**
CONSTRUCTOR_MODIFY	0.9376
DELETE_MO	NA
METHOD_MOVE	NA
MODIFY_CONNECT	0.7281
NEW_MO	0.7249
NON_M2M	0.5802
MODIFY_API_CONNECT	0.49
MODIFY_NEW_API_METHOD	0.3218
MODIFY_DELETE_API_METHOD	0.3135
MODIFY_API_DISCONNECT	0.3133
MO_DISCONNECT	0.3122
MODIFY_NEW_METHOD	0.2415
CLASS_DELETE	0.1454
CLASS_MOVE	0.0827
MODIFY_DISCONNECT	0.053
MO_CONNECT	0.0336
CLASS_ADD	0.0131
MODIFY_DELETE_METHOD	0.0043

detects it as the MVC operation. As for the perfective group, we observe that the design is refactored as part of the new method, segment, or class addition for implementing a new feature or a feature improvement. We found at least five such commits in AzureSDK. Therefore, most of these properties directly or indirectly explain some design decisions and their potential impacts. The impact of each SSC for predicting a group is shown in the last two columns of Table 7.3. This outcome is based on the ArchiNet [171] strategy (Eqn (7.1)) that produces 42% F1 with all SSCs. This is an alternative to the principal component [250] and correlation-coefficient [96] analysis (to avoid broader explanation). We notice that CA and $MCAC$ have significantly negative outcomes for predicting the change groups. We also got the most significant positive impacts with $MCNMA$, MCD, and $MCAD$ for the PF and PV category.

We also analyze the associativity of the SSCs with different categories leveraging the correlation analysis. The ranking of the SSCs based on Pearson Correlation analysis considering all categories in Weka is shown in Fig. 7.4. Moreover, the $p-values$ of Pearson correlation analysis of SSCs to pairwise categories are shown in Tables 7.5, 7.6, and 7.7. These $p-values$ are calculated based on the presence of the SSCs. Our analysis is based on the hypothesis that the correlation of the SSCs follows different patterns in the pair of categories. In this context, if the p-value is low (generally less than 0.05), then the pattern of the categories is statistically significant. The gray color values in these tables indicate a statistically significant difference in the respective categorical pair. It means that those SSCs can significantly differentiate the respective categories. However, only a few SSCs are significantly different between the perfective and adaptive categories meaning distinguishing them would be difficult. Thus, the SSCs provide the most reliable explanation irrespective of project contexts and textual description.

7.5 SSC properties for Change Categorization

7.5.1 SSC Rule-based Change Type Determination:

Multiple SSCs may exist in many change tasks, such as the commit in Table 8.2 has seven types of SSC. Therefore, co-occurred patterns (or associative occurrence) can indicate more specific information than their individual presence. Their dependency relations or associations in a task may also be helpful for a proper summary generation. Therefore, we explore the association rules [137] of SSCs from the training data. We utilize the support, confidence, and leverage for selecting the highest ranked rules. An association rule is presented as $\langle \subseteq SSC_s \rangle \Rightarrow C$. The left side of $\Rightarrow$ is called antecedents which are one or more operations, and the right side is called consequent, which is a change class here. For example, in a commit, such a rule can be $\langle CA, MMC \rangle \Rightarrow perfective$; which means operations CA and MMC mostly occurred together when a perfective change happened. We mined such rules based on the *Apriori* algorithm [66] with python *Mlxtend* library [204]. We consider the top rules after mining the rules having *Confidence*$>$ 0.50, *Lift* $\geq$ 1, and *Support* $\geq$ 0.1. However, rule extraction is problematic for four types at a time using the Mlxtend library. That is why we extracted association rules based on samples pair strategy – $PF + PV$ and $CR + AD$. If we consider all the fours at a time, rules mining for a few remain empty, while for a few, hundreds of rules are generated. We consider only the top 15 rules. However, in this process, many rules are present among multiple types. We assign such rules to a relevant change type that has the highest combined values of the SSCs according to the distribution from Table 6.3. For example, the rule $\{CD, MA\}$ exists both in the PV and CR categories but has the highest distribution within the PV, and we remove it from the CR category. Finally, we consider a total of 40 rules. In this phase, we predict a change type based on the presence of the maximum number of rules from the 40 rulesets for a relevant type. The F1 scores of the rule-based classification is shown with the *R-SSC* graph in Fig. 7.2. Overall, the F1 score is 53% with the test sets, meaning that the SSC properties are promising for change grouping with association rule mining strategy. Since rule mining requires careful strategy along with human intervention, we avoid the 10-fold cross-validation evaluation. SSCs rules can be employed for design artifacts generation- changelogs, design-centric release notes, and design document recovery.

7.5.2 SSC Strength-based Change Type Determination:

Strength can be measured using *TF-IDF* [190], but Mondal et al. [171] found significantly lower outcome than ArchiNet strategy. Therefore, we explore the training and classification strategy of ArchiNet [171] with the SSCs due to the better performance than DNL and RF. It generates a training model using the normalized special frequency value f_{vi} of each token. Here, t is replaced with SSCs. In the special frequency (f_t), the value of a token (t) in a commit message with one or multiple presences is always 1.

$$For\ token\ t,\ f_{vi} = \frac{f_t}{N_i}, \quad weight\ w_{ti} = \frac{f_{vi}}{\sum_{i=1}^{4}(f_{vi})} \tag{7.1}$$

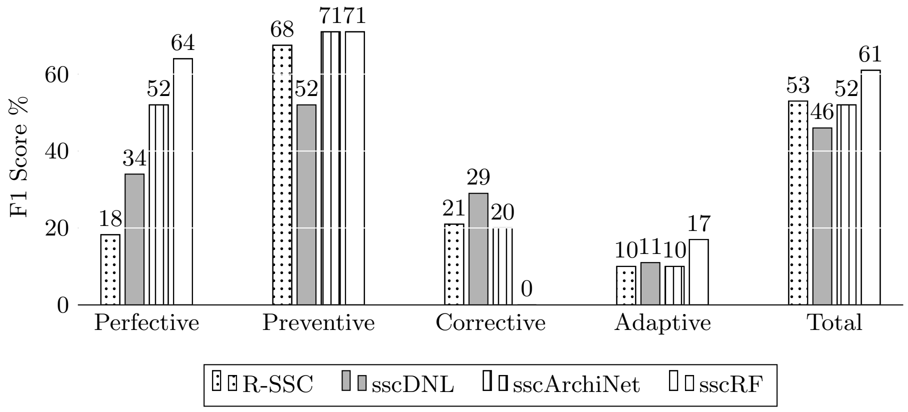


Figure 7.2: F1 score of semantic operations centric models.

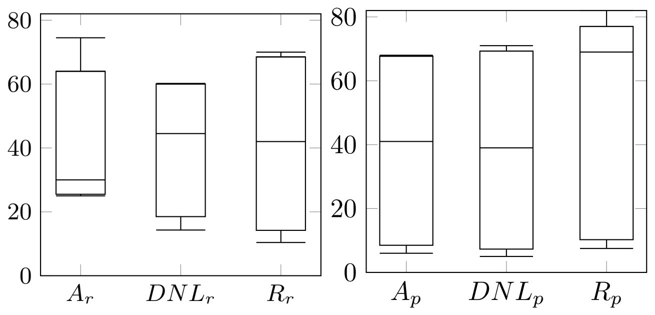


Figure 7.3: Range of Recall (r) and Precision (p) rate

Then, a change group is predicted by measuring the maximum value of summation of all token's weight (w_{ti}) for a relevant class. With this model, the SSC can predict with a 52% F1 score in the best case scenario (excluding CA and $MCAC$). More than 50% scores for multi-classes mean that the model produces a more meaningful outcome than a random guess. This is 19 points (58%) increase than the abstract operations reported by Mondal et al. [171]. The F1 scores for the individual category are shown with the *sscArchiNet* graph in Fig. 7.2. The range of precision rate is 6.2 - 68%, and the recall rate is 25 to 74.4% which are shown with the Box plot in Fig. 7.3. The SSC model produces the lowest outcome for the adaptive. However, the accurate change type extraction is 65% (individual type's range is 28-97%) with the Hit@2 [201] for the Top-2 rank metric. Hit@K is widely used for measuring the significance of suggested information in SE research. The performance is still lower than the concept tokens but can be used to address some challenges for the message triad.

7.5.3 SSCs As Features for Machine Learning Techniques:

We also explore the Deep Neural Learning (DNL) [71] and Random Forest (RF) [146] with the SSC features. We adopt the same configuration of [171]. The classification outcome of DNL and RF with the training and test set is presented as *sscDNL* and *sscRF* graph in Fig. 7.2. F1 scores of sscRF is 61%, but cannot predict the corrective change. It is 46% for sscDNL, which is lower than the rule-based *R-SSC* and sscArchiNet

Table 7.8: Number of samples with no ArchiNet tokens.

Azure	Atrium	SpeedM	Bach	Vooga	WebFX	Aion	Imgui
6	2	10	15	23	16	90	4

strategies. According to the literature, a large scale training dataset might produce better a DNL model [71]. Therefore, SSCs properties are also adaptable to deep learning and machine learning models. The ranges of precision and recall are mostly similar to other models (except the sscRF).

Message for RQ1: Design impactful changes have various crucial semantic structural change (SSC) properties at the code level that have good potentials for grouping them with certain explanation. SSC-based models can produce up to 53% F1 scores.

7.6 Combined Models for Change Type Determination: SSC and Concept

7.6.1 ArchiNet Concept-tokens Expansion:

Concept extraction from the developer's description is valuable for understanding a change. It requires a huge budget and human efforts to experiment all in the field of software development artifacts. However, Mondal et al. [171] extracted around 237 concept tokens that indicate change intentions. We have employed their classification model with the concept tokens referred to as ArchiNet for our module-level change dataset. The ArchiNet has shown significantly better performance than DPLSA [255], LLDA [37], SemiLDA [80], DNL [71], and RF [146]. However, many commit messages in our dataset do not contain any defined concept tokens because ArchiNet are extracted from different projects. Project-wise statistics are presented in Table 7.8. In that cases, no logical classification is possible (predicted weights of all types will be 0) with ArchiNet. Therefore, we have extended the token list following the concepts in ArchiNet and modified the ArchiNet model from the training set. Still, we cannot extract any reasonable tokens from 12 commits like sample 3 in Table 8.1. Within the training set, the number of samples that have no ArchiNet tokens for the perfective, preventive, and adaptive are 33, 53 and 1, respectively. This result reveals the vacuum of explainability of the textual techniques for change analysis. Total, we got 56 new tokens with their respective strengths. Experimental outcomes with this new model are almost similar for the ArchiNet dataset and our dataset (only one-point variations).

7.6.2 Concept-tokens and SSCs with ArchiNet:

We explore a combined model utilizing the concept tokens of the ArchiNet and the 17 SSCs. The training model in Eqn (7.1) with concept tokens and SSC is generated separately. Then the classification model

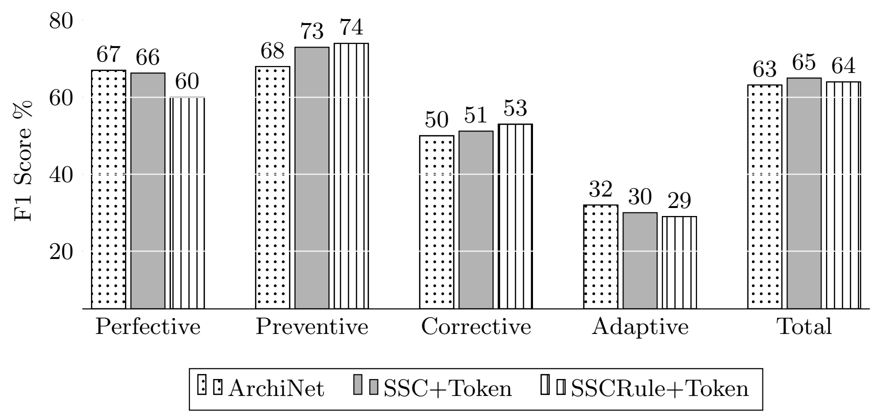


Figure 7.4: F1 score of ArchiNet, Mix-model with SSC, and Mix-model1 with SSC rules.

Table 7.9: Performance (F1 scores) of DNL and RF

Classifier	**Perfective**	**Preventive**	**Corrective**	**Adaptive**	**All**
txtRF	72	78.5	37	7	70
ssc+txtRF	72.4	79	37.5	7	70.5
ctDNL	57	59	48	39	66
ssc+ctDNL	59	69	47	38	69

considers the average value of their total predicted strengths for argument maximization for indicating a change group. If no concept token from the ArchiNet model exists, then the combined model considers the SSC strength. We found 33 such samples in the test set. However, at least one semantic operation is present within a change commit. Therefore, this provides an interpretation of change intention to some extent when there is no informative commit message. We call this combined model semantic operation-centric ArchiNet (sscArchiNet). The outcome of the combined model is shown in Fig. 7.4. We also explore the combined model with concept tokens and SSC association rules. The outcome is promising, as shown in Fig. 7.4. sscArchiNet's F1 scores are 2 points better than textual technique (rule-based model is 1 point better). However, the Recall rate range with Hit@2 [201] is 80-91% (total 86%).

7.6.3 Concept-tokens and SSCs with Machine Learning:

We compared the classification performance of DNL and Random Forest (RF) with and without SSC with concept tokens. The outcomes with the same configuration as of Mondal et al. [171] are shown in Table 7.9. SSC improves the performance of both of the classifiers.

Takeaway message for RQ2: SSC properties enhance the text-based classification techniques significantly for change group determination. It also fills the vacuum of the explainability of the most promising textual approach.

Table 7.10: Performance for tangled commits

TModel	TangleP	Perfective	Preventive	Corrective	Adaptive
R-SSC	54	75	65	50	44
sscANet	60	79	65	63	35

7.7 Predicting Change Types in NI, AM and *archTangled* Commits

7.7.1 Change Types of *archTangled* Commits:

In our collected dataset, we found at least 99 commits that have multiple intentions for architectural changes. As the SSC+tokens with the ArchiNet model shows better performance, we explore it for predicting multiple types within the *archTangled* commits. First, we experiment with the range value compared to the highest measured weight of the four change types from the trained model for deciding multiple types. If V is the highest measured value among the four classes, then all the classes i with weight $(W_m(C_i))$ bellow or equal to the threshold R would be the probable types as follows: $V - W_m(C_i) \leq R$. The impact of threshold values (R) for predicting a tangled commit is shown in Fig. 7.5. The best precision rate reached to 40% when $R = 0.5$ in this equation. It indicates that the intelligent threshold value processing mechanism has a good potential for handling the archTangled commits. We further explore the uniform range values on the normalized outcome. In this process, we transform the predicted values of the four classes in such a way that the summation of the strengths of all is 1. Uniform range (R_u) considers the uniform probability of being a type from a calculated weight among the four classes as 1/4=0.25. If ω is the sum of all classes weights, then we consider all the classes if it satisfies the following condition:

$$\frac{\omega}{W_m(C_i)} \geq R_u \tag{7.2}$$

If we consider the value $R_u = 0.25$ in Eqn (7.2), then the precision rate of determining a tangled commit is 60% with sscArchiNet. Precision, recall and F1 scores for the all change categories among the 60% tangled commits are 68, 67.8 and 68%. After normalization, we also consider other threshold values (i.e., 0.15, 0.30, 0.45, etc.) and found not as promising as the uniform value. For sscRule+tokens, the precision rate is 54%. Precision, recall and F1 scores of the relevant categories among (54%) them are 64, 64.15 and 64%. All the outcomes are shown in Table 7.10. Therefore, uniform threshold with sscArchiNet shows a promising direction for change type determination in the tangled commits. We will utilize this model for generating the change description and changelogs.

7.7.2 Handling NI and AM commits:

We rigorously analyze and define non-informative (NI) and ambiguous (AM) commit messages within the balanced dataset. Defining NI messages is straightforward. The question is how we define AM messages.

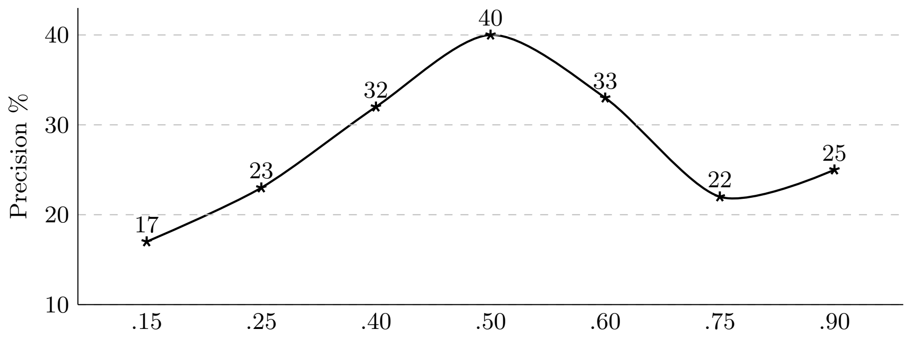


Figure 7.5: Precision rate of tangled commits detection and classification for various threshold values (R).

Inspired by the recent study of Ezzini et al. [73], we designate a message as the coordination ambiguity if it contains *and, or, also, additionally, &, ?*, and multiple sentences with unrelated topics (as shown in Table 8.1). Existence of *and* does not always indicate ambiguity such as - *Add Mongo* ***and*** *the factory method for it.* We do not include such cases. We designated 33 commit messages as the NI and AM messages. SSC properties with the ArchiNet strategy alone can produce 42.5% F1 scores (individual classes have a range of 24 to 64%). SSC with tokens can produce 52% F1 scores (individual classes have a range of 40 to 62%). In comparison, rules and tokens can produce 49% F1 scores. This outcome is below the average of the normal commit samples meaning that grouping them is more challenging. Therefore, SSCs and concept tokens combined are promising for handling the NI and AM messages. This finding is a starting point for experimenting with more robust techniques with a large collection of samples. For example, the names of the involved modules and classes and their respective SSCs can be utilized with our models to generate proper messages and change intentions.

7.8 Related work

Here, we discuss the most prominent and relevant studies.

7.8.1 Software Change Classification using Textual Features

A few of the studies specifically focused on grouping architectural changes (class-level) based on the intentions/development and maintenance activities [246]. Mondal et al. [172] explored keywords and abstract change operations [172, 171]. According to their experiments, the abstract operations (add, delete, modifications, and renaming) are not promising, whereas concept tokens have good potential for change grouping. Our study explored more intuitive code change properties and extended the concept tokens for grouping module-level architectural changes. A handful of studies focused on grouping changes based on large changes (including non-coding files), concerns and tactics [101, 166, 83], which are aligned with the design impactful changes but in a different perspective. We believe that the SSC properties can be leveraged to enhance these techniques. Several studies are available for usual and fine-grained code change classification. Some of them used discriminating set of keywords lists [168, 168, 100, 99, 160, 94, 80, 255, 86]. Most of the techniques that only consider keywords cannot predict all change commits, also not promising in the context of architectural changes [172]. The percentages of skipped samples are 20-28% of the experimental datasets. Several studies also leveraged source code properties along with the keywords (in some cases) [60, 75, 104, 136, 157]). A handful of studies also explored multipurpose within a change commit [99, 255, 86]. Overall, these studies have not focused rigorously on handling the message triad – empty, ambiguous and tangled commit messages. The explored source code properties are mainly abstract operations, method body, roles of the methods, code change density, size of the added and deleted LOCs, and so on. They are intuitive enough neither in the context of architectural change grouping nor for descriptive design artifacts generation. Our study explored models for handling the message triad. Our explored SSC properties are more explicable for change grouping and DDARTS generation.

7.8.2 Combined Model for Change Classification

Dagpinar and Jhanke [60] explored code and class properties for predicting perfective/adaptive, corrective, and preventive classes. The number of statements TNOS, Non-inheritance class-method import coupling NICMIC, Non-inheritance method-method import coupling NIMMIC, and Non-inheritance import coupling NIIC are the best metrics producing 62 -99.7% R^2 values for perfective/adaptive and corrective classes. Most of the metrics represent architectural properties as discussed in Section 2.2.3. That said, this study can be treated as the first to use architectural metrics to classify changes. However, the outcome of the preventive class is not discussed.

Levin and Yehudai [136] utilized Fluri's [75] taxonomy of 48 source code changes (for example *statement_delete, statement_insert, removed_class, additional_class*, and so on) to classify commits into Mockus [168] classes. However, they also attempted keyword searching within the commit message, which produced a poor result. Later, they explored a classification model based on a hybrid classifier (J48, GBM, and RF) that exploits commit keywords and source code changes (e.g., statement added, method removed, and so on). This approach can select an alternate model during predicting a class if there is an ambiguity in determining a type. Finally, the best class's precision is 56%, recall is 96%, and F1 scores would be 70.7%.

Mariano et al. [157] proposed an improvement of Levin and Yehudai approach to classify commits into Mockus [168] classes. Particularly, they included three additional features (method change, statement change, and LOC change) in the classification model for XGBoost (a boosting tree learning algorithm). Hence, the total number of features is 71: 20 keywords, 48 change types, and additional three features. The best accuracy (ac) with this technique was 77% with Levin's [136] dataset. However, precision (how many retrieved samples are relevant to a particular change type), recall, and F1 scores are not presented, and the overall outcome is unlikely to be better than [136].

Wang et al. [241] proposed a method to classify large review commit (having more than two patch sets) into nine categories: bug fix, resource, feature, test, refactor, merge, deprecate, auto, and others. They have employed various types of features for classification: text, developers' profiles, and change operations. Additionally, various machine learning classifiers explored by Hindle et al. [101] are experimented with in this study. However, the specialty of the approach is that the different combinations of features are used for predicting different groups (achieves 67% $F1$ score).

Unlike the previously mentioned classification models, Hönel et al. [104] introduced source code density, a measure of the net size of a commit. The experimental outcome shows it improves the accuracy of automatic commit classification compared to Levin's approach. However, code density is the ratio of functional code and gross size, and functional code is the total code that is executed any how. In comparison, gross code contains comments, whitespaces, dead code, and so on, along with called code. This density feature is calculated on each of the change operations (file or statement deletes or add) individually. For all data, the best accuracy is 73%, and Kappa is 0.58. For cross-project, the best accuracy is 71%. However, they did not present precision, recall, and F1. It is highly likely that the best F1 scores are similar to the baseline study of Levin and Yehudai [136]. However, they reported that the textual properties produce better outcomes in a few cases.

Untangling Commits: Several studies focused on untangling commits [61, 140, 240, 197, 179]. We also explore a lightweight model for untangling architectural commits with a more rich set of information. The model is applied in design artifacts generation, and the outcome is promising.

7.9 Conclusion

In this study, we explore classification models with SSCs and concept-tokens to handle the message triad for architectural change categorization. In particular, we annotated a large collection of high-level architectural change commits into four categories. Then, we explore several classification models. Through performance measurement of them, we found that SSC properties alone can extract change type with 65% Hit@2 accuracy while the combined model's accuracy is 86% Hit@2 which are promising outcomes. The combined model also can handle message triad challenges on a significant scale. That said, our proposed change categorization approach can be deployed in the real world for various purposes, including design change document generation.

Our proposed change classification technique is stable and reliable. In the next study (i.e., Chapter 8), we will utilize the change detection and classification techniques and the explored features for generating change summaries that are useful for change review and developer-centric release logs.

Chapter 8

DDARTS: A Case Study for Descriptive Design Change Artifacts Generation

As a case study, we explore an automated technique to generate architectural change summaries (textual) leveraging the most promising change detection and classification techniques we have proposed in previous studies in Chapters 5, 6 and 7. However, context-aware descriptive design change summary (a subset of design change artifacts) generation is a challenging task [206]. To that end, we consider precise and meaningful contexts based on the SSCs (in Chapter 6), change purposes (in Chapter 7), and relevant concepts related to them (in Chapter 5) for generating the descriptive design change summaries. In this process, we first generate SSCs and concept tokens mapping models from the training dataset created in the previous study (i.e., Chapter 7). This mapping model contains separate models for each change group (with weighted ranked words). Then, during change description and release change logs generation, we determine all the possible change types using the uniform distribution models proposed in our fifth study. Then, the top-ranked concept tokens of those SSC mapping models from the relevant categories are included in the number of unique sets (according to the predicted types). The messages can be generated considering the first 1-3 top-weighted concept tokens. We also integrate commit theme information using various contextual rules. Moreover, we have defined four static rules for generating four types of change descriptions. Finally, we developed a tool for the development and maintenance team for design change information extraction and change log documentation.

The rest of the chapter as follows. Section 8.1 presents a brief overview of our study and Section 8.2 offers a motivating example. Section 8.3 discusses related work, Section 8.4 presents our dataset of study; Section 8.5 presents change logs analysis; Section 8.6 discusses our proposed technique; Section 8.7 presents our tool; Section 8.8 presents performance evaluation; Section 8.9 threats to our study; and Section 8.10 conclude this study.

8.1 Introduction

In recent times, people all over the world have noticed that software anomalies are causing problems in various dimensions of people's daily life, such as healthcare systems, deadly transportation crashes, private data leaks, disruption of energy supply, denial of social networking services, denial of regular socioeconomic activities, and so on. These anomalies originate from software bugs, software security holes, problematic integration of changes, complex-to-understand, and so on. [257, 154, 243, 167, 217, 189, 175]. Analysts, CEOs, and CTOs are also warning of severe problems in retaining the software/IT workforce in upcoming years, which will cause severe business and economic damage to many developed nations. They indicated this as the result of inverse socioeconomic trends where software development and maintenance complexities (cognitive loads) are increasing, but the psychological interests of people in complex jobs are decreasing. Researchers have linked all the mentioned problems in software development and maintenance to inconsistent and complex design and a lack of proper ways to easily understand what is going on and what to plan in a software system (code comprehension). This is due to the fact that there is a significant gap between the information and insights needed by project managers and developers to make good decisions and that which is typically available to them [45]. Hence comes the necessity of generating descriptive design change documents and artifacts.

Proper documentation is vital for any software project, as it helps stakeholders to use, understand, maintain, and evolve a system [13]. However, proper documentation is challenging due to insufficient and inadequate content and obsolete, ambiguous information, and lack of time to write documents [45, 13]. To eradicate these shortcomings and reduce human efforts, researchers are developing advanced systems that automatically generate context-aware document considering the usefulness of a task. Despite this, proper documentation further encounters conceptual and technical challenges related to the collection, inference, interpretation, selection, and presentation of useful information [45, 13, 33]. Moreover, little focus is given to automatically generating summarized documents of design impactful changes despite being used by the development and maintenance teams significantly [13, 33]. On the other hand, software design artifacts can be numerous and complex. Manually inspecting hundreds of change records to discover what they have in common and prioritize their importance is not practical. In this regard, intelligent summarization techniques can be employed to automate these tasks and help managers and developers focus on gathering high-level insights [45, 13].

A development and maintenance team requires to assess and produce (sometimes frequently or sometimes only on-demand) various design artifacts – design document, release notes, descriptive changelogs, design decision associativity with the relevant or impacted components [208, 13, 178]. Then again, adequate design change description is required for code comprehension and code review. However, more than 85% software project managers and 60% developers are likely to use architecture documents, component dependencies information, change type, and other software documents [45]. Moreover, 40.5% of the major and 14.5% of the minor releases contain high-level design change information [33]. As many as 17 people (mostly the core architects of a project) can be involved in producing logs for a single release [33]. Apart from these, empirical studies with the major companies revealed that software change document has widespread impacts on project development and maintenance [13, 44]. These change documents can be of various forms. However, there is a scarcity of automated tools to support specifically the system design changes.

Empirical studies with hundreds of software developers in major software companies reveal that a tool should be easy to use, fast, and produce concise output about software analysis document for them [45, 13].

However, neither heavyweight nor lightweight design change document generation tools are readily available. To complement this gap, in this study, we conduct an exploratory study and propose a lightweight approach for generating design change information of <*Actual purpose*>:<*Reasoning of change*>: <*Code change relation information*> format. Such descriptions are recently being used by major industrial and open-source projects [76, 236]. These are also valuable entities for the design reviewers and various stakeholders. However, there is a clear gap in automated tool supports for producing such rleaselogs [33]. As is discussed in Section 2.4 in Chapter 2, architectural change detection and categorisation are the integral parts of developing a supportive tool in this context. Proper generation of these entities requires proper grouping of changes (such as feature improvement, flaw fixing and refactoring) [33]. An example part of the design change information for the feature improvement task – *"update the API surface area"* is *"InMemoryPM moved to samples"*. Thus, a descriptive design change artifact contains (at least) - *"why the design change has happened", "what high-level program properties are impacted" and "what will be the probable descriptive summary/logs of that change"*. We collectively referred to them as the descriptive design change artifacts (DDARTS).

The basic steps of design artifacts generation are shown in Fig. 2.5. A few of the examples of DDARTS in a real-world project are discussed later in Fig. 8.1 and 8.2. As the steps suggest, it is quite insightful that following these steps for all the commits of a release is almost impossible for humans or infeasible for heavyweight techniques (such as those that require compiling the codebase for the AST generation [131]) considering the time, costs and benefits [33]. Among other things, proper grouping increase the knowledge sharing and discoverability weight of the software documentation, which is a crucial factor of the development and maintenance team's productivity [13, 77]. Hence, our study also consider the *easy-to-understand-the-purpose* knowledge sharing capability from the complex design changes.

A simple example of such an artifact is presented in Fig. 1.5 (crawled from a commit of the Azure SDK). In this artifact, *Feature Improvement* presents the actual purpose of the commit. *SSC* part provides quick information about the design relation and design impact. The *Summary* part is crucial for review because this info will prompt urgent inspection of runtime security implications (given that the $DM2$ is sensitive and restrictive) by the experts. Please note that this is fundamentally different from a commit message that the DeltaDoc [44], ChangeScribe [144] and other existing techniques generate [147, 239, 112, 148], but can be used as commit messages in some cases. Moreover, these studies did not consider tangled commit message generation when multiple unrelated changes happen, which is common in practice [61]. In sharp contrast, our study will seek scalable approaches to provide the information similar to Fig. 1.5 and 8.4 to the development and maintenance team from the change commits. Overall in this study, we focus on the following research questions:

RQ1: *What types of information are development and maintenance teams documenting in design change logs for the releases?* In real-world development and maintenance activities, DDARTS such as summary descriptions and changelogs for major changes from the commits are added for the maintainers, reviewers, and software releases. In this RQ, we explore different types of information contained in the release change logs.

RQ2: *What development artifacts are contributing to the writing of design change logs* In this RQ we investigate different types of development artifacts such as commit message, issue description and change operations that are required to write the information extracted in RQ1. So that, automated techniques and tools can consider and handle them efficiently.

RQ2: *How SSC and concept tokens are promising for generating design change summary?* In real-world development and maintenance activities, DDARTS such as summary descriptions and changelogs for major changes from the commits are added for the maintainers, reviewers, and software releases. In this RQ, we explore the competence of the SSC properties to generate such design artifacts. Answering this research question will help the researchers construct a baseline for scalable design artifacts generation tools research.

RQ3: *How SSC and concept tokens are promising for generating design release change logs?* In real-world development and maintenance activities, DDARTS such as summary descriptions and changelogs for major changes from the commits are added for the maintainers, reviewers, and software releases. In this RQ, we explore the competence of the SSC properties to generate such design artifacts. Answering this research question will help the researchers construct a baseline for scalable design artifacts generation tools research.

To answer RQ1 and RQ2, we collected around 100 recent release change logs from open-source-software projects (commercially important) and manually investigated the contents. Following the findings, we proposed an algorithm to generate descriptive change summaries. To answer RQ3 and RQ4, we propose a lightweight approach for generating descriptive design changelogs that consider more precise and meaningful contexts based on the relations among SSCs, concept tokens and change purposes. We consider different static rules for change commit summary and release logs for natural text description generation considering the textual contents of the explored change logs. Then we compared the generated words with the contents of 50 change comments and 100 release change logs instances. The performance measure for these samples with the text summarizing metric ROUGE [142] (overall 50% precision) reveals that our proposed approach is encouraging in descriptive design change summary and release change logs generation given the complexities in this domain. We also conduct subjective cross-validation of the outcomes by the developers. Finally, we developed the DDARTS tool to generate design change summaries. We also measure the execution time of all the steps for generating change logs and found it very scalable to frequently use in real-world development and maintenance activities. As far as we are aware, this is the first study of automated descriptive design change log generation in contrast with typical commit message and release note generation [148, 239, 178, 120]. Overall contributions to this study:

- Construct a benchmark dataset to study design change logs generation.
- Extracted what types of information are contained in the developer's written change logs.
- Proposed a technique for scalable design change artifacts generation.
- Present various performance evaluations for generated change logs.

8.2 Motivation and Background

8.2.1 Motivation

Scenario 1: Adequate Information for Change Comprehension and Review: Since code reviewers are not the developers of the implemented tasks, they need accurate information to extract considering various perspectives. However, message description does not contain intended and other information, as shown in some real-world examples in Table 8.1. *Motive in the Description* column represents possible motives of the developers. Reviewers or document writers can only get the real motive by discussing with all the involved developers. Instead, our understanding through code analysis is summarized in *Change in Code* column that appears to be inconsistent with the developer's writing. That said, developers may write a change task inconsistently. Meaningless or empty messages make the situation worse [148]. Hence, all the information in Table 8.2 is valuable for the reviewers (as well as the description of the SSCs and modules). However, potential information of the design impactful changes may contain this format (at least) - *Actual purpose*, *Reasoning of change*, *Design change relations*, and *Design Change Activity.* An example format is described in Fig. 1.5. A few more descriptive design changelogs written by the Azure team are shown in Fig. 8.1 (e.g., *A has been flattened to include all properties from C and D classes*). Here, the *Breaking Changes* are related to the preventive change [63]. Furthermore, a detail comment for reasoning the change commit is shown in the *Detail Comment* column of Table 8.2. Many of these samples do not follow specific content structures. Manually writing them requires analyzing all the change revisions of a release (sometimes previous releases). In summary, simplified and useful information by an automated tool is more useful than manually analyzing several hundreds of code segments, methods, classes, and modules (as is required for the 3rd sample in Table 8.1). Once the change purpose and causal relations in the code are extracted, many other design artifacts like these can be generated.

Scenario 2: Design document generation: A DNM team wants to generate a design document for the upcoming release. A part of the document requires extracting the main features and its associated components. However, without efficient technique, the last two in table 8.1 could be falsely processed as feature improvement using description, but originally it was for design simplification. Fig. 8.1 shows a partial release log for the developers of AzureSDK *4.0.0-preview.4* in the log change[1] for corresponding feature level description of the release note[2]. Which includes the design-related changes.

Such release changelogs are different from usual releasenotes. A good example that contains change group information can be found in Azure SDK release logs[3], which is shown in Fig. 8.2. Many changelogs, as we have explored ∼200 instances, contain much more information (as is shown in Fig. 8.1). However, writing such logs requires much more effort and analysis of various information (i.e., all commits, codebase, commit

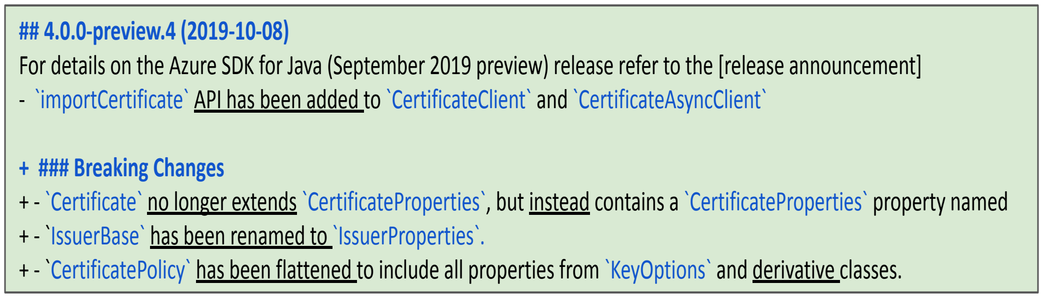
[1]https://tinyurl.com/4xvh2s8j
[2]azure.github.io/azure-sdk/releases/2019-10-11/java.html
[3]https://tinyurl.com/yc2fp3rx

Table 8.1: Design impactful changes (DIS) in various projects and their description.

Serial	**Description**	**Motive in description**	**Change in Code**
1.Aion	*update p2p logging & fetch headers based on td*	Either new feature or refactoring or both. Which part is design impactful?	Feature improvement
2.Aion	*some PR changes*	Does it improve feature or refactor? Which issue is linked with this change?	Design restructuring
3.Azure	*Refactored ADLS set access control and added builders for different types*	Two tasks, which one impacted the design	Both of them impacted the design, feature improvement and design refactoring
4.Azure	*Storage InputStream and OutputStream*	Looks like new feature	Contents of some classes moved into new classes, improve and simplify design
5.webfx	*Improved Action API*	Could be a feature or design improvement	Contents of a class moved into new class, Design improvement

Table 8.2: Detail comment of a DIS. Underline tokens are the most important info to the developers.

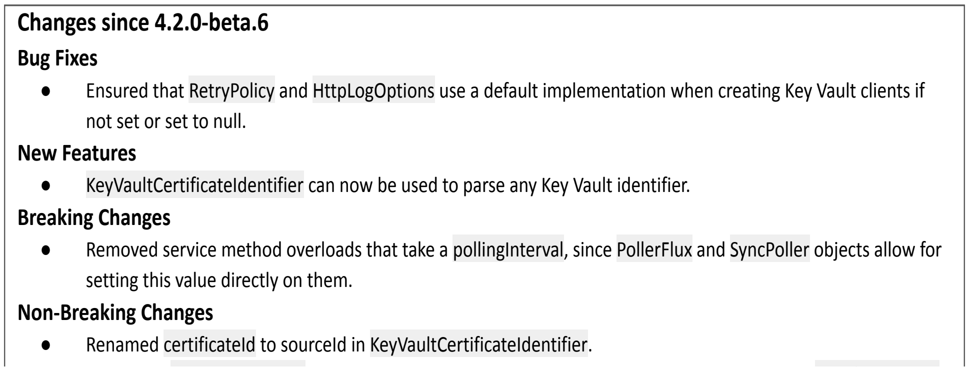
Commit Msg	**Change Summary Comment (Partial)**	**SSCs**
Updates to event processor surface area for preview 5 (#6107)	The intent of this change is to update the API surface area with the following changes: - Handlers for partition processor methods (event, error, init, close) - InMemoryPartitionManager moved to samples - Event processor takes all params required to build the client ... - New types for each event type (PartitionEvent, ExceptionContext... - PartitionManager renamed to EventProcessorStore - Added fully qualified namespace to store interface and a getter in EventHubAsyncClient	7



```
## 4.0.0-preview.4 (2019-10-08)
For details on the Azure SDK for Java (September 2019 preview) release refer to the [release announcement]
- `importCertificate` API has been added to `CertificateClient` and `CertificateAsyncClient`

+ ### Breaking Changes
+ - `Certificate` no longer extends `CertificateProperties`, but instead contains a `CertificateProperties` property named
+ - `IssuerBase` has been renamed to `IssuerProperties`.
+ - `CertificatePolicy` has been flattened to include all properties from `KeyOptions` and derivative classes.
```

Figure 8.1: Descriptive changelogs of DIS of AzureSDK.



Changes since 4.2.0-beta.6

Bug Fixes

- Ensured that RetryPolicy and HttpLogOptions use a default implementation when creating Key Vault clients if not set or set to null.

New Features

- KeyVaultCertificateIdentifier can now be used to parse any Key Vault identifier.

Breaking Changes

- Removed service method overloads that take a pollingInterval, since PollerFlux and SyncPoller objects allow for setting this value directly on them.

Non-Breaking Changes

- Renamed certificateId to sourceId in KeyVaultCertificateIdentifier.

Figure 8.2: Partial release changelogs for DIS of AzureSDK.

message, issue reports, and review reports) than writing the commit messages. As many as 17 people (mostly the core architects of a project) can be involved in producing logs for a single release [33]. Consequently, tool support for descriptive design changelogs is more critical than for commit message generation.

8.2.2 Background

Architectural Change: Software architecture/design may be modified intentionally or unintentionally during the development and maintenance life-cycles. Software architecture modification is considered as of [192] – configuration change [72, 87], source-code layers (i.e., directories, package structures, and location of code files within the directories) changes [151], design model change (i.e., UML diagram) [245, 152, 81], architectural document in natural language [63, 115], and code component change operations (i.e., addition, deletion, moving, and merging components) [246, 213]. However, software architecture is studied at three abstract levels: high level, intermediate level, and low level. In this study, we focus on the commits having module/system (higher) level changes. A module can be a sub-system, 3rd party library, and cluster of packages [41]. Overall, the change commits having architectural design changes contain additions, removals, and moves of implementation-level entities from one module to another. This also includes additions and removals of modules themselves. Moreover, include and symbol dependency changes [151, 87] are also architectural changes. In summary, our consideration of these metrics as architectural changes are based on a number of existing studies [47, 81, 130, 213, 151, 245].

Architectural Change Type: Architectural changes can be grouped on focusing various perspectives [246, 63, 169]. In this study, we consider change grouping based on the development and maintenance activities [246]. They are as follows (details are discussed in Chapter 2):

Adaptive (A): This change is a reflection [246, 143] of system portability, adapting to a new environment or a new platform. *Corrective (Cr):* This change refers to defect repair, and the errors in specification, design and implementation. *Preventive (PV):* Preventive change [170, 246] refers to actionable means to prevent, retard, or remediate code decay. This is related to inappropriate architecture that does not support the changes or abstractions required for the system. *Perfective (PF):* Perfective changes are the most common and inherent in development activities. These changes mainly focus on adding new features or requirements changes [226, 246, 63] including improving processing efficiency and enhancing the performance of the software.

Design Change Artifacts: We refer to design change artifacts as the collection of one or more design change-related entities - change impacted components, modification operations, modified dependency relations, purposes of change, associated design decisions/features, descriptive change summary, modified design document, design changelogs, design tactics, descriptive comments for the reviewers, and so on. A few of the artifacts are discussed in the *Motivation* section. These artifacts are valuable for reverse-engineering as well [50].

8.3 Related work

8.3.1 Automatic Code Change Document Generation

One of the pioneering works for code change description generation is the DeltaDoc tool by Buse and Westley [44] that generates a lengthy textual message for a large change. Later, a *changedescribe* tool focusing different balanced message content was developed by Linares-Vásquez et al. [144]. A few studies also focused on the automated summary generation of commits. Jiang et al. [114] proposed a lightweight technique to generate a short phrase for a change commit that is basically a commit message. Likewise, our proposed technique also generates commit change summary focusing on the high-level design changes. Rastkar and Murphy [206] proposed a summarization technique to generate concise descriptions of the motivation behind the code change based on the information present in the related documents. They can locate the most relevant sentences in a change set to be included in a summary. Although it is a lightweight technique, it does not consider source code properties and is not elegant when the input contains non-informative and empty descriptions. Similar to this study, Shen et al. [216] proposed a technique generating *Why* and *What* information for commit messages for code changes by considering methods and textual description. However, the proposed technique requires providing change type information for generating *why* info. In contrast, our technique predicts change type without providing that information. In conclusion, our proposed tool does not replace these studies but rather complements these studies for design change summary.

8.3.2 Automatic Commit Message Generation

Our study is closely related to change message generation. Following the previous studies, Jiang et al. [112] proposed a neural machine translator to select the most relevant commit message from given code *diff* information from a pre-collected ground truth dataset. Later, they enhanced their technique by considering more semantic code information [111]. Liu et al. [148] proposed a promising technique that considers abstract syntax tree structure to match the most relevant commit message from the previously collected commit messages to recommend. Most recently, Wang et al. [239] proposed a neural-machine-translator that considers code *diff* information and previously constructed a larger ground truth commit messages to enhance the contexts (low-frequency word and exposure bias). These studies are excellent for recommending a commit message but are not applicable to generating design change summaries. However, the Commit message and our referred design change document have substantially different contents, although a part is extracted from the commit message if available (otherwise, our technique can also generate a theme for the NI and empty messages). In addition, these studies did not consider predicting change purposes (i.e., new features or bug fixing), without which change contexts are not properly captured. Moreover, integrating design change component relations as described in [173] with these approaches is really complex considering the change categories. In comparison, we propose a model based on design relation properties extracted with a lightweight process (code as string properties [173]) and concept tokens that are much more logical and explainable in the context of design impactful changes (DIS). While our case study is similar to the commit message but a bit more change description of how design has changed as shown in Table 8.2, which is focused on and adaptable to two types of summaries – change description and release change logs. Furthermore, our model inherently includes the predicted change group information within the message, which is required for reasoning a change.

8.3.3 Automatic Release Note Generation

Our work is also related to release note generation. Several excellent studies are available for release note generation for the users/issues [178, 120, 183, 33]. Some of them generate a broader description (class-level) of the related classes/files and methods [178]. In sharp contrast, our work focuses on design impactful (module/system-level) releaselogs, which are neither too short nor too long.

The most promising tool developed for the automatic generation of release notes is ARENA [178]. Unlike others, this tool generates summaries from source code changes and then purifies them with the description of other textual information such as issue trackers, commit messages and pull requests. In this approach, code change revisions are processed by srcML (for Java only). This tool also categorizes the information which is dependent on the tags. However, it needs to parse AST, which is heavyweight for frequently producing release notes for large projects. As the authors mentioned, ARENA does not support high-level architectural change information. Moreover, it generates very lengthy release notes.

Klepper et al. [119] proposed an approach for user-specific release note generation frequently by collecting information from various sources (filtered, grouped, and sorted). In this approach, considering contexts is also valuable. However, it only automatically collects change and issue descriptions; it does not generate notes automatically. Thus, it generates very lengthy release notes. As a result, manual intervention is required for enhancing and tailoring the release notes produced by their approach. One crucial finding in this study is that not all information is necessary for all audiences (rather, it is chaotic for them).

Nath and Roy [183] proposed a text summarization technique to filter more precise but not lengthy information for generating release notes. To that end, they deploy the TextRank algorithm with GloVe for semantic similarity of the commit messages and merge pull-request titles. However, this technique does not process source code, and cannot handle if the commit messages are empty and meaningless.

However, our work does not replace these studies, rather provides additional supports when integrated with them, because, 40.5% major and 14.5% minor releases contain system-level design changes [33].

8.4 Dataset Collection

We collected release change logs and change descriptions from two open-source software. In the data collection phase, we followed some criteria to ensure the standard of the experimental projects – (i) the projects should have well-defined high-level modules since high-level module extraction is infeasible (took 200 hours to 2 years according to the project size), (ii) projects are also commercially important, (iii) projects should have at least 100KLOC, (iv) projects are in a very active phase of development and maintenance in recent times, and (v) projects must have some releases. Thus, the Azure JavaSDK and Hibernate-search projects are the best choices that fulfill these criteria. The summary of these projects is presented in Table 8.3. Initially, we focused on collecting 50 major releases of each of them during the 2021 to 2019 period (prioritize the selection from the recent releases first, starting from 2021 until we get 50 releases). However, among the 50 change logs, we found that some of them do not contain much change modification information; instead, they provide links to change logs. Therefore, for the analysis, we consider the release that contains *WHAT*, *WHY*, and *HOW* information [44]. We followed the *WHAT* and *WHY* concepts described by Buse et al. [44]. Here, *WHAT* - explains what happened in the change itself (e.g., "Replaced a warning with an IllegalArgumentException"); *WHY* - explains the context why it was made (e.g., "Fixed Bug #14235"); *HOW* - explains how changed impacted the components (e.g., "InMemoryPM[*class*] moved to samples[*module*]"). Hence, for the Azure JavaSDK, 39 releases have expected change logs. Then again, for Hibernate-search, we found that 26 release logs (among the selected 50 release logs) have considerable change modification information. Therefore, our collection contains around 64 major change releases.

Table 8.3: Projects for studied change logs.

Project	Commits	Size	Devs	Releases	Domain
AzureSDK[76]	15,180	6.2MLOC	472	4000+	Cloud computing and data analytic services
Hibernate Search[236]	9504	263KLOC	61	100+	Data extracting and searching library for cloud platform

8.5 Exploring the Design Change Logs Contents

8.5.1 RQ1: Types of Information Contained in the Descriptive Design Artifacts

We have manually analyzed the collected change logs and extracted types of information contained in change logs. Collected AzureSDK change logs have 2,618 lines (excluding empty lines) of texts that took 24 hours (per person) to analyze. While hibernate search has 26 release change logs having a total of 1,085 lines of texts that took around 16 hours to analyze. In the annotation process, we followed the Java coding convention to detect class, method, and module elements inside the textual descriptions. Whereas, detecting APIs/libraries is straightforward. When we complete the manual annotation of types of information contained in each of all releases, we have found at least 23 types of information. The summarized form and histogram of them are shown in Fig. 8.3 (for Azure SDK 39 samples). In this collection, the most frequent information is change type, class, module, methods, and API name. However, we have found an interesting trend that around 33.33% change logs only contain change type information with the description of operations. They do not mention the associated feature/issue/design decision with them. However, the mentioned change types information is bug fixing, feature addition, and feature improvement. Azure SDK team writes breaking changes (refactoring) and dependency change types, while Hibernate search has a special group of information called *tasks* which basically contains documentation, tests refactoring, and other logs.

Most of the logs for both projects have information about the modified classes and modules. However, classes are mentioned in as many as ten instances in a single log. At the same time, the method and API/library names are frequent. Attribute variable/properties and parameters are also included at a considerable amount, but their instances are up to three in a single release. However, the change operations are more descriptive for the Hibernate project logs than the Azure SDK project. These change operations are contained mostly in a semantic or meaningful way. Overall, the variation of frequent change operations is - removing, refactoring, deprecation, adding, moving, replacing, renaming, exposing, merging, splitting, usage, including, dependency update, and so on., for class, module, method, and APIs. However, in some rare instances, change operations include inheritance, subclass, and superclass. Much non-coding related information is also present in the logs, as is shown in Fig. 8.3 but not frequent. Moreover, we have observed the major release documents of

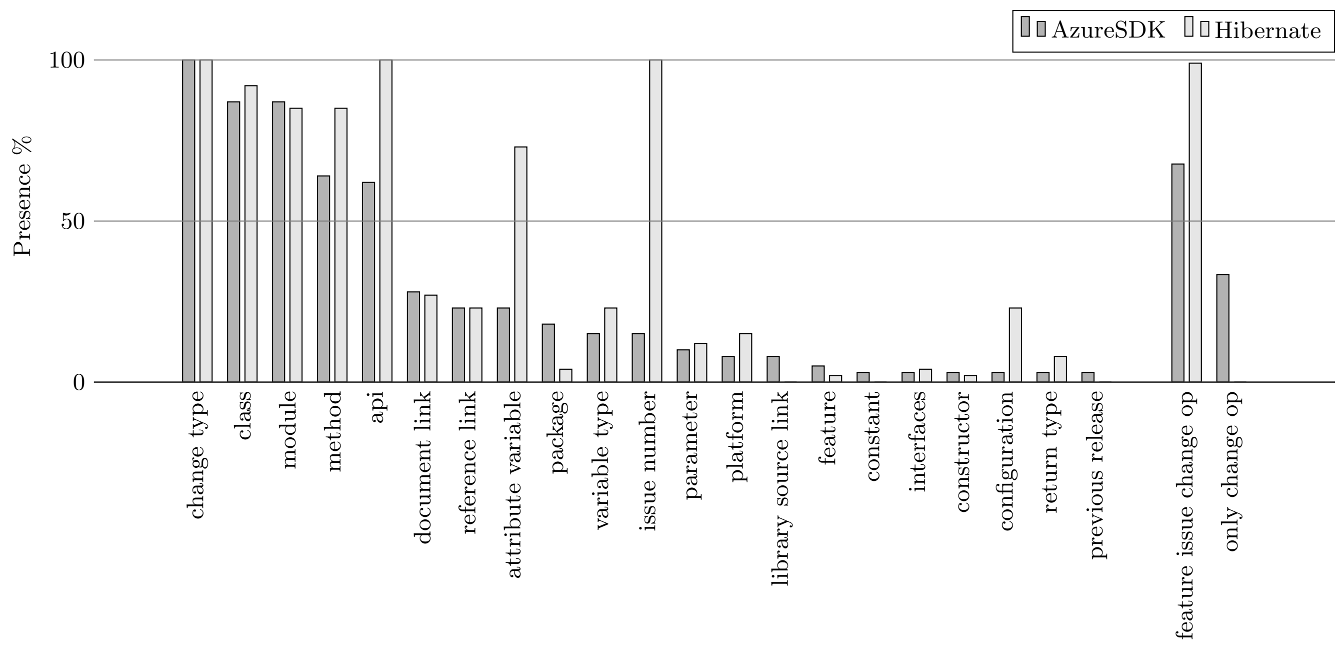


Figure 8.3: Histogram of different types of information contained within a release change logs.

Hibernate-search that also write the changed semantic architecture with high-level components description (presence is 100% for all the major releases). However, one notable difference between Azure and Hibernate is that Azure also maintains separate logs for each module. However, the release change logs are formally included in recent years (since 2019) for the Azure SDK releases. It is highly likely that the logs will contain more formal and understandable implementation change information gradually. In summary, these findings are helpful in developing automated tools for design change logs for the practitioners and techniques for automatic extraction of change type information for large-scale empirical study for the researchers.

8.5.2 RQ2: What development artifacts are contributing for documenting the design change logs

Knowing the sources of information for documenting design change logs is crucial for developing automated tools. To answer this RQ2, we have subjectively tracked the presented information types in the previous section. We found that this information is aggregated from commit messages, parent issues, comments on the parent issue, API documentation, release notes, change logs of each module, change operations, and source code. We also notice that this information is combined mostly in a short but meaningful way (neither too long nor too short). In many cases, the commit messages are mostly grouped to change-type information when aggregating.

8.6 Descriptive Design Change Summary Generation Model

8.6.1 Phases of Design Change Summary Generation

The details about the phases of generating design change artifacts are discussed in Section 2.4. Here, we briefly introduce our strategies in the major phases.

Design Change Revisions Detection: Not all the change revisions or commits contain architectural changes. That said, the revisions that are not architectural are not the candidates to extract design change information. Therefore, in the first step, revisions containing architectural change are identified, and then SSCs are extracted. However, existing studies for detecting high-level architectural changes are heavyweight that require byte-code generation, human intervention, and high-performance computing, limiting the frequency of deployment and number of change revisions [150]. Consequently, we deploy our proposed lightweight tool in Chapter 6 for these purposes.

Change Category Determination: The next crucial step is determining the causes and types of change. For this step, we deploy our proposed classification model in Chapter 7 that can also handle NI, empty, and tangled changes.

Textual Description Generation: The final phase is generating natural language description aggregating and considering the information of the previous phases. In this phase, we employ a rule-based natural language model for this purpose. However, one of the most challenging tasks in automated change summary generation is sensing the proper change contexts [44, 144, 112, 148, 239]. To handle this challenge, we attempt to consider 4X17 (68) dimensions of contexts consisting of four change types (i.e., perfective, preventive, corrective, and adaptive) and 17 semantic structural change relations. In the next subsections, we describe our proposed technique in detail.

8.6.2 Main Algorithm

An abstract view of reasoning for DDARTS generation is displayed in Fig. 8.9. The promising techniques for commit message generation consider ground truth words from the training data for a context-aware message to reduce low-frequency words and exposure bias [239], but such a context is a bit vague. In contrast, we consider more precise and meaningful contexts based on the SSCs, change purposes, and relevant concepts related to them for generating DDARTS. Surprisingly, we notice that our collected 100 release logs contain 86 concept tokens (discovered in Chapter 5) with a total of 383 instances, meaning that our extracted tokens are valuable when embedded in the proper contexts. The details of these phases are shown in Algorithm 2. We first generate SSC and concept tokens mapping models (STM_{ti}) from the training dataset containing a single change type created in the previous study in Chapter 7. This mapping model contains separate models for each change group (with weighted ranked words). Then, during change description and changelog generation, we determine all the possible change types using the uniform distribution models (*UModel*) deploying the extended ArchiNet strategy presented in Section 7.7.1 in Chapter 7. Then, the top-ranked concept tokens of those SSC mapping models from the relevant categories are included in the number of unique sets ($UTokens$) (according to the predicted types). The messages can be generated considering the top-weighted concept tokens. In Algorithm 2, STM_{ti}, *getRankedToken()*, and *generateMsg()* have degree of freedom of enhancement and customisation options. The *generateMsg()* function will produce sentences based on a few templates similar to rules in [44], but a bit different considering the components C_{Compos} and contextual prepositions [171] selected from the change type (with SSCs), e.g., class A in module $DM1$ moved ***to*** module $DM2$. In the next sections, we discuss strategies of *generateMsg()* function in our proposed algorithm.

```
Data: $TrainData_S$ - Training data containing single changes, $C_{Text}$ - commit message or other texts,
      $C_{SSC}$ - sscs in a commit $C$, $C_{Comps}$ - involved components in the commit
Result: $SD$ - Description for all change types in a commit
begin
    SD = []                                      ▷ Store summary for predicted changes
    DIS = determineArchiChange($C_{CR}$, $C_{CR(i-1)}$)   ▷ If architectural change then returns True
    if $DIS$ is $True$ then
        $C_{SSC}$ = extractSSC($C_{CR}$)
        $STM_{ti} \Leftarrow$ getSSCTokenMap($TrainData_S$, $C_{SSC}$)
        $T_{Change} \Leftarrow$ predictChangeType(UModel, $C_{Text}$, $C_{SSC}$)
        for Type $Ti$ in $T_{Change}$ do
            $Token[SSC] \Leftarrow$ getSSCToken($STM_{ti}$, $Ti$)
            UTokens = set()                      ▷ unique collection
            for SSC in $C_{SSC}$ do
                CTokens $\Leftarrow$ getRankedToken(Range, $Token[SSC]$, SSC)
                UTokens.add(CTokens)
            end
            SD.add(generateMsg($Rule(Ti)$, UTokens, $C_{Comps}$, $Ti$,$CM_{Theme}$))
        end
    else
        Exit
    end
end
```

Algorithm 2: Descriptive design summary generation.

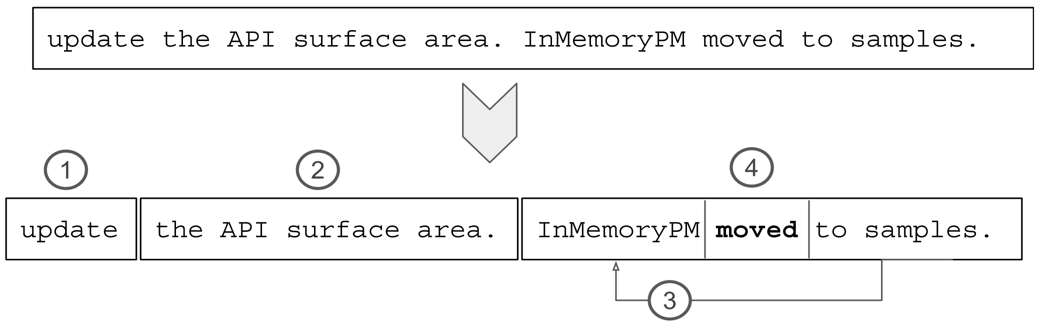


Figure 8.4: Structure of a design change text.

8.6.3 Complete Sentence Generation Rules for generateMsg()

An example structure of a descriptive change is shown in Fig. 8.4 that contains ① purpose token/tokens, ② main theme or parent issue, ③ design components and relations, and ④ change operations. In this study, our target is to generate a similar structure. To that end, we employ some rules for generating complete sentence for change logs for release as follows -

1. Prioritize operations based on the category
2. Reverse stemmed keytokens (KW) + components (CMP) + selected operations (OP) + preposition (P) + Pull some parts from the commit message noun phrase (NP).

The closest technique for generating a sentence is found in DELTADOC [44] which generates one sentence for each modified method in a commit. The sentence rules (mostly consisted of *Do* and *Instead of* phrases) are considered for statement-level changes based on modifications and conditions. Our approach considers 17 properties of architectural change operations and relations in contrast to DELTADOC. Furthermore, they generate up to ten lines of sentences. However, our target is to generate one sentence that is neither too short nor too long, focusing on high-level design change. In comparison, as we are focusing on generating design change logs, we consider a more aligned set of rules for four high-level change types considering four structural units as described in Fig. 8.4. In this structure, ③ and ④ are generated from extracted SSCs, and their associated modules, classes and methods. These categorical rules are provided in Table 8.4. They are formed by analyzing about 2,000 sentences of the collected releases (so-called training samples). However, for the perfective sentence generation, the code change must contain at least ADD or MODIFY related SSC. The semantic meaning for it should focus on feature addition and improvement context. In contrast, for the preventive sentence generation, the code change must contain at least REMOVE or MOVE related SSC. The semantic meaning for it should focus on restructuring or design simplification. On the other hand, the corrective sentence should focus on the problematic theme. Thus, the phrase organization may vary in their sentences. In the sentences, the purpose concepts (structure ①) are generated dynamically using the

Table 8.4: Baseline rules for sentence generation

Change Type	**Must include**	**Sentence Structure**	**Commit theme (CM_{Theme})**	**Comment**
Perfective	Add/Modify	KW + SSCop + P + CMP + F	ADJ∧NP (F)	
Corrective	∼	KW + Problem + P + SSCop + CMP	ADJ∧NP (Problem)	
Preventive	Move/remove	KW + P + SSCop + P + CMP + P +Special word	ADJ∧Special word (SW)	SW if exists
Adaptive	∼ Add/modify	KW + P + Adaptive element + P + CMP+ P + F	ADJ∧NP (F)∧Adaptive element (AE)	API as AE if not provided

SSC-KW mapping models from a predicted change type. One important thing to note is that the SSC-KW model is a stemmed words model which must be reverse stemmed for generating a sentence. However, no automated technique is available to do that. Therefore, we manually reverse stemmed all 293 concept tokens and maintain a static link between these two sets. In this process, different types of feature-related themes (structure ②) can be generated by selecting Noun (N), Noun-phrase (NP), Adjectives (ADJ), and special words (SW, AE). Placing prepositions (P) should also maintain rules according to the change category. Such options are summarized in Table 8.4. In the next section, we will discuss the theme generation strategy for structure ②.

8.6.4 Commit Theme Generation

We dynamically generate a commit theme (short) from the commit message. A theme consists of a noun phrase and adjective such as *"for non-blocking I/O"*. The algorithm for generating such a theme is shown in Algorithm 3. Many ways are possible to generate a theme in the algorithm. In this process, we need to intelligently rank the most relevant words. However, we have observed that those that have the highest length among the lists of adjectives or noun phrases within a commit message are a bit more likely a better theme. To extract a theme, we use *nltk*[1] and *textblob*[2] library to generate noun phrases which are not able to extract for many commit messages. For those cases, the top ranked noun is used.

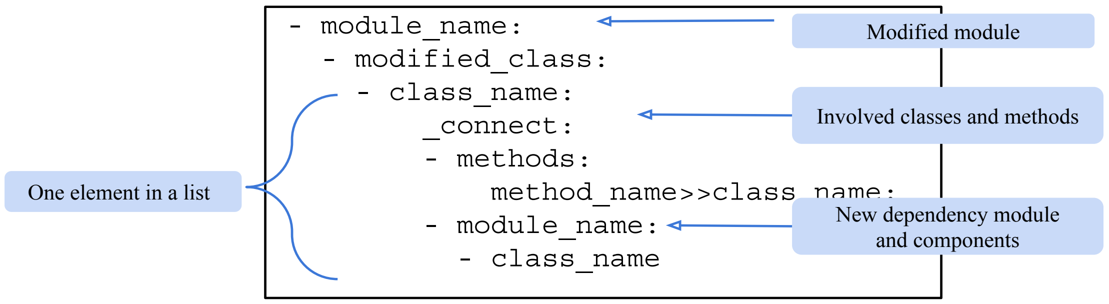
[1]https://www.nltk.org/
[2]https://textblob.readthedocs.io/en/dev/

```
Data: CM - Commit message
Result: $CM_{Theme}$ - Commit Message theme
begin
    CM ⇒ Commit message
    NP = extractAndRankNounPhrases(CM)            ▷ Excluding non-Alpha words
    AJ = extractAndRankAdjectives(CM)             ▷ Excluding non-Alpha words
    if NP is not Empty then
        $CM_{Theme}$ = generateTheme($NP_{Top}$, $AJ_{Top}$)
    else
        NN = extractAndRankNouns(CM)              ▷ Including non-Alpha words
        $CM_{Theme}$ = generateTheme($NN_{Top}$, $AJ_{Top}$)
    end
end
```

Algorithm 3: Commit theme generation algorithm.

8.7 DDARTS Tool

Several tools are available to extract, visualize and analyze an implemented architecture of a released version, for example, EVA [182] and SAIN [82]. However, as far as we are aware, no tool is available to generate descriptive design artifacts and documentation updates based on all the design change versions contained in a release. We have developed a desktop tool for generating high-level design artifacts in Python, combining our proposed approaches. It takes the list of commits for a release from a CSV file and directory of the local branch of the project git repository. It automatically downloads the complete codebase for a committed version from the Git repository branch of the project and extracts changed code segments returned by the GitPython API. Then, the tool compares the two consecutive versions and detects the architectural change version. DARTS tool can be used and extended with minimal effort by the users. The architecture of our developed tool is shown in Fig. 8.7. Our tool has three main independent modules: DISDetect, TangledCategory, and ChangeSummary. The byproducts of each of these modules can also be used for other analysis purposes by the development and maintenance teams. The modules communicate thorough the input and produced data, as can be seen in Fig. 8.7, and the tool script can independently be modified. The performance of DDARTS is primarily dependent on the provided concept tokens model, SSC model, and SSC-Token mapping model for change-purpose-centric document generation. Therefore, DDARTS performance can be enhanced by replacing these datasets with the updated model without writing any script. The description of this module's actions is described as follows:


Modified module
Involved classes and methods
One element in a list
New dependency module and components

```
- module_name:
  - modified_class:
    - class_name:
       _connect:
       - methods:
         method_name>>class_name:
       - module_name:
         - class_name
```

Figure 8.5: A Partial Structure of SSC information saved into YAML. Where, *connect* key indicates new dependency

DISDetect

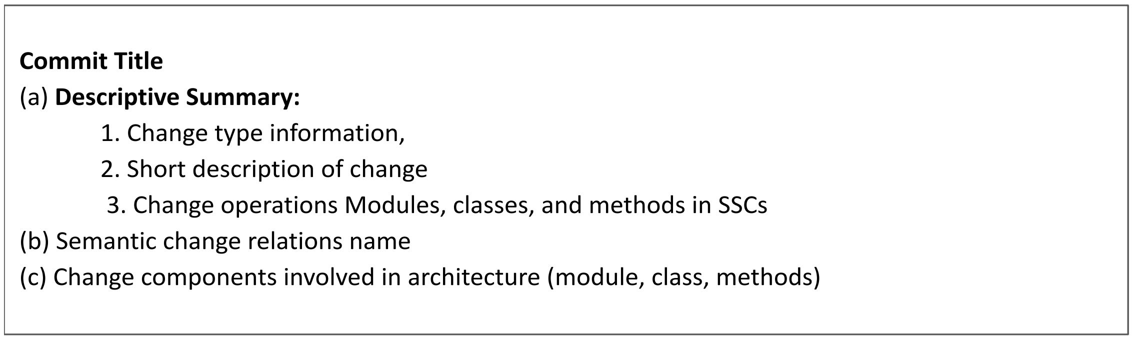
DISDetect module detects the change and produces SSC instances, and saves it to YAML files. This module is implemented based on the approach described in Chapter 6. YAML files contain the name of involved modules, classes, and methods. A partial format is shown in Fig. 8.5. This information can be used for many purposes. We have provided a script for extracting information from the YAML file that can be easily used for adding new features to the DDARTS tool for design change analysis. DISDetect also produces the abstract names of the SSCs from the YAML file to easily understand the change relations meaning (as shown in Table 6.3) such as *MCNM* – Modify class that adds a new method with a new cross-module dependency.

TangledCategory

This module is responsible for determining change purposes using the uniform model. It takes input concept tokens (from Chapter 5) and SSC models (from Chapter 7), SSCs and commit messages are saved in CSV format and produces the predicted category. Predicted category information for each commit is stored in the CSV file that can be later used for many purposes.

ChangeSummary

This module produces change summaries and release logs based on our proposed algorithm described in Section 8.6. It takes input as the SSC word mapping models (from Chapter 7), SSC, and predicted category. Summary against each change commit is stored in CSV, and release logs are stored in two text files. A partial format of this file content is shown in Fig. 8.8. Two variations of this format are produced – (1) one for each individual module (as is done in the Azure SDK project), and (2) all in general in a single place.



Commit Title

(a) **Descriptive Summary:**

1. Change type information,
2. Short description of change
3. Change operations Modules, classes, and methods in SSCs

(b) Semantic change relations name

(c) Change components involved in architecture (module, class, methods)

Figure 8.6: A Partial Structure of a design change commit information for reviewer.

Produced Content Structure

Partial structure of the design change summary is summarized in Fig. 8.6. This structure of the document content also follows experts opining of the written document[1]. It is constructed keeping in mind following criteria –

- Convince the development and maintenance team to act a certain way or believe a certain idea
- To spur conversation
- To motivate
- To persuade

These are aligned in the analyzed release logs.

8.8 Performance Evaluation

Consistent and adequate code change summary generation is one of the most challenging tasks [148, 33]. Challenges - redundancy, consistency, sentence ordering, conciseness, adequacy, while constructing summaries have made this field more difficult [185]. In this section, we conduct a case study of generating design impactful change comments and changelogs, which are special types of design artifacts (DDARTS). We leverage the SSCs and concept tokens with the change type prediction models for that purpose. As the experiments indicate, change group prediction with the SSC+TokenArchiNet is better than DNL, a bit more explainable since it is based on the strength of being a particular category, lightweight, and flexible to adjust. However, the recall rate with Hit@2 is 86%, which is an excellent outcome and can provide an option for the documenter to select from the best two outcomes. We selected ~50 commits from the balanced training set from the

[1]https://libguides.tru.ca/c.php?g=193952&p=1276162

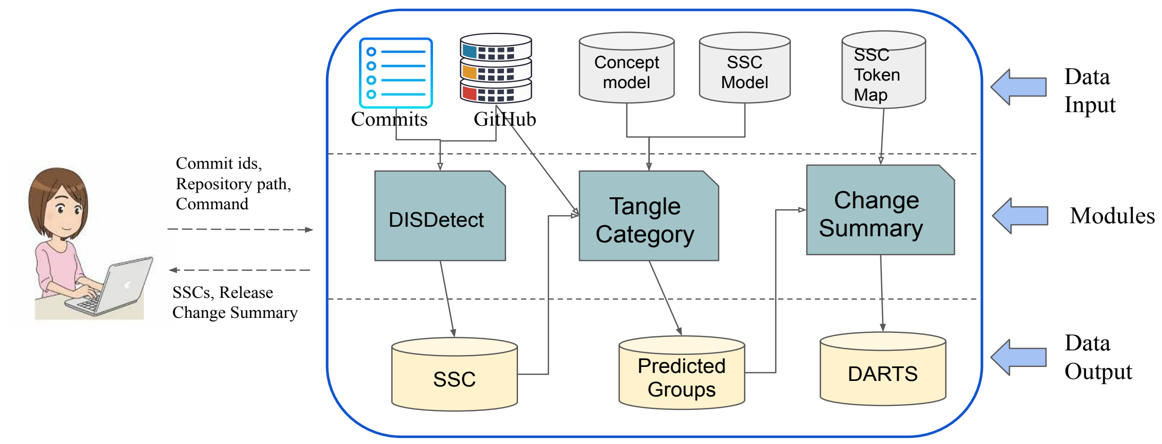


Figure 8.7: High-level architecture of the DDARRTS tool.

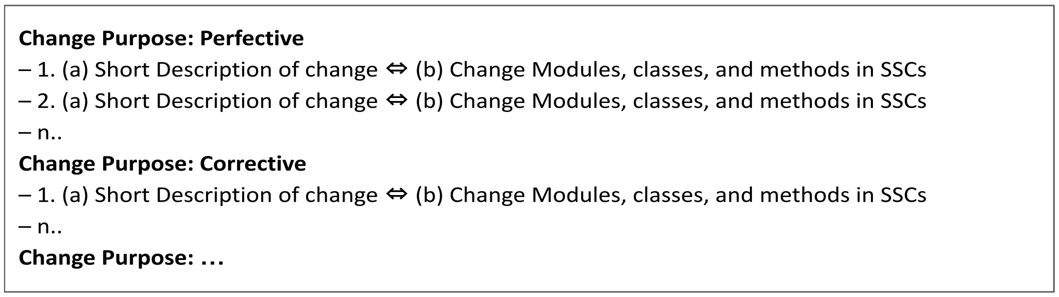

Change Purpose: Perfective
– 1. (a) Short Description of change ⇔ (b) Change Modules, classes, and methods in SSCs
– 2. (a) Short Description of change ⇔ (b) Change Modules, classes, and methods in SSCs
– n..
Change Purpose: Corrective
– 1. (a) Short Description of change ⇔ (b) Change Modules, classes, and methods in SSCs
– n..
Change Purpose: ...

Figure 8.8: Saved release change logs format.

Azure SDK project for the preliminary experiment. Those commits have a good explanation in the parent issue link other than the commit message of what major changes happened. Such an example is shown in Table 8.2. However, more than half of the commits of Azure SDK do not have detailed explanations like them. Summary also requires thematic information from the commit message. Our target is also to include meaningful themes even if there are empty and meaningless commit messages [148]. We have also collected the descriptions of the change logs instances of various types of changes (Fig. 8.2) of Azure projects as of Fig. 8.1. These are the representative formats for changelogs (how changes happened), and half of them are not precisely the requirements described in the release notes. We make sure that the multiple descriptions are not completely the same and reach the collection of around ∼100 instances; the statistics of them are shown in Fig. 8.5. These two types of change descriptions confirm that our approach will be able to produce reviewer-centric change summaries and release changelogs. We conduct three types of performance analysis of the DDARTS tool - (i) by summarization metrics, (ii) by human evaluation, and (iii) by execution time. In

Table 8.5: Change Log Description Instances

Type→	New Featur	Bug Fix	Breakin Change	Minor Change	Depend Update	No Type
#Instance	**53**	9	29	3	2	9
Group	Perfective	Correct	Prevent	~	Adaptive	~

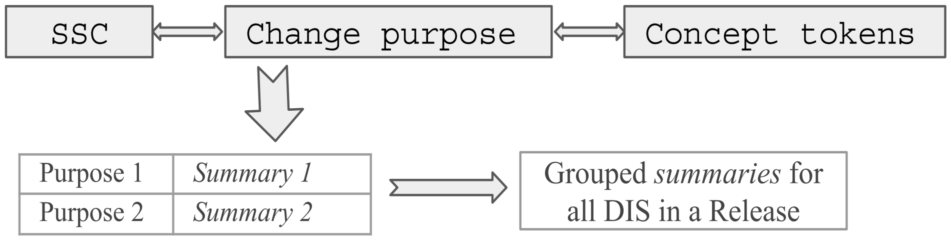


Figure 8.9: Abstract reasoning of design artifacts generation.

the next subsections, we present our performance outcomes.

8.8.1 Accuracy Metrics

For testing the performance, we process the comments/logs by removing the stop words (using API) and code/feature-like components (manually). For example, after cleaning the sentence - *"Fixing event hub consumer"* it would contain only *"Fixing"* as *"event hub consumer"* is related to component names and features. We use ROUGE [142] metrics (an advancement of BLEU [148]) for measuring the performance of our proposed model. ROUGE means Recall-Oriented Understudy for Gisting Evaluation. It is used for evaluating automatic summarization of texts as well as machine translation. It works by comparing an automatically produced summary or translation against a set of reference summaries. In this study, our target is to produce the most relevant words in the initial phase. Therefore, we discuss the performance outcome of our proposed model based on the precision rate of the ROUGE metric, meaning how much of our system summary is, in

Table 8.6: $ROUGE_{Precision}$ of two types of summaries

Artifacts Type	Perfective	Preventive	Corrective	Adaptive	All
Change Desc	22.5	36	15	33	50%
Design release Logs	37	29.5	37	12	42%

fact, relevant or needed? Precision is measured as -

$$\frac{number\ of\ overlapping\ words\ with\ the\ reference}{total\ words\ in\ system\ summary} \tag{8.1}$$

The performance outcome of our algorithm is shown in Table 8.6.

Change Description for Reviewers

The highest precision and F1 measure of ROUGE [142] metrics reached 66% and 55% for the ∼50 samples. Sample count histogram for various ranges of $ROUGE1_{Precision}$ values for 1-gram is shown in Fig. 8.10. Around 36 samples are in the 20 to 80% precision range. When we consider all the samples in a change type as a single doc then the $ROUGE1_{Precision}$ rate are 22.5, 36, 33 and 15% for PF, PV, A, and CR type. Overall $ROUGE1_{Precision}$ is 50%. Given that sometimes developers write a too short or inconsistent description, a machine generated description will be moderate and consistent. Therefore, our initial study reveals an encouraging result of the SSC and change grouping model.

Change Logs for Releases

Shuffling all the generated SDs from Algorithm 2 for all the DIS commits in a release according to similar change types (T_i) forms the release changelogs for the development and maintenance team. Please note that these release logs, as shown in Fig. 8.1 and 8.2 are quite different than the usual release notes generated by the existing studies [178, 120, 183]. We experiment with 100 changelog message instances. We do not include exact descriptions multiple times, which means it covers a wider range of writing variations. We compared the generated tokens with the categorical (distribution is shown in Table 8.5) tokens of those changelogs. We calculated ROUGE scores for four types of changes of all the predicted outcomes from the previous change samples as a single doc. We compared them with the change categorical logs (all the instances in a category are considered as a single doc) as shown in Tables 8.5. The $ROUGE1_{Precision}$ rate are 37, 29.5, 12, and 37% for PF, PV, A, and CR type. Overall 42.2% $ROUGE1_{Precision}$. This outcome indicates that the concept tokens are a significant portion of the log messages. Since these results are from the release log instances, our model is also suitable for release changelogs generation.

Takeaway message for RQ1: SSC properties and change classification models serve as a baseline construction for various design artifacts generation.

8.8.2 Manual Cross-validation

BLEU [196] and ROGUE [142] metrics consider the number of keywords for measuring precision and recall that cannot evaluate the semantic meaning and context of the generated summaries. Only humans can understand the semantics and concepts of a document perfectly. Therefore, we designed a manual cross-validation measurement to evaluate overall semantics and contexts. In this process, we requested five people (not involved with this study) having professional software development experience to evaluate the outcomes of

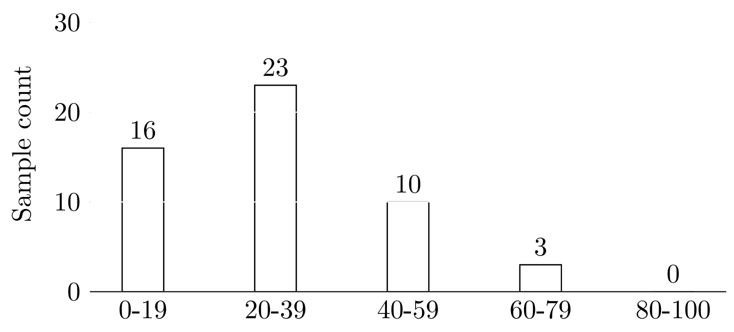


Figure 8.10: Samples over the ROUGE1-P scores ranges.

our tool with the reference summary samples subjectively. We followed the human cross-validation study by Liu et al. [147] for generated commit messages but on a small scale. This section is intended to evaluate the contents of the outcomes of our tool by experienced developers.

Cross-validation Evaluation Design

The involved people for the cross-validation have 3 to 24 years professional software development experience; and four of them are solely assigned to code review during their development job. We designed the developer's evaluation session for the generated contents of DDARTS in such a way that it takes a maximum of 30-35 minutes to complete the evaluation by a single developer. To that end, the study material contains the followings:

- For commit-wise summary evaluation, we selected five representative commits from Azure SDK (perfective, preventive, corrective).
- We provide the commit messages and detailed comments written by the developers of the relevant projects as a reference. Then, we provide the produced summaries of our tool.
- We consider the release *azure-resourcemanager-batch-1.0.0* of Azure JavaSDK project as the reference release log[1]. This particular release log is considered based on the criteria that it likely contains at least the first five and last two types of information as presented in the chart of Fig. 8.3. We selected six commits associated with this release ensuring different types of changes from a release of the (two perfective, two preventive, one tangled commit and one corrective).
- We did not consider the outcome of DDARTS for selecting the samples for evaluation.
- We provide reference release logs and our tool's produced change logs for comparison.
- Then asked them to evaluate seven questions on a scale of 0 to 9 as summarized in Table 8.7. These questionnaires are formed considering various crucial perspectives of the tool. Some of the perspectives

[1]https://tinyurl.com/jby4evdh

Table 8.7: Cross-validation evaluation questionnaires.

Serial	Questionnaires	Perspectives
EQ1	How helpful of the commit change summary content and structure in terms of design/architectural change understanding and review?	Information structure quality
EQ2	How helpful of the release change log content and structure for design change understanding quickly?	Information structure quality
EQ3	How module change relations (architectural properties) useful as the content of design change comment and release logs?	Architectural information
EQ4	How elegant of the textual description content focusing the change context they are representing in general?	Change context
EQ5	How much relevance of meaning of the generated description compared to the example samples that already exist in the projects?	Semantic meaning
EQ6	How much generated sentence keywords are contextual?	Semantic context
EQ7	How much intervention/efforts do you need to make the sentences perfect?	Description quality

are inspired from[1] [118, 237].

Evaluation Results Analysis

The average evaluation scores and their subjective indication of all EQs are shown in Table 8.8. The score of EQ7 indicates that generated sentence is 42% proper ((9-5.2)=3.8) which is aligned with the ROGUE scores. It also indicate that considerable intervention is required to make the generated sentence more perfect. The evaluation outcomes of the evaluation queries (EQs) are shown in the box plots [161] in Fig. 8.11. The box plots summarized the range and median values of the evaluated scales. From the box plot, we observe that EQ1-EQ6 score ranges do not vary significantly, while EQ7 varies considerably, which may be due to the experience of the developers. For the overall significance measurement (in %) of the evaluation of the EQs, we calculate the total average values as follows:

$$AVG_{sig} = \frac{\sum_{n=1}^{7}(M_{EQn})}{N} * \frac{100}{9} \tag{8.2}$$

Here, AVG_{sig} is the overall significance in percentage (%), M_{EQ} median value of an EQ, and N is the number of EQs. We consider the median value because it does not change the meaning if there are outliers; for example, a few developers scored the highest value of 9, but most of the evaluation values are below 6 for

[1]https://libguides.tru.ca/c.php?g=193952&p=1276162

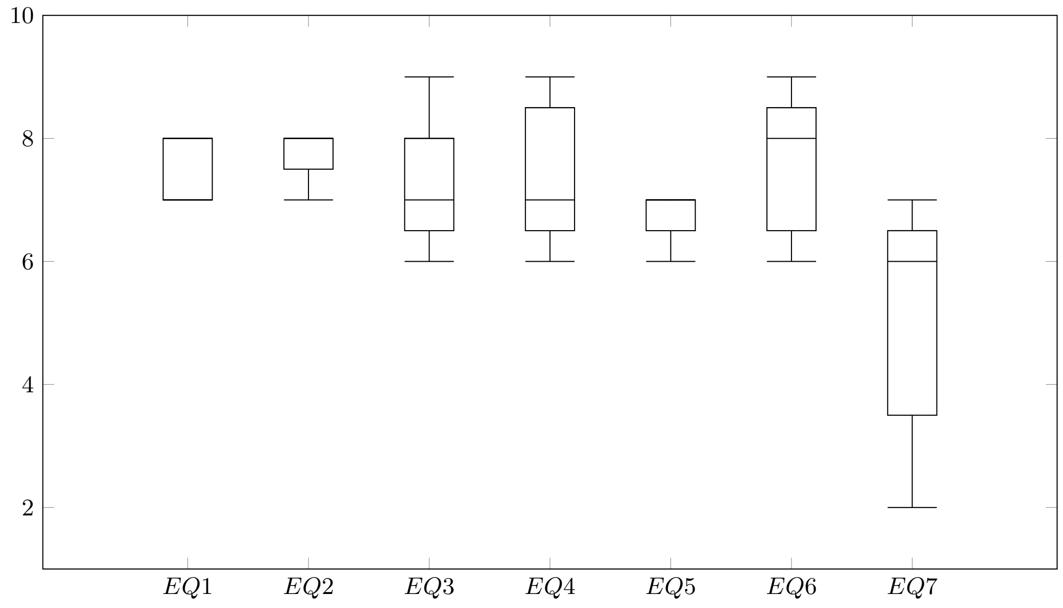


Figure 8.11: Range of evaluated scores of the evaluation queries.

an EQ. For EQ7, we revere the evaluated score as (9 - V_{EQ7}), meaning that if the efforts are required on 3 points, it reduces 6 points effort. Overall average subjective significance is 75% compared to the substantial significance which indicates that our tool is promising.

8.8.3 Scalability

One of the major attributes of a good documentation tool is accessibility, meaning documents can be generated frequently and on-demand with little effort [13]. Another crucial point is the cost of generating such a document, meaning it can be deployed with low-cost infrastructure frequently [13]. Therefore, to address these two crucial attributes, we experiment with the scalability of generating design change documents with a general-purpose computing machine. Users of the DDARTS are only required to input commit ids and directory or local repository path as is displayed in Fig. 8.7. The databases used are CSV and text files. The most promising tool available in public for architectural change analysis for the Java projects requires uploading both all the codebases and compiled bytecodes of all the revisions [182, 213]. That is not the case for our tools. Therefore our tool is very lightweight. The execution time for all phases of change detection, change purpose determination, and change logs generation are shown in Fig. 8.9. Speedment is a medium size project. 243 revisions of the codebase (commits), each having 407KLOC size, requires 35.8 seconds with a 8 cores and 16 GB RAM computer (Core i7-2600, 3.40GHz). AzureSDK is a large project having around 6 million lines of code (MLOC). 276 revisions of the codebase for it requires 136 seconds. This indicates that the DDARTS is quite scalable that satisfies the accessibility and cost-benefit attributes.

Table 8.8: Average scores of the cross-validation measurements and their indications.

Serial	Average scores	Score indication
EQ1	7.4	Very good
EQ2	7.8	Very good
EQ3	7.2	Very good
EQ4	7.4	Very good change contexts
EQ5	6.8	Good semantic meaning
EQ6	7.6	Very contextual meaning
EQ7	5.2	Considerable intervention for proper sentence

Table 8.9: Execution time for all phases of DDARTS, 1KLOC = 1000 Lines of code.

Project	Revisions	Execution time	Each Revision Size
Hibernate	52	6.34 seconds	$\sim$ 263 KLOC
Speedment	243	35.8 seconds	$\sim$ 407 KLOC
AzureSDK	276	136 seconds	$\sim$ 6164KLOC (6MLOC)

8.9 Limitations and Threats to validity

Require paraphrasing or ordering the generated words. One major issue in our experiment is proper SSC extraction. However, 90-100% precision is reported by Mondal et al. [173]. We also check our additional three SSCs with their train set and found around 100% accuracy, meaning that at least those instances are available. Although it is infeasible to get the exact number of presence of them, it reduces the threat significantly. Dataset annotation is biased to human perception. However, two authors annotated independently and, in some cases, clarification from developers, which mitigates the threats. Another threat remains in the generalizability of the collected projects. Our collected projects are built with Java and Kotlin. Since the SSC properties are not context-dependent, mining modules with the existing tools (such as Bunch [155], MojoFM [245], ACDC, ARC [213], etc. for C/C++) would facilitate the SSC extraction in other coding platforms. Another threat remains in the quality of change description samples. Since they are written by the developers, these samples are reliable. Other threats remain the unbalanced collection of change groups. However, we experimented with the balanced training and test data and found approximate results with 2-5 points variations. Thus, it reduces the unbalanced samples threat. We also experiment with 10-fold cross validation of sscArchiNet outcomes, which do not vary significantly (4 points fewer). Thus, it reduces the model over-fitting threats to a certain extent.

8.10 Conclusion

In this study, we explored the semantic code change relations (SSC) of the design impactful changes for generating design change artifacts (for both individual commits and a complete release). In particular, we prepared a benchmark dataset by manually analyzing around 100 change release logs, 100 log instances, and 50 change comments for experimenting with design change artifacts generation. Then, we extracted various types of information required by the developers to write such documents so that the researchers could focus more on the automatic generation of such documents. Given the scarcity of benchmark data, our prepared dataset will be used to enhance further research in this domain. Furthermore, we have proposed a lightweight technique for generating descriptive summaries and release change logs of the design changes. Performance analysis of our technique with the ROUGE metric shows that the change grouping models and SSCs are promising (50% precision) with the proposed technique. Furthermore, human evaluation of our tool's outcome also indicates a very promising result. In addition, our tool's processing time is scalable with general-purpose computing. In conclusion, it will be an interesting work to collect and process a large collection of design change logs and explore the neural machine translator [239] embedding the thematic knowledge from the commit messages with the SSCs for generating more accurate sentences.

Chapter 9

Conclusion

9.1 Concluding Remarks

Design inconsistency and lack of pertinent information about the design of software systems create hurdles in making decisions and implementing changes. As a result, software bugs are produced, and security backdoors are created [257, 154, 243, 167, 217, 189, 175]. Development and maintenance teams also suffer various unnecessary mental pressures. All these things have severe consequences for the economy and people's daily life. Proper design documents need to be generated through architectural change detection, categorization, and change description generation to mitigate these problems. However, many research challenges exist in the literature for developing automated techniques for these phases (Chapter 1, Section 1.2). According to our systematic literature review, some of the crucial challenges are as follows:

First, a lack of sufficient datasets affects the deep insights extraction and performance of the architectural change information analysis [34]. Unfortunately, annotated datasets for high-level architectural change revisions are unavailable for research. It requires substantial experts human efforts to annotate a good quantity of samples.

Second, most of the existing studies consider either conversion of byte code or other formats of architectural definition coding as the primary step of architecture change detection [24, 245, 253, 65, 29, 69, 12]. As a result, they are either heavyweight or require substantial human intervention. Thus, it a may take several hours to a few years for extracting a design change [150]. However, 88.9% of architecture information is written as natural language texts whose size does not vary significantly from project to project [64, 115]. But, no study focused on textual properties for developing lightweight techniques.

Third, existing change detection studies overlooked directory, naming structure properties, and string patterns in the code change information returned by the *diff* tool of each codebase revision for extracting and coding the design architecture of a system. Existing studies that focus on implemented architecture process all the source code of each change revision. As a result, the execution time is several hours for each revision for extracting the high-level design change [130, 81, 129, 151, 150].

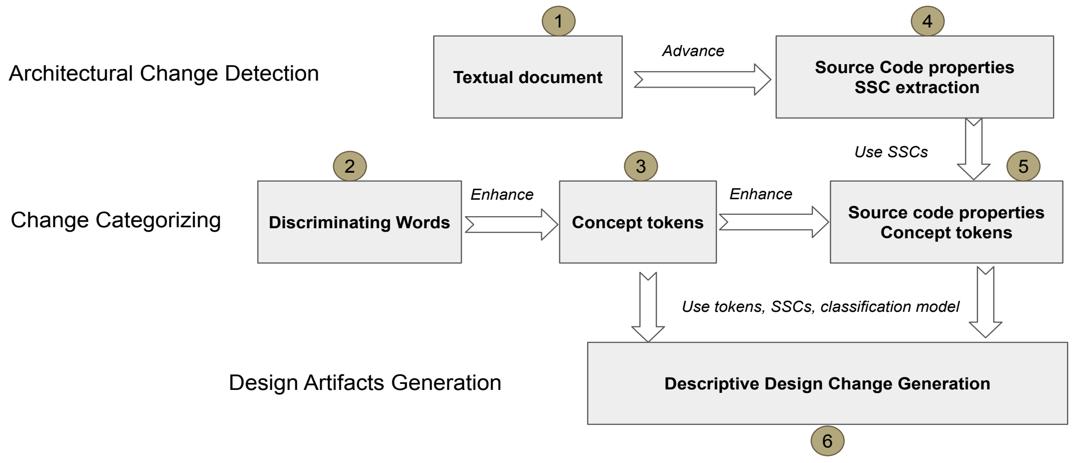
Fourth, existing studies are typical change commits categorizations that consider traditional text properties and source code properties [101, 95, 160, 86, 255, 135, 104]. Consequently, adapting them for architectural context can not handle the commit triad (tangled, ambiguous, and non-informative or empty message) challenges properly. However, they consider neither more intuitive semantic tokens/themes nor semantic code change relations properties for commit triad handling in proper architectural change categorization.

Fifth, some excellent studies exist that automatically generate commit summary descriptions and release notes [44, 144, 112, 239, 148, 178, 120, 33]. However, change summaries and change logs for high-level design changes are not covered by them due to cost, efforts and substantial experts intervention. Thus they are not helpful in getting high-level design change information and impacts.

In this thesis, we have proposed several approaches for architectural change detection, categorization, and documentation. We have explored various textual properties and code change properties from new perspectives for solving the mentioned challenges. In our first textual approach for architectural change detection (Chapter 3), we extract key terms and co-occurred terms for expressing architectural changes [172]. Our extracted co-occurred terms and change detection technique leveraging them is lightweight and does not interact with the codebase. Thus, it can be deployed to large-scale datasets available in the software repositories (i.e., GitHub, BitBucket, etc.) for empirical studies on architectural change analysis, design review issues, and concern mining. This study also reveals that the developer's descriptions (including code review comments) contain natural language properties that are the indicators of architectural changes to a significant extent. In another study for architectural change detection (Chapter 6), we found that directory and naming structure properties of the software codebase can be leveraged for efficient and lightweight detection of the implemented (descriptive) architecture [173]. However, our exploratory study reveals that for architectural change categorization (Chapter 4), the discriminating set of keywords is not reliable. In contrast, the concept tokens present within the commit messages are promising and can be deployed with structural code change relations (SSC) for determining the change purposes (Chapters 5 and 7). In our final study in Chapter 8, we have proposed an automated system for various design change document generation using structural code change relations, change purpose prediction model, and concept tokens. Our case study with the change summaries and design change logs of the Microsoft Azure SDK project shows encouraging results. Overall, our proposed approaches in this thesis are readily applicable to develop tools for design change traceability, features and design component mapping, architectural versioning scheme, decision making on project release activities, developers profile buildup, code review, change summary for the developers, design document updating, software documentation, release notes generation, software change analysis, and so on.

9.2 Discussions

The studies of this thesis are conducted in addressing the overall goals of the thesis (discussed in Chapter 1) and are related as of Fig. 9.1. In this thesis, study 4 is a complementary study of study 1 for architectural change detection. Furthermore, studies 2, 3, and 5 for change categorization are subsequently enhanced. While study 5 used the dataset and SSCs of study 4 and concept tokens from study 3. On the other hand, our final study (6) utilized concept tokens from study 3, SSCs from study 4, and the classification model



Architectural Change Detection
Change Categorizing
Design Artifacts Generation
1
Textual document
Advance
4
Source Code properties SSC extraction
Use SSCs
2
Discriminating Words
Enhance
3
Concept tokens
Enhance
5
Source code properties Concept tokens
Use tokens, SSCs, classification model
Descriptive Design Change Generation
6

Figure 9.1: Relation mapping of our studies.

from study 5. Another important thing to mention is that most of our studies are foundational to this particular topic. Moreover, we have created datasets to solve the challenges of architectural change detection, classification, and summary generation. These datasets are summarized in Table 9.1. We have invested great efforts in creating them and made them available to the public. With them, we also provide our extracted 130 co-occurred terms (with 166 keywords) that express architectural changes, a set of keywords for four types of architectural changes, and 293 concept tokens for architectural change categories. That said, the researchers can start exploring insights and leveraging them without much effort. Thus, these datasets can be used as the benchmark to advance research in this domain further.

9.3 Limitations and Future Work

Most of the studies in the thesis are explored for the first time in the context of descriptive design change artifacts generation. Naturally, some challenges remained unresolved. There are many opportunities to enhance our proposed approaches considering design change traceability, feature, and change mapping, design change versioning scheme, release management, descriptive change summary generation, code review, and so on. Following sections discuss some of the limitations of our conducted studies and future works.

1. Our proposed textual technique for determining architectural changes is based on the co-occurred key terms extraction. Those are extracted from a small dataset and might not cover a broader area because collecting developer's discussions that actually triggered architectural changes in the codebase is a challenging and time-consuming task. Therefore, collecting a larger collection of samples would facilitate covering key terms for broader contexts and better techniques. Moreover, our explored techniques' outcome is promising but not excellent (around 61% F1). Another major challenge in textual approaches is that many samples contain tacit and ambiguous descriptions in the software engineering contexts.

Table 9.1: Our prepared datasets

Approach	**# Projects**	**Project type**	**#Samples**	**Data type**	**Available Source**
S1:ACCOTERM	5	OSS, CV	1350	Textual messages, architectural changes	github.com/akm523/ scam-renedata
S2: ArchDFM	5	OSS, CV	362	Four architectural change types	github.com/akm523/ scam-renedata
S3: ArchiNet	5	OSS, CV	1133	Four architectural change types	github.com/akm523/ archinet
S4: ArchSlice	10	OSS, CV	3547	Code revision, Archi Changes	github.com/akm523/ arch-slice
S5: ArchSSC	8	OSS, CV	2697	Four architectural change types	tinyurl.com/yjpkws3s
S6: DDARTS	2	OSS, CV	214	Comment summaries, design change release logs	tinyurl.com/4w5yd6d6

To that end, concept mining approaches can be explored for developing better techniques since the developer's intention is embedded within the natural language text.

2. Our proposed technique for detecting implemented architectural changes using DANS properties covers high-level changes for the Java and Kotlin projects. However, these DANS properties may vary for projects developed with other coding platforms, and our tool may not be suitable for them. Therefore, DANS properties needed to be studied and enhanced for supporting Python, C/C++, and so on.

3. Our SSC extraction technique only considered the high-level concrete modules defined in Java and Kotlin. But, modules and packages have different structures and conventions for Python, C/C++, and others. Moreover, SSC cannot be extracted where high-level modules are not defined. However, a few heavy-weight approaches ([150, 213]) are available to extract approximate high-level modules leveraging the clustering techniques, and those tools can be extended to extract SSCs.

4. For architectural change categorization, concept tokens are the most promising approach. However, our extracted concept tokens are from a limited set of change samples from open-source projects, which might not cover a broader area. Therefore, extracting more concept tokens from a larger collection of commit messages, including commercial projects, could enhance our model. Moreover, a better natural language approach with code change concept mining could enhance the change purpose determining technique.

5. In the case study of descriptive design change artifacts, we analyzed a small collection of samples (due to the scarcity of representative projects) of change summary comments and release logs of design impactful changes. Therefore, it may not cover a broader context, and some challenges might have been undetected for generating a proper summary according to the developer's need. In future, researchers should collect and analyze more samples covering more OSS and commercial projects.

6. Our preliminary study for design change document generation considered information about change purposes, semantic change relations, and short commit theme. However, more thematic information extracted from commit messages, issue descriptions, and code concepts can be leveraged for more proper change description generation. That said, it will be interesting to develop a neural machine translator with this information for more accurate description generation.

7. Our developed tool for design change document produces approximate but incomplete sentences for change description (part a) in Fig. 8.8) for many cases that requires human intervention to make it perfect before final use. Thus, there are opportunities for further research works to enhance our tool for more proper and complete sentence generation.

References

[1] DNL Text Classification Basic. www.tensorflow.org/text/tutorials/text_classification_rnn.

[2] Github projects:. https://thenextweb.com/news/github-now-hosts-over-100-million-repositories.

[3] Llda program: https://github.com/taskehamano/llda.

[4] Plsa program: https://github.com/laserwave/plsa.

[5] Pydriller: github.com/ishepard/pydriller.

[6] RNN Basic. https://www.analyticsvidhya.com/blog/2022/03/a-brief-overview-of-recurrent-neural-networks-rnn.

[7] Semilda program: https://github.com/fancyspeed/semi-lda/tree/master/python.

[8] Sentiwordnet: http://sentiwordnet.isti.cnr.it/. June, 2019.

[9] Study artifacts: https://github.com/akm523/scamrenedata. June, 2019.

[10] Tensorflow: www.tensorflow.org/tutorials/text.

[11] Ieee recommended practice for architectural description of software-intensive systems. *IEEE Std 1471-2000*, pages i–23, 2000.

[12] Amjad AbuHassan and Mohammad Alshayeb. A metrics suite for uml model stability. *Software & Systems Modeling*, 18(1):557–583, 2019.

[13] Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C Shepherd. Software documentation: the practitioners' perspective. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 590–601. IEEE, 2020.

[14] Aakash Ahmad, Pooyan Jamshidi, Muteer Arshad, and Claus Pahl. Graph-based implicit knowledge discovery from architecture change logs. In *Proceedings of the WICSA/ECSA 2012 Companion Volume*, pages 116–123. ACM, 2012.

[15] Aakash Ahmad, Pooyan Jamshidi, and Claus Pahl. Graph-based pattern identification from architecture change logs. In *Proc. of CAiSE*, pages 200–213, 2012.

[16] Mamdouh Alenezi. Software architecture quality measurement stability and understandability. *IJACSA*, 2016.

[17] Nicolli SR Alves, Thiago S Mendes, Manoel G de Mendonça, Rodrigo O Spínola, Forrest Shull, and Carolyn Seaman. Identification and management of technical debt: A systematic mapping study. *Information and Software Technology*, 70:100–121, 2016.

[18] Aoin. :github.com/aionnetwork/aion, 2020.

[19] Elvira Maria Arvanitou, Apostolos Ampatzoglou, Konstantinos Tzouvalidis, Alexander Chatzigeorgiou, Paris Avgeriou, and Ignatios Deligiannis. Assessing change proneness at the architecture level: An empirical validation. In *2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW)*, pages 98–105. IEEE, 2017.

[20] Atrium. : github.com/robstoll/atrium, 2020.

[21] Stefano Baccianella, Andrea Esuli, and Fabrizio Sebastiani. Sentiwordnet 3.0: an enhanced lexical resource for sentiment analysis and opinion mining. In *Proceedings of the Seventh conference on International Language Resources and Evaluation*, volume 10, pages 2200–2204, 2010.

[22] Felix Bachmann, Len Bass, Paul Clements, David Garlan, James Ivers, M. Little, Paulo Merson, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, second edition, 2010.

[23] Rami Bahsoon and Wolfgang Emmerich. Architectural stability. In *Proc. of OTM*, pages 304–315, 2009.

[24] Carliss Young Baldwin and Kim B Clark. *Design rules: The power of modularity*, volume 1. MIT press, 2000.

[25] Carliss Young Baldwin, Kim B Clark, Kim B Clark, et al. *Design rules: The power of modularity*, volume 1. MIT press, 2000.

[26] Olivier Barais, Anne Françoise Le Meur, Laurence Duchien, and Julia Lawall. Software architecture evolution. In *Software Evolution*, pages 233–262. Springer, 2008.

[27] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003.

[28] Pooyan Behnamghader, Reem Alfayez, Kamonphop Srisopha, and Barry Boehm. Towards better understanding of software quality evolution through commit-impact analysis. In *Proc. of QRS*, pages 251–262, 2017.

[29] Ameni ben Fadhel, Marouane Kessentini, Philip Langer, and Manuel Wimmer. Search-based detection of high-level model changes. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 212–221. IEEE, 2012.

[30] Gunnar R Bergersen, Dag IK Sjøberg, and Tore Dybå. Construction and validation of an instrument for measuring programming skill. *IEEE Transactions on Software Engineering*, 40(12):1163–1184, 2014.

[31] Marcello M Bersani, Francesco Marconi, Damian A Tamburri, Pooyan Jamshidi, and Andrea Nodari. Continuous architecting of stream-based systems. In *Proc. of WICSA*, pages 146–151. IEEE, 2016.

[32] Manoj Bhat, Klym Shumaiev, Uwe Hohenstein, Andreas Biesdorf, and Florian Matthes. The evolution of architectural decision making as a key focus area of software architecture research: A semi-systematic literature study. In *2020 IEEE International Conference on Software Architecture (ICSA)*, pages 69–80. IEEE, 2020.

[33] T. Bi, X. Xia, D. Lo, J. Grundy, and T. Zimmermann. An empirical study of release note production and usage in practice. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[34] Tingting Bi, Peng Liang, Antony Tang, and Chen Yang. A systematic mapping study on text analysis techniques in software architecture. *Journal of Systems and Software*, 144:533–558, 2018.

[35] Nate Black. Nicolai parlog on java 9 modules. *IEEE Software*, (3):101–104, 2018.

[36] Roi Blanco and Christina Lioma. Graph-based term weighting for information retrieval. *IR*, pages 54–92, 2012.

[37] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3:993–1022, 2003.

[38] Mikael Boden. A guide to recurrent neural networks and backpropagation. *the Dallas project*, 2(2):1–10, 2002.

[39] Jan Bosch. Software architecture: The next step. In *European Workshop on Software Architecture*, pages 194–199. Springer, 2004.

[40] Eric Bouwers, Jose Pedro Correia, Arie van Deursen, and Joost Visser. Quantifying the analyzability of software architectures. In *2011 Ninth Working IEEE/IFIP Conference on Software Architecture*, pages 83–92. IEEE, 2011.

[41] Ivan T Bowman, Richard C Holt, and Neil V Brewster. Linux as a case study: Its extracted software architecture. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002)*, pages 555–563. IEEE, 1999.

[42] Peter F. Brown, Peter V. deSouza, Robert L. Mercer, Vincent J. Della Pietra, and Jenifer C. Lai. Class-based n-gram models of natural language. *Comput. Linguist.*, 18(4):467–479, December 1992.

[43] Bach-Java Shell Builder. : github.com/sormuras/bach, 2020.

[44] Raymond PL Buse and Westley R Weimer. Automatically documenting program changes. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 33–42, 2010.

[45] Raymond PL Buse and Thomas Zimmermann. Information needs for software development analytics. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 987–996. IEEE, 2012.

[46] Yuanfang Cai and Kevin J. Sullivan. Modularity analysis of logical design models. In *Proc. of ASE*, pages 91–102, 2006.

[47] Yuanfang Cai and Kevin J Sullivan. Modularity analysis of logical design models. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 91–102. IEEE, 2006.

[48] William M Campbell and Fred S Richardson. Discriminative keyword selection using support vector machines. In *NIPS*, pages 209–216. Citeseer, 2007.

[49] Ivan Candela, Gabriele Bavota, Barbara Russo, and Rocco Oliveto. Using cohesion and coupling for software remodularization: Is it enough? *TOSEM*, 25(3):24, 2016.

[50] Gerardo Canfora, Massimiliano Di Penta, and Luigi Cerulo. Achievements and challenges in software reverse engineering. *Communications of the ACM*, 54(4):142–151, 2011.

[51] Rafael Capilla, Anton Jansen, Antony Tang, Paris Avgeriou, and Muhammad Ali Babar. 10 years of software architecture knowledge management: Practice and future. *JSS*, pages 191 – 205, 2016.

[52] Jeromy Carriere, Rick Kazman, and Ipek Ozkaya. A cost-benefit framework for making architectural decisions in a business context. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 2, pages 149–157. IEEE, 2010.

[53] Maria Caulo, Bin Lin, Gabriele Bavota, Giuseppe Scanniello, and Michele Lanza. Knowledge transfer in modern code review. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 230–240, 2020.

[54] Debasish Chakroborti, Kevin A Schneider, and Chanchal K Roy. Backports: Change types, challenges and strategies. In *International Conference on Program Comprehension*, 2022.

[55] Ned Chapin, Joanne E Hale, Khaled Md Khan, Juan F Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of software maintenance and evolution: Research and Practice*, 13(1):3–30, 2001.

[56] Paul Clements, David Garlan, Reed Little, Robert Nord, and Judith Stafford. Documenting software architectures: views and beyond. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 740–741. IEEE, 2003.

[57] M. Codoban, S. S. Ragavan, D. Dig, and B. Bailey. Software history under the lens: A study on why and how developers examine it. In *Proc. of the 2015 ICSME*, pages 1–10, 2015.

[58] Azure SDK commit. : ..azure-sdk-for-java/commit/ 7da63b6374005efe6dadbfb4e46f956e64e535a0, 2020.

[59] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[60] Melis Dagpinar and Jens H Jahnke. Predicting maintainability with object-oriented metrics-an empirical comparison. In *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.*, pages 155–155. IEEE Computer Society, 2003.

[61] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. Untangling fine-grained code changes. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 341–350. IEEE, 2015.

[62] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated detection of refactorings in evolving components. In *European conference on object-oriented programming*, pages 404–428. Springer, 2006.

[63] Wei Ding, Peng Liang, Antony Tang, and Hans Van Vliet. Causes of architecture changes: An empirical study through the communication in oss mailing lists. In *SEKE*, pages 403–408, 2015.

[64] Wei Ding, Peng Liang, Antony Tang, Hans Van Vliet, and Mojtaba Shahin. How do open source communities document software architecture: An exploratory survey. In *19th International conference on engineering of complex computer systems*, pages 136–145, 2014.

[65] Xinyi Dong and Michael W Godfrey. Identifying architectural change patterns in object-oriented systems. In *2008 16th IEEE International Conference on Program Comprehension*, pages 33–42. IEEE, 2008.

[66] Jugendra Dongre, Gend Lai Prajapati, and SV Tokekar. The role of apriori algorithm for finding the association rules in data mining. In *2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, pages 657–660. IEEE, 2014.

[67] Natalia Dragan, Michael L Collard, Maen Hammad, and Jonathan I Maletic. Using stereotypes to help characterize commits. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 520–523. IEEE, 2011.

[68] Stephan Dreiseitl and Lucila Ohno-Machado. Logistic regression and artificial neural network classification models: a methodology review. *Journal of biomedical informatics*, 35(5-6):352–359, 2002.

[69] Darko Durisic, Martin Nilsson, Miroslaw Staron, and Jörgen Hansson. Measuring the impact of changes to the complexity and coupling properties of automotive software systems. *Journal of Systems and Software*, 86(5):1275–1293, 2013.

[70] Darko Durisic, Miroslaw Staron, and Martin Nilsson. Measuring the size of changes in automotive software systems and their impact on product quality. In *Proceedings of the 12th International Conference on Product Focused Software Development and Process Improvement*, pages 10–13, 2011.

[71] Fatih Ertam and Galip Aydın. Data classification with deep learning using tensorflow. In *2017 International Conference on Computer Science and Engineering (UBMK)*, pages 755–758. IEEE, 2017.

[72] Jacky Estublier, David Leblang, André van der Hoek, Reidar Conradi, Geoffrey Clemm, Walter Tichy, and Darcy Wiborg-Weber. Impact of software engineering research on the practice of software configuration management. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(4):383–430, 2005.

[73] Saad Ezzini, Sallam Abualhaija, Chetan Arora, Mehrdad Sabetzadeh, and Lionel C Briand. Using domain-specific corpora for improved handling of ambiguity in requirements. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1485–1497. IEEE, 2021.

[74] B. Fluri, E. Giger, and H. C. Gall. Discovering patterns of change types. In *Proc. of ASE*, pages 463–466, 2008.

[75] Beat Fluri and Harald C Gall. Classifying change types for qualifying change couplings. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 35–45. IEEE, 2006.

[76] Azure SDK for Java. : github.com/azure/azure-sdk-for-java, 2020.

[77] Nicole Forsgren, Margaret-Anne Storey, Chandra Maddila, Thomas Zimmermann, Brian Houck, and Jenna Butler. The space of developer productivity: There's more to it than you think. *Queue*, 19(1):20–48, 2021.

[78] M. Fowler. Design-who needs an architect? *IEEE Software*, pages 11–13, 2003.

[79] Mark A Friedl and Carla E Brodley. Decision tree classification of land cover from remotely sensed data. *Remote sensing of environment*, 61(3):399–409, 1997.

[80] Ying Fu, Meng Yan, Xiaohong Zhang, Ling Xu, Dan Yang, and Jeffrey D Kymer. Automated classification of software change messages by semi-supervised latent dirichlet allocation. *IST*, 57:369–377, 2015.

[81] Joshua Garcia, Igor Ivkovic, and Nenad Medvidovic. A comparative analysis of software architecture recovery techniques. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 486–496. IEEE, 2013.

[82] Joshua Garcia, Mehdi Mirakhorli, Lu Xiao, Yutong Zhao, Ibrahim Mujhid, Khoi Pham, Ahmet Okutan, Sam Malek, Rick Kazman, Yuanfang Cai, et al. Constructing a shared infrastructure for software architecture analysis and maintenance. In *2021 IEEE 18th International Conference on Software Architecture (ICSA)*, pages 150–161. IEEE, 2021.

[83] Joshua Garcia, Daniel Popescu, Chris Mattmann, Nenad Medvidovic, and Yuanfang Cai. Enhancing architectural recovery using concerns. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 552–555. IEEE, 2011.

[84] David Garlan, Felix Bachmann, James Ivers, Judith Stafford, Len Bass, Paul Clements, and Paulo Merson. *Documenting Software Architectures: Views and Beyond.* Addison-Wesley Professional, 2nd edition, 2010.

[85] David Garlan and Mary Shaw. An introduction to software architecture. Technical report, Pittsburgh, PA, USA, 1994.

[86] Sirine Gharbi, Mohamed Wiem Mkaouer, Ilyes Jenhani, and Montassar Ben Messaoud. On the classification of software change messages using multi-label active learning. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 1760–1767, 2019.

[87] Negar Ghorbani, Joshua Garcia, and Sam Malek. Detection and repair of architectural inconsistencies in java. In *Proceedings of the 41st International Conference on Software Engineering*, pages 560–571. IEEE Press, 2019.

[88] GitPython. gitpython.readthedocs.io/en/stable, 2020.

[89] Cyril Goutte and Eric Gaussier. A probabilistic interpretation of precision, recall and f-score, with implication for evaluation. In *Proc. of ECIR*, pages 345–359. Springer, 2005.

[90] John Grundy and John Hosking. High-level static and dynamic visualisation of software architectures. In *Proceeding 2000 IEEE International Symposium on Visual Languages*, pages 5–12. IEEE, 2000.

[91] Philipp Haindl and Reinhold Plösch. Towards continuous quality: Measuring and evaluating feature-dependent non-functional requirements in devops. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 91–94. IEEE, 2019.

[92] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[93] Maen Hammad, Michael L Collard, and Jonathan I Maletic. Automatically identifying changes that impact code-to-design traceability. In *2009 IEEE 17th International Conference on Program Comprehension*, pages 20–29. IEEE, 2009.

[94] Ahmed E Hassan. Automated classification of change messages in open source projects. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 837–841, 2008.

[95] Lile P Hattori and Michele Lanza. On the nature of commits. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops*, pages 63–71. IEEE, 2008.

[96] Larry V Hedges. Distribution theory for glass's estimator of effect size and related estimators. *journal of Educational Statistics*, 6(2):107–128, 1981.

[97] Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 392–401. IEEE, 2013.

[98] Eben Hewitt. *Semantic Software Design: A New Theory and Practical Guide for Modern Architects*. O'Reilly Media, 2019.

[99] Abram Hindle, Neil A Ernst, Michael W Godfrey, and John Mylopoulos. Automated topic naming to support cross-project analysis of software maintenance activities. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 163–172, 2011.

[100] Abram Hindle, Daniel M German, Michael W Godfrey, and Richard C Holt. Automatic classication of large changes into maintenance categories. In *2009 IEEE 17th International Conference on Program Comprehension*, pages 30–39. IEEE, 2009.

[101] Abram Hindle, Daniel M German, and Ric 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*, pages 99–108, 2008.

[102] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[103] Benedikt Holmes and Ana Nicolaescu. Continuous architecting: Just another buzzword? *Full-scale Software Engineering*, page 1, 2017.

[104] Sebastian Hönel, Morgan Ericsson, Welf Löwe, and Anna Wingkvist. Using source code density to improve the accuracy of automatic commit classification into maintenance activities. *Journal of Systems and Software*, page 110673, 2020.

[105] Guang-Bin Huang, Yan-Qiu Chen, and Haroon A Babri. Classification ability of single hidden layer feedforward neural networks. *IEEE transactions on neural networks*, 11(3):799–801, 2000.

[106] ImGui. : github.com/kotlin-graphics/imgui, 2020.

[107] May I.S.O. Systems and software engineering–architecture description. Technical report, ISO/IEC/IEEE 42010, 2011.

[108] P. Jamshidi, M. Ghafari, A. Ahmad, and C. Pahl. A framework for classifying and comparing architecture-centric software evolution research. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, pages 305–314, 2013.

[109] Anton Jansen and Jan Bosch. Software architecture as a set of architectural design decisions. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, pages 109–120, 2005.

[110] Anton Jansen, Jan Bosch, and Paris Avgeriou. Documenting after the fact: Recovering architectural design decisions. *Journal of Systems and Software*, 81(4):536–557, 2008.

[111] Shuyao Jiang. Boosting neural commit message generation with code semantic analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1280–1282. IEEE, 2019.

[112] Siyuan Jiang, Ameer Armaly, and Collin McMillan. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 135–146. IEEE, 2017.

[113] Siyuan Jiang and Collin McMillan. Towards automatic generation of short summaries of commits. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 320–323. IEEE, 2017.

[114] Siyuan Jiang and Collin McMillan. Towards automatic generation of short summaries of commits. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 320–323. IEEE, 2017.

[115] Rick Kazman, Dennis Goldenson, Ira Monarch, William Nichols, and Giuseppe Valetto. Evaluating the effects of architectural documentation: A case study of a large scale open source project. *Transactions on SE*, pages 220–260, 2016.

[116] Stefan Kehrer, Florian Riebandt, and Wolfgang Blochinger. Container-based module isolation for cloud services. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 177–17709. IEEE, 2019.

[117] Safoora Shakil Khan, Phil Greenwood, Alessandro Garcia, and Awais Rashid. On the impact of evolving requirements-architecture dependencies: An exploratory study. In *International Conference on Advanced Information Systems Engineering*, pages 243–257. Springer, 2008.

[118] Mark Kishlansky et al. How to read a document. *Sources of the West: Readings in Western Civilization*, 1991.

[119] Sebastian Klepper, Stephan Krusche, and Bernd Bruegge. Semi-automatic generation of audience-specific release notes. In *Proc. of CSED*, pages 19–22, 2016.

[120] Sebastian Klepper, Stephan Krusche, and Bernd Brügge. Semi-automatic generation of audience-specific release notes. In *CSED@ICSE*, 2016.

[121] Kristjan Korjus, Martin N Hebart, and Raul Vicente. An efficient data partitioning to improve classification performance while keeping parameters interpretable. *PloS one*, 11(8):e0161788, 2016.

[122] Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital investigation*, 3:91–97, 2006.

[123] Oleksandr Kosenkov, Michael Unterkalmsteiner, Daniel Mendez, and Davide Fucci. Vision for an artefact-based approach to regulatory requirements engineering. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6, 2021.

[124] Patrick Kreutzer, Georg Dotzler, Matthias Ring, Bjoern M Eskofier, and Michael Philippsen. Automatic clustering of code changes. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 61–72. IEEE, 2016.

[125] Philippe Kruchten. An ontology of architectural design decisions in software intensive systems. In *2nd Groningen workshop on software variability*, pages 54–61. Citeseer, 2004.

[126] Philippe Kruchten. *The rational unified process: an introduction.* Addison-Wesley Professional, 2004.

[127] Zijad Kurtanović and Walid Maalej. Automatically classifying functional and non-functional requirements using supervised machine learning. In *2017 IEEE 25th International Requirements Engineering Conference (RE)*, pages 490–495. Ieee, 2017.

[128] Siwei Lai, Liheng Xu, Kang Liu, and Jun Zhao. Recurrent convolutional neural networks for text classification. In *Twenty-ninth AAAI conference on artificial intelligence*, 2015.

[129] Duc Le and Nenad Medvidovic. Architectural-based speculative analysis to predict bugs in a software system. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 807–810, 2016.

[130] 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 *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 235–245. IEEE, 2015.

[131] 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 *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 235–245. IEEE, 2015.

[132] Manny M Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124. Springer, 1996.

[133] Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[134] Meir M Lehman and Juan F Ramil. Software evolution and software evolution processes. *Annals of Software Engineering*, 14(1-4):275–309, 2002.

[135] Stanislav Levin and Amiram Yehudai. Using temporal and semantic developer-level information to predict maintenance activity profiles. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 463–467. IEEE, 2016.

[136] Stanislav Levin and Amiram 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*, pages 97–106, 2017.

[137] Wenmin Li, Jiawei Han, and Jian Pei. Cmar: Accurate and efficient classification based on multiple class-association rules. In *Proceedings 2001 IEEE international conference on data mining*, pages 369–376. IEEE, 2001.

[138] Yi Li, Chenguang Zhu, Milos Gligoric, Julia Rubin, and Marsha Chechik. Precise semantic history slicing through dynamic delta refinement. *Automated Software Engineering*, 26(4):757–793, 2019.

[139] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. Semantic slicing of software version histories. *IEEE Transactions on Software Engineering*, 44(2):182–201, 2017.

[140] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. Semantic slicing of software version histories. *IEEE Transactions on Software Engineering*, 44(2):182–201, 2017.

[141] Yong H Li and Anil K Jain. Classification of text documents. *The Computer Journal*, 41(8):537–546, 1998.

[142] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81, 2004.

[143] I-H Lin and David A Gustafson. Classifying software maintenance. In *Proceedings. Conference on Software Maintenance, 1988.*, pages 241–247. IEEE, 1988.

[144] Mario Linares-Vásquez, Luis Fernando Cortés-Coy, Jairo Aponte, and Denys Poshyvanyk. Changescribe: A tool for automatically generating commit messages. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 709–712. IEEE, 2015.

[145] Kurt R Linberg. Software developer perceptions about software project failure: a case study. *Journal of Systems and Software*, 49(2-3):177–192, 1999.

[146] Dimitris Liparas, Yaakov HaCohen-Kerner, Anastasia Moumtzidou, Stefanos Vrochidis, and Ioannis Kompatsiaris. News articles classification using random forests and weighted multimodal features. In *Information Retrieval Facility Conference*, pages 63–75. Springer, 2014.

[147] Qin Liu, Zihe Liu, Hongming Zhu, Hongfei Fan, Bowen Du, and Yu Qian. Generating commit messages from diffs using pointer-generator network. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 299–309. IEEE, 2019.

[148] Shangqing Liu, Cuiyun Gao, Sen Chen, Nie Lun Yiu, and Yang Liu. Atom: Commit message generation based on abstract syntax tree and hybrid ranking. *IEEE Transactions on Software Engineering*, 2020.

[149] Yue Lu, Qiaozhu Mei, and ChengXiang Zhai. Investigating task performance of probabilistic topic models: an empirical study of plsa and lda. *IR*, pages 178–203, 2011.

[150] Thibaud Lutellier, Devin Chollak, Joshua Garcia, Lin Tan, Derek Rayside, Nenad Medvidovic, and Robert Kroeger. Comparing software architecture recovery techniques using accurate dependencies. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 69–78. IEEE, 2015.

[151] Thibaud Lutellier, Devin Chollak, Joshua Garcia, Lin Tan, Derek Rayside, Nenad Medvidovic, and Robert Kroeger. Comparing software architecture recovery techniques using accurate dependencies. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 69–78. IEEE, 2015.

[152] Haohai Ma, Weizhong Shao, Lu Zhang, Zhiyi Ma, and Yanbing Jiang. Applying oo metrics to assess uml meta-models. In *International Conference on the Unified Modeling Language*, pages 12–26. Springer, 2004.

[153] Sander Mak and Paul Bakker. *Java 9 Modularity: Patterns and Practices for Developing Maintainable Applications.* " O'Reilly Media, Inc.", 2017.

[154] Pratyusa K Manadhata and Jeannette M Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 37(03):371–386, 2011.

[155] Spiros Mancoridis, Brian S Mitchell, Yihfarn Chen, and Emden R Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pages 50–59. IEEE, 1999.

[156] Spiros Mancoridis, Brian S Mitchell, Chris Rorres, Y Chen, and Emden R Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No. 98TB100242)*, pages 45–52. IEEE, 1998.

[157] Richard VR Mariano, Geanderson E dos Santos, Markos V de Almeida, and Wladmir 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)*, pages 515–518. IEEE, 2019.

[158] Antonio Martini and Jan Bosch. A multiple case study of continuous architecting in large agile companies: current gaps and the caffea framework. In *13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 1–10, 2016.

[159] Andreas Mauczka, Florian Brosch, Christian Schanes, and Thomas Grechenig. Dataset of developer-labeled commit messages. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 490–493. IEEE, 2015.

[160] Andreas Mauczka, Markus Huber, Christian Schanes, Wolfgang Schramm, Mario Bernhart, and Thomas Grechenig. Tracing your maintenance work–a cross-project validation of an automated classification dictionary for commit messages. In *International Conference on Fundamental Approaches to Software Engineering*, pages 301–315. Springer, 2012.

[161] Robert McGill, John W Tukey, and Wayne A Larsen. Variations of box plots. *The american statistician*, 32(1):12–16, 1978.

[162] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving software architecture descriptions of critical systems. *Computer*, 43(5):42–48, 2010.

[163] Ronaldo Messina and Jerome Louradour. Segmentation-free handwritten chinese text recognition with lstm-rnn. In *2015 13th International Conference on Document Analysis and Recognition (ICDAR)*, pages 171–175. IEEE, 2015.

[164] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[165] Tomáš Mikolov et al. Statistical language models based on neural networks. *Presentation at Google, Mountain View, 2nd April*, 80(26), 2012.

[166] Mehdi Mirakhorli, Yonghee Shin, Jane Cleland-Huang, and Murat Cinar. A tactic-centric approach for automating traceability of quality concerns. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 639–649. IEEE, 2012.

[167] Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, and Qiong Feng. Architecture anti-patterns: Automatically detectable violations of design principles. *IEEE Transactions on Software Engineering*, 2019.

[168] Audris Mockus and Lawrence G Votta. Identifying reasons for software changes using historic databases. In *International Conference on Software Maintenance*, pages 120–130, 2000.

[169] Audris Mockus and Lawrence G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance*, pages 120–130, 2000.

[170] Parastoo Mohagheghi and Reidar Conradi. An empirical study of software change: origin, acceptance rate, and functionality vs. quality attributes. In *Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. ISESE'04.*, pages 7–16. IEEE, 2004.

[171] Amit Kumar Mondal, Banani Roy, Sristy Sumana Nath, and Kevin A Schneider. A concept-token based approach for determining architectural change categories. In *Proceedings of the 33rd International Conference on Software Engineering & Knowledge Engineering*, pages 7–14, 2021.

[172] Amit Kumar Mondal, Banani Roy, and Kevin A Schneider. An exploratory study on automatic architectural change analysis using natural language processing techniques. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 62–73. IEEE.

[173] Amit Kumar Mondal, Chanchal K Roy, Kevin A Schneider, Banani Roy, and Sristy Sumana Nath. Semantic slicing of architectural change commits: Towards semantic design review. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6, 2021.

[174] Amit Kumar Mondal, Kevin A Schneider, Banani Roy, and Chanchal K Roy. A survey of software architectural change detection and categorization techniques. *Journal of System and Software (Under Revision)*, 2022.

[175] David Monschein, Manar Mazkatli, Robert Heinrich, and Anne Koziolek. Enabling consistency between software artefacts for software adaption and evolution. In *2021 IEEE 18th International Conference on Software Architecture (ICSA)*, pages 1–12. IEEE, 2021.

[176] Joao Eduardo Montandon, Luciana Lourdes Silva, and Marco Tulio Valente. Identifying experts in software libraries and frameworks among github users. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 276–287. IEEE, 2019.

[177] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. Automatic generation of release notes. In *Proc. of FSE*, pages 484–495, 2014.

[178] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. Arena: An approach for the automated generation of release notes. *IEEE Transactions on Software Engineering*, 43:106–127, 2017.

[179] Ward Muylaert and Coen De Roover. Untangling composite commits using program slicing. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 193–202. IEEE, 2018.

[180] MvvmFX. : github.com/sialcasa/mvvmfx, 2020.

[181] Taiga Nakamura and Victor R Basili. Metrics of software architecture changes based on structural distance. In *11th IEEE International Software Metrics Symposium (METRICS'05)*, pages 24–24. IEEE, 2005.

[182] Daye Nam, Youn Kyu Lee, and Nenad Medvidovic. Eva: A tool for visualizing software architectural evolution. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 53–56, 2018.

[183] Sristy Sumana Nath and Banani Roy. Towards automatically generating release notes using extractive summarization technique. In *International Conference on Software Engineering & Knowledge Engineering, SEKE 2021. Proceedings.*, pages 241–248, 2021.

[184] BODJE N'Kauh Nathan-Regis and NASIRA GM. Software architecture at the glance——to make a long story short.

[185] Najam Nazar, Yan Hu, and He Jiang. Summarizing software artifacts: A literature review. *JCST*, pages 883–909, 2016.

[186] Shiva Nejati, Mehrdad Sabetzadeh, Chetan Arora, Lionel C Briand, and Felix Mandoux. Automated change impact analysis between sysml models of requirements and design. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 242–253, 2016.

[187] Matías Nicoletti, J Andrés Diaz-Pace, and Silvia Schiaffino. Towards software architecture documents matching stakeholders' interests. In *International Conference on Advances in New Technologies, Interactive Interfaces, and Communicability*, pages 176–185. Springer, 2011.

[188] Arif Nurwidyantoro, Mojtaba Shahin, Michel Chaudron, Waqar Hussain, Harsha Perera, Rifat Ara Shams, and Jon Whittle. Towards a human values dashboard for software development: An exploratory study. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, 2021.

[189] Peyman Oreizy, Nenad Medvidovic, and Richard N Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th international conference on Software engineering*, pages 177–186. IEEE, 1998.

[190] Ksenia Oskina. Text classification in the domain of applied linguistics as part of a pre-editing module for machine translation systems. In *International Conference on Speech and Computer*, pages 691–698, 2016.

[191] Tosin Daniel Oyetoyan, Daniela S. Cruzes, and Reidar Conradi. A study of cyclic dependencies on defect profile of software components. *JSS*, pages 3162 – 3182, 2013.

[192] Ipek Ozkaya, Peter Wallin, and Jakob Axelsson. Architecture knowledge management during system evolution: observations from practitioners. In *Proceedings of the 2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge*, pages 52–59, 2010.

[193] Claus Pahl, Pooyan Jamshidi, and Danny Weyns. Cloud architecture continuity: Change models and change rules for sustainable cloud software architectures. *JSOFTW-EVOL*, page e1849, 2017.

[194] Matheus Paixao, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman. Are developers aware of the architectural impact of their changes? In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 95–105, 2017.

[195] Matheus Paixao, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman. The impact of code review on architectural changes. *IEEE Transactions on Software Engineering*, 2019.

[196] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[197] Profir-Petru Pârţachi, Santanu Kumar Dash, Miltiadis Allamanis, and Earl T Barr. Flexeme: Untangling commits using lexical flows. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 63–74, 2020.

[198] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software engineering notes*, pages 40–52, 1992.

[199] Marco Piccioni, Carlo A. Furia, and Bertrand Meyer. An empirical study of api usability. In *ESEM*, 2013.

[200] Jonas Poelmans, Dmitry I Ignatov, Sergei O Kuznetsov, and Guido Dedene. Formal concept analysis in knowledge processing: A survey on applications. *Expert systems with applications*, 40(16):6538–6560, 2013.

[201] Mohammad Masudur Rahman and Chanchal K Roy. Improving ir-based bug localization with context-aware query reformulation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 621–632. ACM, 2018.

[202] Daniel Ramage, David Hall, Ramesh Nallapati, and Christopher D Manning. Labeled lda: A supervised topic model for credit attribution in multi-labeled corpora. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 248–256, 2009.

[203] Daniel Ramage, David Hall, Ramesh Nallapati, and Christopher D. Manning. Labeled lda: A supervised topic model for credit attribution in multi-labeled corpora. In *Proc. of EMNLP*, pages 248–256, 2009.

[204] Sebastian Raschka. Mlxtend: Providing machine learning and data science utilities and extensions to python's scientific computing stack. *Journal of open source software*, 3(24):638, 2018.

[205] Ghulam Rasool and Nancy Fazal. Evolution prediction and process support of oss studies: a systematic mapping. *Arabian Journal for Science and Engineering*, 42(8):3465–3502, 2017.

[206] Sarah Rastkar and Gail C Murphy. Why did this code change? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1193–1196. IEEE, 2013.

[207] Mark Richards and Neal Ford. *Fundamentals of Software Architecture: An Engineering Approach.* O'Reilly, 2020.

[208] Martin P Robillard, Andrian Marcus, Christoph Treude, Gabriele Bavota, Oscar Chaparro, Neil Ernst, Marco Aurélio Gerosa, Michael Godfrey, Michele Lanza, Mario Linares-Vásquez, et al. On-demand developer documentation. In *2017 IEEE International conference on software maintenance and evolution (ICSME)*, pages 479–483. IEEE, 2017.

[209] Roshanak Roshandel, André Van Der Hoek, Marija Mikic-Rakic, and Nenad Medvidovic. Mae—a system model and environment for managing architectural evolution. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 13(2):240–276, 2004.

[210] Banani Roy, Amit Kumar Mondal, Chanchal K Roy, Kevin A Schneider, and Kawser Wazed. Towards a reference architecture for cloud-based plant genotyping and phenotyping analysis frameworks. In *Proc. of ICSA*, pages 41–50, 2017.

[211] Nick Rozanski and Eóin Woods. *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley, 2012.

[212] Jessica M Rudd et al. An empirical study of downstream analysis effects of model pre-processing choices. *Open journal of statistics*, 10(5):735–809, 2020.

[213] Marcelo Schmitt Laser, Nenad Medvidovic, Duc Minh Le, and Joshua Garcia. Arcade: an extensible workbench for architecture recovery, change, and decay evaluation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1546–1550, 2020.

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

[215] Gaganpreet Sharma. Pros and cons of different sampling techniques. *International journal of applied research*, 3(7):749–752, 2017.

[216] Jinfeng Shen, Xiaobing Sun, Bin Li, Hui Yang, and Jiajun Hu. On automatic summarization of what and why information in source code changes. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 103–112. IEEE, 2016.

[217] Marcelino Campos Oliveira Silva, Marco Tulio Valente, and Ricardo Terra. Does technical debt lead to the rejection of pull requests? In *in 12th Brazilian Symposium on Information Systems on Brazilian Symposium on Information Systems: Information Systems in the Cloud Computing Era*, pages 248–254, 2016.

[218] Kari Smolander. Four metaphors of architecture in software organizations: finding out the meaning of architecture in practice. In *Proceedings International Symposium on Empirical Software Engineering*, pages 211–221. IEEE, 2002.

[219] Richard Socher, Eric Huang, Jeffrey Pennin, Christopher D Manning, and Andrew Ng. Dynamic pooling and unfolding recursive autoencoders for paraphrase detection. *Advances in neural information processing systems*, 24, 2011.

[220] Mohamed Soliman, Amr Rekaby Salama, Matthias Galster, Olaf Zimmermann, and Matthias Riebisch. Improving the search for architecture knowledge in online developer communities. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 186–18609. IEEE, 2018.

[221] Kalyanasundaram Somasundaram and Gail C Murphy. Automatic categorization of bug reports using latent dirichlet allocation. In *Proceedings of the 5th India software engineering conference*, pages 125–130, 2012.

[222] Speedment. : github.com/speedment/speedment, 2020.

[223] Maximilian Steff and Barbara Russo. Measuring architectural change for defect estimation and localization. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 225–234. IEEE, 2011.

[224] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[225] Jeffrey Svajlenko and Chanchal K Roy. Cloneworks: A fast and flexible large-scale near-miss clone detection tool. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 177–179. IEEE Press, 2017.

[226] E Burton Swanson. The dimensions of maintenance. In *Proceedings of the 2nd international conference on Software engineering*, pages 492–497. IEEE Computer Society Press, 1976.

[227] Antony Tang and Man F Lau. Software architecture review by association. *Journal of systems and software*, 88:87–101, 2014.

[228] Antony Tang and Man F Lau. Software architecture review by association. *Journal of systems and software*, 88:87–101, 2014.

[229] Richard N Taylor, Nenad Medvidovic, and Eric Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[230] Vassilios Tzerpos and Richard C Holt. Mojo: A distance metric for software clusterings. In *Sixth Working Conference on Reverse Engineering (Cat. No. PR00303)*, pages 187–193. IEEE, 1999.

[231] Vassilios Tzerpos and Richard C Holt. Accd: an algorithm for comprehension-driven clustering. In *Proceedings Seventh Working Conference on Reverse Engineering*, pages 258–267. IEEE, 2000.

[232] Anderson Uchôa, Caio Barbosa, Daniel Coutinho, Willian Oizumi, Wesley KG Assunçao, Silvia Regina Vergilio, Juliana Alves Pereira, Anderson Oliveira, and Alessandro Garcia. Predicting design impactful changes in modern code review: A large-scale empirical study. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 471–482. IEEE, 2021.

[233] Rajesh Vasa, J-G Schneider, Clinton Woodward, and Andrew Cain. Detecting structural changes in object oriented software systems. In *2005 International Symposium on Empirical Software Engineering, 2005.*, pages 8–pp. IEEE, 2005.

[234] Roberto Verdecchia, Philippe Kruchten, Patricia Lago, and Ivano Malavolta. Building and evaluating a theory of architectural technical debt in software-intensive systems. *Journal of Systems and Software*, 176:110925, 2021.

[235] Vooga. : github.com/anna-dwish/vooga, 2020.

[236] Vooga. Hibernate search: github.com/hibernate/hibernate-search, 2020.

[237] Rob Waller. What makes a good document. *The criteria we use. Technical paper*, 2, 2011.

[238] Dong Wang, Yuki Ueda, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. Can we benchmark code review studies? a systematic mapping study of methodology, dataset, and metric. *Journal of Systems and Software*, page 111009, 2021.

[239] Haoye Wang, Xin Xia, David Lo, Qiang He, Xinyu Wang, and John Grundy. Context-aware retrieval-based deep commit message generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(4):1–30, 2021.

[240] Min Wang, Zeqi Lin, Yanzhen Zou, and Bing Xie. Cora: decomposing and describing tangled code changes for reviewer. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1050–1061. IEEE, 2019.

[241] Song Wang, Chetan Bansal, Nachiappan Nagappan, and Adithya Abraham Philip. Leveraging change intents for characterizing and identifying large-review-effort changes. In *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 46–55, 2019.

[242] Tong Wang, Dongdong Wang, Ying Zhou, and Bixin Li. Software multiple-level change detection based on two-step mpat matching. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 4–14. IEEE, 2019.

[243] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. An empirical study of usages, updates and risks of thirdparty libraries in java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–45. IEEE, 2020.

[244] Webfx. : github.com/webfx-project/webfx, 2020.

[245] Zhihua Wen and Vassilios Tzerpos. An effectiveness measure for software clustering algorithms. In *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.*, pages 194–203. IEEE, 2004.

[246] Byron J. Williams and Jeffrey C. Carver. Characterizing software architecture changes: A systematic review. *Information and Software Technology*, pages 31–51, 2010.

[247] Byron J Williams and Jeffrey C Carver. Examination of the software architecture change characterization scheme using three empirical studies. *ESE*, 19(3):419–464, 2014.

[248] Ronald J Williams and David Zipser. Gradient-based learning algorithms for recurrent. *Backpropagation: Theory, architectures, and applications*, 433:17, 1995.

[249] Manuel Wimmer, Nathalie Moreno, and Antonio Vallecillo. Viewpoint co-evolution through coarse-grained changes and coupled transformations. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 336–352. Springer, 2012.

[250] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

[251] S. Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi. Design rule hierarchies and parallelism in software development tasks. In *Proc. of ASE*, page 197, 2009.

[252] Eoin Woods. Harnessing the power of architectural design principles. *IEEE Software*, 33(4):15–17, 2016.

[253] Zhenchang Xing and Eleni Stroulia. Differencing logical uml models. *Automated Software Engineering*, 14(2):215–259, 2007.

[254] Kenji Yamauchi, Jiachen Yang, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. Clustering commits for understanding the intents of implementation. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 406–410. IEEE, 2014.

[255] Meng Yan, Ying Fu, Xiaohong Zhang, Dan Yang, Ling Xu, and Jeffrey D Kymer. Automatically classifying software changes via discriminative topic model: Supporting multi-category and cross-project. *Journal of Systems and Software*, pages 296–308, 2016.

[256] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. Automatically recommending peer reviewers in modern code review. *Transactions on SE*, pages 530–543, 2016.

[257] Su Zhang, Xinwen Zhang, Xinming Ou, Liqun Chen, Nigel Edwards, and Jing Jin. Assessing attack surface with componentbased package dependency. In *International Conference on Network and System Security*, pages 405–417. Springer, 2015.

[258] Andreas Zwinkau. Definitions of software architecture, 2019. beza1e1.tuxen.de/definitions$_software_architecture.html.$