DDARTS: A Case Study for Descriptive Design Change Artifacts Generation

AMIT KUMAR MONDAL*, Khulna University, Bangladesh
CHANCHAL K. ROY, University of Saskatchewan, Canada
SRISTY SUMANA NATH, University of Saskatchewan, Canada
BANANI ROY, University of Saskatchewan, Canada
KEVIN A. SCHNEIDER, University of Saskatchewan, Canada

,

Researchers and practitioners have linked recurring 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 [12]. Hence comes the necessity of generating descriptive design change documents and artifacts because it is the primary artifacts to track and search other artifacts.

In this study, we explore an automated technique to generate architectural design change description (textual) leveraging the most promising change detection and classification techniques that we have enhanced in this study. However, context-aware descriptive design change summary (a subset of design change artifacts) generation is a challenging task [61]. To that end, we consider precise and meaningful contexts based on the structural code change relations (SSCs), change purposes, and 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 our proposed uniform distribution models. 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). Moreover, we have defined four static rules for generating four types of change descriptions. Descriptive texts can be generated considering the ranking of distributions of the concept tokens. We also integrate commit theme information with the proposed generative model using various contextual rules. Finally, we developed a tool for the development and maintenance (DEVEM) team for design change information extraction and change log documentation. Performance evaluation with ROUGE metrics and developers reveal promising performance of our proposed system which can be easily deployed in the DEVEM process.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Additional Key Words and Phrases: datasets, neural networks, gaze detection, text tagging

Authors' addresses: Amit Kumar Mondal, trovato@corporation.com, amit@cse.ku.ac.bd, Khulna University, Gollamari Road, Khulna, Bangladesh, 9208; Chanchal K. Roy, University of Saskatchewan, 110 Science Place, Sasaktoon, SK, Canada; Sristy Sumana Nath, University of Saskatchewan, 110 Science Place, Sasaktoon, SK, Canada; Banani Roy, University of Saskatchewan, 110 Science Place, Sasaktoon, SK, Canada; Kevin A. Schneider, University of Saskatchewan, 110 Science Place, Sasaktoon, SK, Canada.

0004-5411/2018/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Amit Kumar Mondal, Chanchal K. Roy, Sristy Sumana Nath, Banani Roy, and Kevin A. Schneider. 2018. DDARTS: A Case Study for Descriptive Design Change Artifacts Generation. *J. ACM* 37, 4, Article 111 (August 2018), 32 pages. https://doi.org/XXXXXXX.XXXXXXX

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. [43, 46, 52, 57, 66, 74, 79]. 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 [12]. 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 [5]. However, proper documentation is challenging due to insufficient and inadequate content and obsolete, ambiguous information, and lack of time to write documents [5, 12]. 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 [5, 8, 12]. Moreover, little focus is given to automatically generating summarized documents of design impactful changes despite being used by the development and maintenance teams significantly [5, 8]. 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 [5, 12].

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 [5, 53, 62]. 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 [12]. Moreover, 40.5% of the major and 14.5% of the minor releases contain high-level design change information [8]. As many as 17 people (mostly the core architects of a project) can be involved in producing logs for a single release [8]. Apart from these, empirical studies with the major companies revealed that software change document has widespread impacts on project development and maintenance [5, 11]. These change documents can be of various forms. However, there is a scarcity of automated tools to support specifically the system design changes.

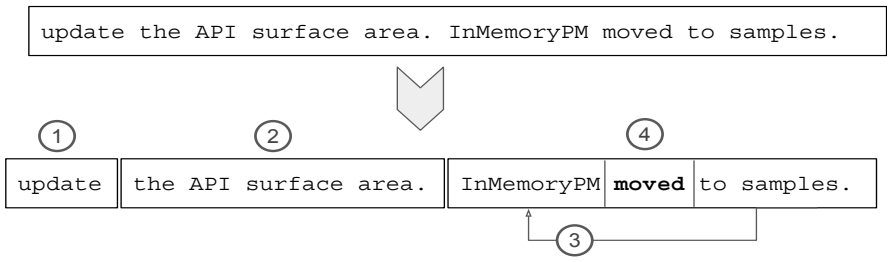

Fig. 1. Structure of a design change text.

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 [5, 12]. 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 [19, 71]. 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 [8]. As is summarized in Figure 5, 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) [8]. An example part (presented in Fig. 1) 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. 5. A few of the examples of DDARTS in a real-world project are discussed later in Fig. 2 and 3. 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 [33]) considering the time, costs and benefits [8]. 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 [5, 20]. 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 (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 [11], ChangeScribe [37] and other existing techniques generate [28, 38, 39, 73], 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 [15]. In sharp contrast, our study will seek scalable approaches

to provide the information similar to Fig. 1 and 1 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.

RQ3: *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.

RQ4: *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 [35] (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 [32, 39, 53, 73]. 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.

The rest of the study as follows. Section 3 discusses related work, Section 4 presents our dataset of study; Section 5 presents change logs analysis; Section 5.2 discusses our proposed technique;

Table 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

Section 8 presents our tool; Section 9 presents performance evaluation; Section 10 threats to our study; and Section 11 conclude this study.

2 MOTIVATION AND BACKGROUND

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 1. Meaningless or empty messages make the situation worse [39]. Hence, all the information in Table 10 is valuable for the reviewers (as well as the description of the SSCs 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 Figure 1. Descriptive design changelogs written by the Azure team are shown in Figure 2 (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 [16]. Furthermore, a detail comment for reasoning the change commit is shown in the *Detail Comment* column of Table 10. Thus, 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). Once the change purpose and causal relations in the code are extracted, many other design artifacts like these can be generated.

Scenario 3: 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 1 could be falsely processed as feature improvement using description, but originally it was for design simplification. Figure 2 shows a partial release log for the developers of AzureSDK *4.0.0-preview.4* in the log change in [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 [3], which is shown in Figure 3. Many changelogs, as we have explored ~200 instances, contain much more information (Figure 2). Writing such logs requires much more effort and analysis of various information (i.e., all commits, codebase, commit

Table 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.
```

Fig. 2. 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`.

Fig. 3. 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 [8]. Consequently, tool support for descriptive design changelogs is more critical than for commit message generation.

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 [58] – configuration change [17, 23], source-code layers (i.e., directories, package structures, and location of code files within the directories) changes [40], design model change (i.e., UML diagram) [21, 42, 76], architectural document in natural language [16, 30], and code component change operations (i.e., addition, deletion, moving, and merging components) [63, 77].

Software architecture is studied at three abstract levels: high level, intermediate level, and low level. In this paper, 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 [9]. 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 [23, 40] are also architectural changes. Our consideration of these metrics as architectural changes are based on a number of existing studies [13, 21, 34, 40, 63, 76].

Architectural Change Type: Architectural changes can be grouped on focusing various perspectives [16, 47, 77]. In this study, we consider change grouping based on the development and maintenance activities [77]. *Adaptive (A):* This change is a reflection [36, 77] 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 [48, 77] 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 [16, 69, 77] including improving processing efficiency and enhancing the performance of the software.

Design Artifacts: We refer to 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, etc. A few of the artifacts are discussed in the *Motivation* section. These artifacts are valuable for reverse-engineering as well [14].

3 RELATED WORK

Here, we discuss the most prominent and relevant studies.

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 [11] 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. [37]. A few studies also focused on the automated summary generation of commits. Jiang et al. [29] 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 [61] 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. [65] 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.

3.2 Automatic Commit Message Generation

Our study is closely related to change message generation. Following the previous studies, Jiang et al. [28] 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 [27]. Liu et al. [39] 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. [73] 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 [51] 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 [51]) 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 10, 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.

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 [8, 32, 53, 55]. Some of them generate a broader description (class-level) of the related classes/files and methods [53]. In sharp contrast, our work focuses on design impactful (module/system-level) releaselogs, which are neither too short nor too long. 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 [8].

4 DATASET COLLECTION

4.1 Dataset for Change to Text Models

We collect the module-level architectural change commits of Mondal et al. [51]. 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 4.

4.1.1 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. 3, 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

Table 3. Total number of samples in each type (nontangled)

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

of samples from each group) [64]. We consider 168 (size of the corrective class) as the minimum number of samples as the base size for the preventive and perfective.

Table 4. Selected projects and DIS commits (until 2020).

Project	**All**	**2017-2020**	**DIS**	**Domain**
Aion[6]	4718	1064	863	A multi-tier efficient blockchain network
Webfx[75]	3770	778	563	Providing a web port of JavaFx to JavaScript
Speedment[67]	4483	243	222	Creates a Java representation of the data model from SQL
AzureSDK[19]	15,180	276	244	Azure SDK for java
Atrium(Kotlin)[7]	1988	379	210	Assertion library for Kotlin with support for JVM, JS and Android
Bach[10]	2114	365	145	Java Shell Builder
Vooga[70]	1210	447	416	Game development engine
Imgui(Kotlin)[26]	1703	35	34	Game engine and 3D application framework
	Total	3587	2697	

4.1.2 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 [16, 49, 50] as well adopting the knowledge from more than 150 categorical change descriptions of the AzureSDK project (as shown in Fig. 3). 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.

4.2 Dataset for Design Change Logs Exploration

We collected release change logs and change descriptions from two open-source software. In the collecting phase, we follow some criteria to ensure the standard of the experimental projects as – (i) the projects should have well defined high-level module 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) they are in a very active phase of development and maintenance in recent times, and (v) have some releases. Thus, the Azure JavaSDK and Hibernate-search projects are one of the best choices that fulfill these criteria. 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 50 change logs, we found some of them do not contain much change modification information; rather, they provide links to change logs. Hence, for Azure JavaSDK, 39 releases have expected change logs. Then again, for Hibernate-search, we found 26 release logs (among selected 50 release logs) have considerable change modification information. Therefore, our collection contains 64 major change releases.

5 EXPLORING THE DESIGN CHANGE LOGS CONTENTS

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. Total AzureSDK change logs have 2,618 lines (excluding empty lines) of texts, and it took 24 hours (per person) to analyze. While hibernate search has 26 release change logs having a total of 1,085 lines of texts. It took around 16 hours. In the annotation process, we followed the Java coding convention to detect class, method, and module elements inside the textual descriptions. While detecting, the API/libraries are 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 Figure 4 (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 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, etc., 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 Figure 4 but not frequent. Moreover, we have observed the major release documents of 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.

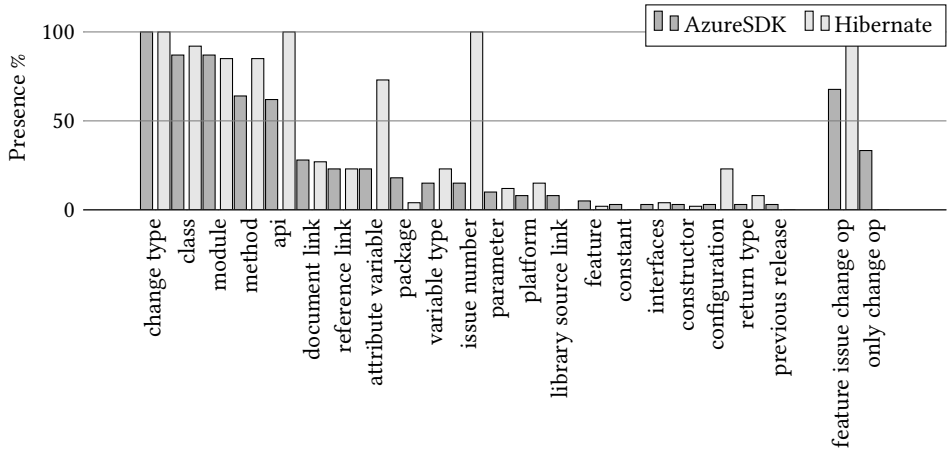

Fig. 4. Histogram of different types of information contained within a release change logs.

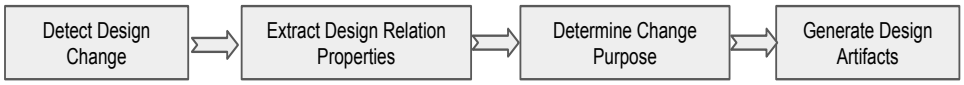

Fig. 5. Operational phases of design artifacts generation.

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.

6 CODE CHANGE TO NATURAL LANGUAGE MODEL DEVELOPMENT

6.1 Phases of Design Change Summary Generation

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

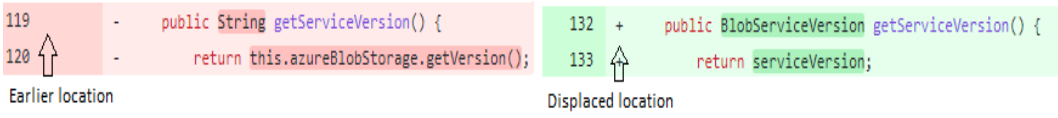

Fig. 6. Method displaced with content changes

revisions [41]. Consequently, we enhance and deploy a lightweight tool ArchSlice [51] for these purposes (presented in Section 6.2).

Change Category Determination: Next crucial step is determining the causes and types of change. For this step, we enhance and deploy the most promising classification model ArchiNet [49] that can also handle NI, empty, and tangled changes (summarized in Section 6.3).

Textual Description Generation: 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 [11, 28, 37, 39, 73]. 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 Section 6.4 and 1, we describe our proposed technique in detail.

6.2 ArchSlice Tool Extension for 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 [41], 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. [51]. 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 [51]. 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 [68] 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. 6. We found around 99% accuracy within the training set of Mondal et al. [51]. Git [24] and PyDriller [4] provide moving method as addition and deletion operation. Those methods can be moved with change or renamed (which is sometimes appear as method overloading). For example, such a moving is shown in Fig. 6 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. 6 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.

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

Table 5. 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	(+)

- 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.

6.2.1 Relation Between Change Purposes and SSCs. The presence of the 17 SSCs among the four change groups is shown in Table 5. 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

Table 6. 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. 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

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,

Table 8. 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

VCS APIs return the renaming of *EventProcessorBlobPartitionManagerSample* class in commit[1] as class add and delete, and our technique 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 5. This outcome is based on the ArchiNet [49] strategy (Eqn (1)) that produces 42% F1 with all SSCs.

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

This is an alternative to the principal component [78] and correlation-coefficient [25] 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. 6. Moreover, the $p - values$ of Pearson correlation analysis of SSCs to pairwise categories are shown in Tables 7, 8, and 9. 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

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

Table 9. 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

them would be difficult. Thus, the SSCs provide the most reliable explanation irrespective of project contexts and textual description.

6.3 Enhanced Model for Change Categorisation

We explore a combined model utilizing the concept tokens of the ArchiNet and the 17 SSCs. The training model in Eqn (1) with concept tokens and SSC is generated separately. Then the classification model 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. We also explore the combined model with concept tokens and SSC association rules. The outcome is promising, as shown in Fig. 7. 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 [60] is 80-91% (total 86%).

6.3.1 Predicting Change Types in NI, AM and archTangled Commits.

6.3.2 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. 8. The best precision rate reached to

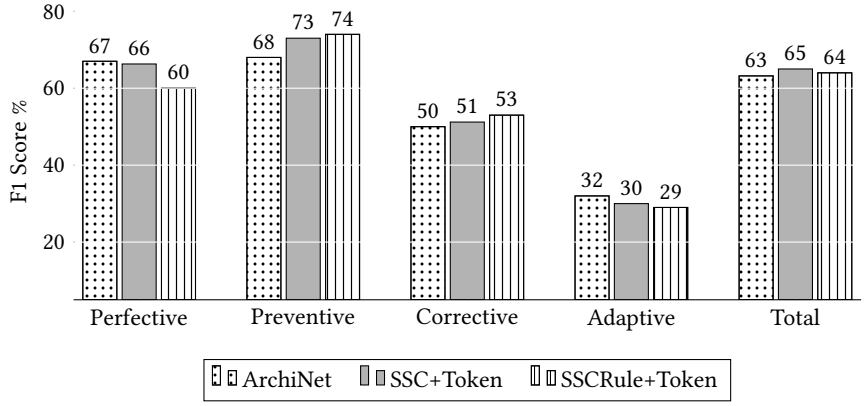

Fig. 7. F1 score of ArchiNet, Mix-model with SSC, and Mix-model1 with SSC rules.

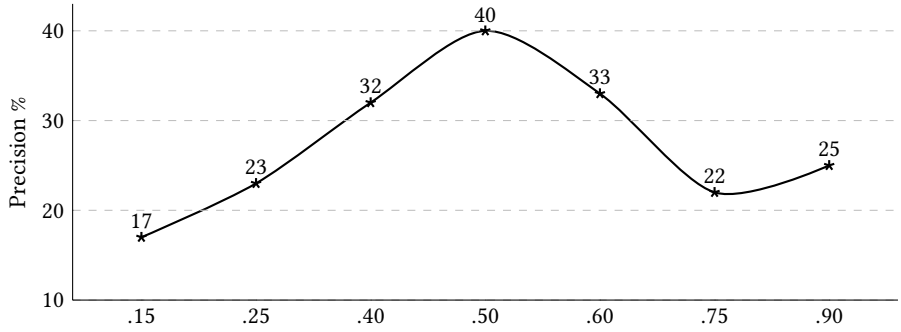

Fig. 8. Precision rate of tangled commits detection and classification for various threshold values (R).

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:

$$\frac{\omega}{W_m(C_i)} \geq R_u \tag{2}$$

If we consider the value $R_u = 0.25$ in Eqn (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 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.

Table 10. Performance for tangled commits

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

6.3.3 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. Inspired by the recent study of Ezzini et al. [18], 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 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.

6.4 Our proposed DesignChangeToText Model

We construct a baseline model for design change to texts from the balanced training set of 427 samples for generating key concept tokens from the design change operations. The total number of concept tokens are 293 and operations are 17 SSC and change categories are Perfective (P), Preventive (PV), Corrective (CR), and Adaptive (AP). It contains a collection of tokens that are present within a change category for a SSC operation and its probability distribution. This probability distribution is later utilized for choosing proper words. Thus, our proposed model is as follows:

$$TextSSC_{iC} \Leftarrow \{T \mid T \in C \wedge SSC_i \in C, T : PDF\} \tag{3}$$

Here, T is token, PDF is probability distribution function, C = $\{P, PV, CR, AP\}$, i = {1, 2,...17}. We consider term-frequency (tf) in a change category of a token as the probability distribution.

7 NATURAL LANGUAGE DESCRIPTION GENERATION

7.1 Main Algorithm

Abstract view of reasoning for DDARTS generation is displayed in Figure 13. 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 [73], but such 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. Our collected 100 releaselogs contain 86 concept tokens with a total of 383 instances meaning that the tokens are valuable when embedded in the proper contexts. The detailed process is shown in Algorithm 1. 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. 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 [49]. 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 1, 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 [11], but a bit different considering the components C_{Compos} and contextual prepositions [49] selected from the change type (with SSCs), e.g., class *A* in module *DM*1 moved ***to*** module *DM*2.

Data: $TextSSC_{iC}$ - Design change to texts model, 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} ⇐ getSSCTokenMap($TextSSC_{iC}$, C_{SSC})
    T_{Change} ⇐ predictChangeType(UModel, C_{Text}, C_{SSC})
    for Type Ti in T_{Change} **do**
      $Token[SSC]$ ⇐ getSSCToken(STM_{ti}, Ti)
      UTokens = set() ▷ unique collection
      for SSC in C_{SSC} **do**
        CTokens ⇐ getRankedToken(Range, $Token[SSC]$, SSC)
        UTokens.add(CTokens)
      end
      SD.add(generateMsg(Rules, UTokens, C_{Comps}, Ti,CM_{Theme}))
    end
  else
    Exit
  end
end

Algorithm 1: Descriptive design summary generation.

7.2 Complete Sentence Generation Rules for generateMsg()

An example structure of a descriptive change is shown in Fig. 1 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 [11] which generates one sentence for each modified method in a commit. The sentence rules (mostly consisted of *Do*

Table 11. 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

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. 1. 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 11. 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 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 11. In the next section, we will discuss the theme generation strategy for structure ②.

7.3 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 2. 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]

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

library to generate noun phrases which are not able to extract for many commit messages. For those cases, the top ranked noun is used.

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 2: Commit theme generation algorithm.

8 DDARTS TOOL

Several tools are available to extract, visualize and analyze an implemented architecture of a released version, for example, EVA [54] and SAIN [22]. 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 Figure 11. 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 DEVEM teams. The modules communicate thorough the input and produced data, as can be seen in Figure 11, 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:

8.0.1 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 Section 6.2. YAML files contain the name of involved modules, classes, and methods. A partial format is shown in Figure 9. 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 5) such as *MCNM* – Modify class that adds a new method with a new cross-module dependency.

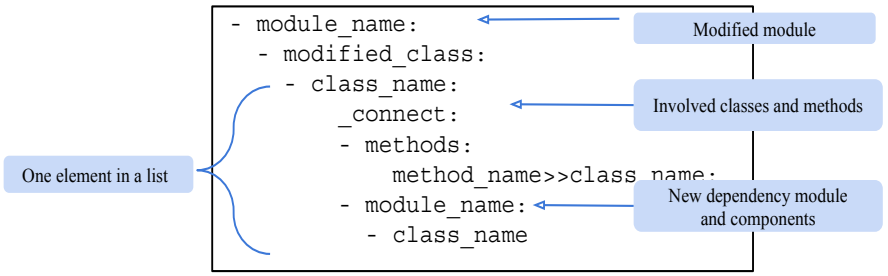

Fig. 9. A Partial Structure of SSC information saved into YAML. Where, *connect* key indicates new dependency

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)

Fig. 10. A Partial Structure of a design change commit information for reviewer.

8.0.2 TangledCategory. This module is responsible for determining change purposes using the uniform model. It takes input concept tokens (from Section 6.3) and SSC models (from Section 6.2), 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.

8.0.3 ChangeSummary. This module produces change summaries and release logs based on our proposed algorithm described in Section 5.2. It takes input as the SSC word mapping models (from Section 6.4), 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 Figure 12. 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.

8.0.4 Produced Content Structure. The basic structure of the document content follows experts opining of the written document[1] [72]. It is constructed keeping in mind following criteria –

- Convince the DEVEM 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.

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

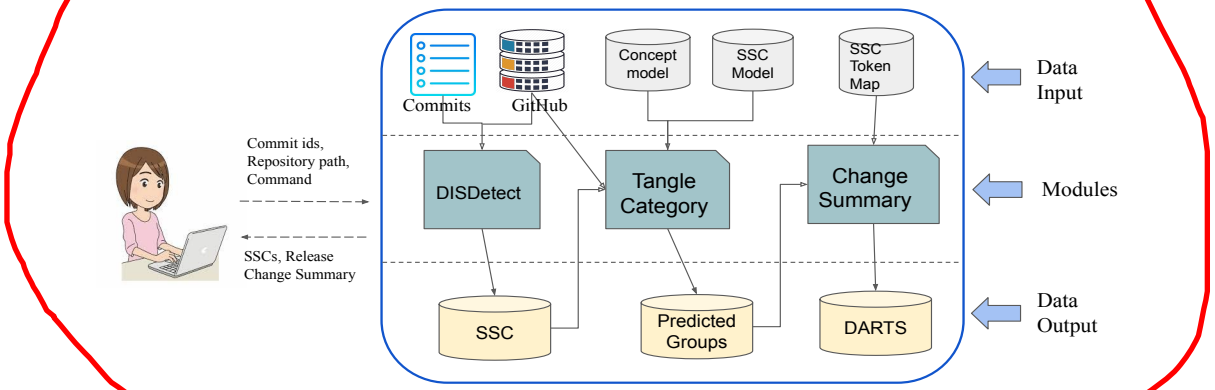

Fig. 11. High-level architecture of the DDARRTS tool.

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: ...

Fig. 12. Saved release change logs format.

9 PERFORMANCE EVALUATION

Consistent and adequate code change summary generation is one of the most challenging tasks [8, 39]. Challenges - redundancy, consistency, sentence ordering, conciseness, adequacy, while constructing summaries have made this field more difficult [56]. 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 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 10. However, more than half of the commits of Azure SDK do not have detailed

Table 12. 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	~

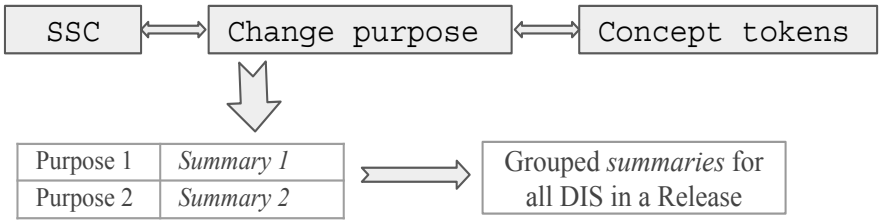

Fig. 13. Abstract reasoning of design artifacts generation.

Table 13. $ROUGE_{Precision}$ 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%

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 [39]. We have also collected the descriptions of the change logs instances of various types of changes (Figure 3) of Azure projects as of Figure 2. 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 Figure 12. 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 the next subsections, we present our performance outcomes.

9.1 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 [35] metrics (an advancement of BLEU [39]) 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 paper, our target is to produce the most relevant words in the initial phase.

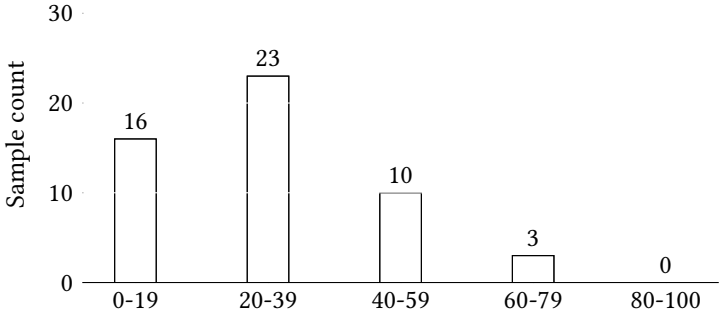

Fig. 14. Samples over the ROUGE1-P scores ranges.

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 fact, relevant or needed? Precision is measured as -

$$\frac{number\ of\ overlapping\ words\ with\ the\ reference}{total\ words\ in\ system\ summary} \tag{4}$$

The performance outcome of our algorithm is shown in Table 13.

9.1.1 Change Description for Reviewers. The highest precision and F1 measure of ROUGE [35] 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 Figure 14. 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.

9.1.2 Change Logs for Releases. Shuffling all the generated *SDs* from Algorithm 1 for all the DIS commits in a release according to similar change types (T_i) forms the release changelogs for the DEVEM team. Please note that these release logs, as shown in Figure 2 and 3 are quite different than the usual release notes generated by the existing studies [32, 53, 55]. 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 12) 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 12. 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.

9.2 Manual Cross-validation

BLEU [59] and ROGUE [35] 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 our tool with the reference summary samples subjectively. We followed the human cross-validation study by Liu et al. [38] 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.

9.2.1 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 Figure 4. 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 14. These questionnaires are formed considering various crucial perspectives of the tool. Some of the perspectives are inspired from[2] [31, 72].

9.2.2 Evaluation Results Analysis. The average evaluation scores and their subjective indication of all EQs are shown in Table 15. 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 [45] in Figure 15. 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:

$$AVG_{sig} = \frac{\sum_{n=1}^{7}(M_{EQn})}{N} * \frac{100}{9} \tag{5}$$

[1] search.maven.org/remotecontent?filepath=com/azure/resourcemanager/azure-resourcemanager-batch/1.0.0/azure-resourcemanager-batch-1.0.0-changelog.md

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

Table 14. 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

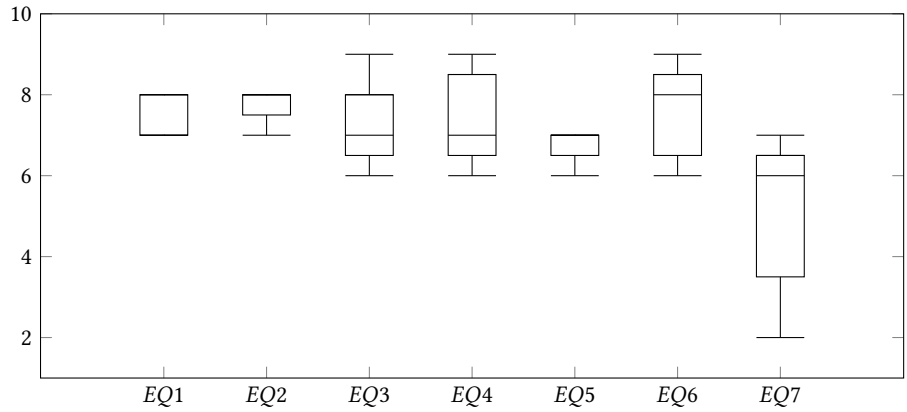

Fig. 15. Range of evaluated scales of EQs

Table 15. Cross-validation evaluation average scores and their indication.

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

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

Table 16. Execution time for all phases of DDARTS, 1KLOC = 1000 Lines of code.

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

values are below 6 for 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.

9.3 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 [5]. Another crucial point is the cost of generating such a document, meaning it can be deployed with low-cost infrastructure frequently [5]. 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 Figure 11. 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 [54, 63]. 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 Figure 16. 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.

10 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. [51]. 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 [44], MojoFM [76], ACDC, ARC [63], 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.

11 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 [73] embedding the thematic knowledge from the commit messages with the SSCs for generating more accurate sentences.

REFERENCES

[1] 2021. azure-sdk-for-java/commit/e54ecd867cbf09dc722491ffe6f138002321df2f.

[2] 2021. azure.github.io/azure-sdk/releases/2019-10-11/java.html.

[3] 2021. azure-sdk-for-java/blob/main/sdk/keyvault/azure-security-keyvault-certificates/CHANGELOG.md.

[4] 2021. PyDriller:github.com/ishepard/pydriller.

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

[6] Aoin. 2020. :github.com/aionnetwork/aion.

[7] Atrium. 2020. : github.com/robstoll/atrium.

[8] T. Bi, X. Xia, D. Lo, J. Grundy, and T. Zimmermann. 2020. An Empirical Study of Release Note Production and Usage in Practice. *IEEE Transactions on Software Engineering* (2020), 1–1. https://doi.org/10.1109/TSE.2020.3038881

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

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

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

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

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

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

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

[16] Wei Ding, Peng Liang, Antony Tang, and Hans Van Vliet. 2015. Causes of Architecture Changes: An Empirical Study through the Communication in OSS Mailing Lists.. In *SEKE*. 403–408.

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

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

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

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

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

[22] Joshua Garcia, Mehdi Mirakhorli, Lu Xiao, Yutong Zhao, Ibrahim Mujhid, Khoi Pham, Ahmet Okutan, Sam Malek, Rick Kazman, Yuanfang Cai, et al. 2021. Constructing a Shared Infrastructure for Software Architecture Analysis and Maintenance. In *2021 IEEE 18th International Conference on Software Architecture (ICSA)*. IEEE, 150–161.

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

[24] GitPython. 2020. gitpython.readthedocs.io/en/stable.

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

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

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

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

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

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

[31] Mark Kishlansky et al. 1991. How to Read a Document. *Sources of the West: Readings in Western Civilization* (1991).

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

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

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

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

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

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

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

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

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

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

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

[43] Pratyusa K Manadhata and Jeannette M Wing. 2011. An Attack Surface Metric. *IEEE Transactions on Software Engineering* 37, 03 (2011), 371–386.

[44] Spiros Mancoridis, Brian S Mitchell, Yihfarn Chen, and Emden R Gansner. 1999. 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)*. IEEE, 50–59.

[45] Robert McGill, John W Tukey, and Wayne A Larsen. 1978. Variations of box plots. *The american statistician* 32, 1 (1978), 12–16.
[46] Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, and Qiong Feng. 2019. Architecture Anti-patterns: Automatically Detectable Violations of Design Principles. *IEEE Transactions on Software Engineering* (2019).
[47] Audris Mockus and Lawrence G. Votta. 2000. Identifying Reasons for Software Changes Using Historic Databases. In *Proceedings of the International Conference on Software Maintenance*. 120–130.
[48] Parastoo Mohagheghi and Reidar Conradi. 2004. 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.* IEEE, 7–16.
[49] Amit Kumar Mondal, Banani Roy, Sristy Sumana Nath, and Kevin A Schneider. 2021. A Concept-token based Approach for Determining Architectural Change Categories. In *Proceedings of the 33rd International Conference on Software Engineering & Knowledge Engineering*. 7–14.
[50] Amit Kumar Mondal, Banani Roy, and Kevin A Schneider. 2019. 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)*. IEEE, 62–73.
[51] Amit Kumar Mondal, Chanchal K Roy, Kevin A Schneider, Banani Roy, and Sristy Sumana Nath. 2021. 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)*. 1–6.
[52] David Monschein, Manar Mazkatli, Robert Heinrich, and Anne Koziolek. 2021. Enabling Consistency between Software Artefacts for Software Adaption and Evolution. In *2021 IEEE 18th International Conference on Software Architecture (ICSA)*. IEEE, 1–12.
[53] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. 2017. ARENA: An Approach for the Automated Generation of Release Notes. *IEEE Transactions on Software Engineering* 43 (2017), 106–127.
[54] Daye Nam, Youn Kyu Lee, and Nenad Medvidovic. 2018. Eva: A tool for visualizing software architectural evolution. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 53–56.
[55] Sristy Sumana Nath and Banani Roy. 2021. Towards Automatically Generating Release Notes using Extractive Summarization Technique. In *International Conference on Software Engineering & Knowledge Engineering, SEKE 2021. Proceedings*. 241–248. https://doi.org/10.18293/SEKE2021-119
[56] Najam Nazar, Yan Hu, and He Jiang. 2016. Summarizing Software Artifacts: A Literature Review. *JCST* (2016), 883–909.
[57] Peyman Oreizy, Nenad Medvidovic, and Richard N Taylor. 1998. Architecture-based runtime software evolution. In *Proceedings of the 20th international conference on Software engineering*. IEEE, 177–186.
[58] Ipek Ozkaya, Peter Wallin, and Jakob Axelsson. 2010. Architecture knowledge management during system evolution: observations from practitioners. In *Proceedings of the 2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge*. 52–59.
[59] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.
[60] Mohammad Masudur Rahman and Chanchal K Roy. 2018. 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*. ACM, 621–632.
[61] Sarah Rastkar and Gail C Murphy. 2013. Why did this code change?. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 1193–1196.
[62] 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. 2017. On-demand developer documentation. In *2017 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 479–483.
[63] Marcelo Schmitt Laser, Nenad Medvidovic, Duc Minh Le, and Joshua Garcia. 2020. 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*. 1546–1550.
[64] Gaganpreet Sharma. 2017. Pros and cons of different sampling techniques. *International journal of applied research* 3, 7 (2017), 749–752.
[65] Jinfeng Shen, Xiaobing Sun, Bin Li, Hui Yang, and Jiajun Hu. 2016. On automatic summarization of what and why information in source code changes. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 103–112.
[66] Marcelino Campos Oliveira Silva, Marco Tulio Valente, and Ricardo Terra. 2016. 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*. 248–254.
[67] Speedment. 2020. : github.com/speedment/speedment.

[68] Jeffrey Svajlenko and Chanchal K Roy. 2017. Cloneworks: A fast and flexible large-scale near-miss clone detection tool. In *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 177–179.
[69] E Burton Swanson. 1976. The dimensions of maintenance. In *Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press, 492–497.
[70] Vooga. 2020. : github.com/anna-dwish/vooga.
[71] Vooga. 2020. Hibernate Search: github.com/hibernate/hibernate-search.
[72] Rob Waller. 2011. What makes a good document. *The criteria we use. Technical paper* 2 (2011).
[73] Haoye Wang, Xin Xia, David Lo, Qiang He, Xinyu Wang, and John Grundy. 2021. Context-aware retrieval-based deep commit message Generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 4 (2021), 1–30.
[74] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. 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)*. IEEE, 35–45.
[75] Webfx. 2020. : github.com/webfx-project/webfx.
[76] Zhihua Wen and Vassilios Tzerpos. 2004. An effectiveness measure for software clustering algorithms. In *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.* IEEE, 194–203.
[77] Byron J. Williams and Jeffrey C. Carver. 2010. Characterizing Software Architecture Changes: A Systematic Review. *Information and Software Technology* (2010), 31–51.
[78] Svante Wold, Kim Esbensen, and Paul Geladi. 1987. Principal component analysis. *Chemometrics and intelligent laboratory systems* 2, 1-3 (1987), 37–52.
[79] Su Zhang, Xinwen Zhang, Xinming Ou, Liqun Chen, Nigel Edwards, and Jing Jin. 2015. Assessing attack surface with componentbased package dependency. In *International Conference on Network and System Security*. Springer, 405–417.