Noname manuscript No.
(will be inserted by the editor)

Automatically Estimating the Effort Required to Repay Self-Admitted Technical Debt

Yikun Li · Mohamed Soliman · Paris Avgeriou

Received: date / Accepted: date

Abstract Technical debt refers to the consequences of sub-optimal decisions made during software development that prioritize short-term benefits over long-term maintainability. Self-Admitted Technical Debt (SATD) is a specific form of technical debt, explicitly documented by developers within software artifacts such as source code comments and commit messages. As SATD can hinder software development and maintenance, it is crucial to address and prioritize it effectively. However, current methodologies lack the ability to automatically estimate the repayment effort of SATD based on its textual descriptions. To address this limitation, we propose a novel approach for automatically estimating SATD repayment effort, utilizing a comprehensive dataset comprising 341,740 SATD items from 2,568,728 commits across 1,060 Apache repositories. Our findings show that different types of SATD require varying levels of repayment effort, with code/design, requirement, and test debt demanding greater effort compared to non-SATD items, while documentation debt requires less. We introduce and evaluate machine learning methodologies, particularly BERT and TextCNN, which outperforms classic machine learning methods and the naive baseline in estimating repayment effort. Additionally, we summarize keywords associated with varying levels of repayment effort that

Yikun Li
Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence, University of Groningen, The Netherlands
E-mail: yikun.li@rug.nl

Mohamed Soliman
Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence, University of Groningen, The Netherlands
E-mail: m.a.m.soliman@rug.nl

Paris Avgeriou
Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence, University of Groningen, The Netherlands
E-mail: p.avgeriou@rug.nl

occur during SATD repayment. Our contributions aim to enhance the prioritization of SATD repayment effort and resource allocation efficiency, ultimately benefiting software development and maintainability.

1 Introduction

Technical debt is a metaphor used to describe the consequences of sub-optimal decisions made during software development that prioritize short-term benefits over long-term software maintainability and evolvability (Avgeriou et al., 2016). As technical debt accumulates, it can negatively impact the development process, hindering the ability to make changes to the software, such as fixing bugs or implementing new features. *Self-Admitted Technical Debt* (SATD) (Potdar and Shihab, 2014) is a form of technical debt that is explicitly documented by developers within different software artifacts, such as source code comments, commit messages, issue tracking systems, and pull requests (Zampetti et al., 2021). For instance, a developer may leave a comment in the code suggesting the removal of unnecessary code in the future: *"TODO: we need to remove the dead code"*. In another example, a developer may document the low performance of a method: *"This method is inefficient and could be refactored for better performance"*.

In recent years, there has been a growing interest in SATD identification, with numerous studies focusing on identifying SATD from various sources (Ren et al., 2019; Li et al., 2022a, 2023a). However, effective SATD management requires not only identification but also estimation of the effort needed for its repayment. This estimation is essential for prioritizing the repayment of technical debt items and allocating resources efficiently; sometimes developers are faced with hundreds or even thousands of technical debt items, and prioritizing them becomes a major challenge. While the measurement of technical debt repayment (commonly known as technical debt principal) has been extensively studied and even integrated into industrial tools like SonarQube, this has not been accomplished for SATD. Estimating the repayment effort of SATD involves natural language documentation by developers, which is fundamentally different from detecting technical debt through methods such as static code analysis. SATD is essential in addition to technical debt detection through static code analysis because they complement each other (Li et al., 2020). Previous research emphasizes the automatic estimation of SATD repayment effort as a critical feature desired by software engineers and managers (Li et al., 2022b). However, no existing approaches address this issue, creating a knowledge gap that hinders the prioritization of SATD repayment and efficient resource allocation in software development processes.

To address this challenge, we first explore the effort required to repay different types of SATD and non-SATD items and subsequently evaluate various machine learning approaches for automatically estimating the repayment effort required for SATD using textual information. To achieve this goal, we gathered all accessible commits, along with their corresponding commit messages and code changes, from 1,060 Apache repositories. We then identified SATD items by analyzing the commit messages and assessed the repayment effort for each SATD item based on the related code changes. Using this dataset for training, we assessed the performance of various machine learning models and contrasted the outcomes with a naive baseline for automatically estimating the effort required to pay back SATD. Finally, we summarized the keywords that correlate with varying levels of repayment effort, aiming to uncover patterns that could assist in understanding the factors influencing the complexity of repaying SATD.

We decided to use commit messages for SATD identification, instead of the other aforementioned sources (e.g. source code comments, issues), for three main reasons: 1) commit messages usually document resolved SATD, while other sources also discuss SATD that is not resolved yet; 2) commits consist of code changes and are accompanied by commit messages that provide context and purpose for those changes, unlike other sources where code changes and comments are not directly connected; 3) a sufficient quantity of commit messages is available for analysis.

The primary contributions of this paper encompass the following:

– **Assembling an extensive dataset on SATD repayment effort.** We gathered a comprehensive dataset consisting of 341,740 SATD items from 2,568,728 commits derived from 1,060 Apache repositories. For each SATD item, this dataset includes lines of code added and deleted, the number of files added, modified, and deleted, as well as varying significance levels of code changes (Fluri and Gall, 2006; Fluri et al., 2007). To foster further research in this domain, we make our dataset publicly available[1].
– **Presenting the difference in repayment effort between diverse SATD and non-SATD items.** Our findings reveal that code/design debt, requirement debt, and test debt necessitate greater repayment effort compared to non-SATD items, whereas documentation debt demands less repayment effort.
– **Introducing and evaluating approaches for automatically estimating SATD repayment effort.** Our results demonstrate that SATD effort can be accurately predicted. Particularly, deep learning methods, such as BERT and TextCNN, outperform classic machine learning techniques and the naive baseline by a considerable margin in estimating repayment effort.
– **Summarizing keywords associated with disparate levels of repayment effort.** We analyze and summarize the keywords correlated with varying levels of repayment effort that arise during the process of repaying SATD. This contribution helps in understanding the factors influencing the complexity of SATD repayment.

The organization of this paper is as follows. In Section 2, we discuss related work. The case study design is elaborated in Section 3. The results are presented and discussed in Section 4 and Section 5, respectively. In Section 6, we evaluate the threats to validity. Finally, we draw conclusions in Section 7.

[1] `https://github.com/yikun-li/satd-repayment-effort`

2 Related Work

This study aims to investigate the effort required to repay SATD. Accordingly, we categorize the relevant literature into two main sections: A) prior research on SATD in general, and B) earlier studies related to the analysis of repayment effort for technical debt.

2.1 Self-Admitted Technical Debt in General

The initial investigation into SATD in source code comments was conducted by Potdar and Shihab (2014), who examined four open-source projects and discovered that SATD comments were present in 2.4% to 31% of source files, and only 26.3% to 63.5% of the identified SATD comments were removed after being introduced. Maldonado and Shihab (2015) expanded on this work by classifying SATD into five categories (design, requirement, defect, documentation, and test debt) based on the analysis of 33,000 code comments from open-source five projects. Their results showed that design debt was the most frequent form of SATD, accounting for 42% to 84% of classified cases.

Subsequent to the exploration of SATD, researchers have shown considerable interest in devising automated methods for SATD detection. Several machine learning approaches (Ren et al., 2019; Li et al., 2022a, 2023a; Guo et al., 2021) have been employed to detect different types of SATD items from various sources. Ren et al. (2019) developed a Convolutional Neural Network-based approach to enhance the accuracy and explainability of SATD detection, particularly for cross-project prediction. Li et al. (2022a) generated a dataset of 4,200 issues from seven open-source projects and proposed a machine learning approach to detect SATD in issue tracking systems, outperforming baseline methods, benefiting from knowledge transfer, and extracting intuitive SATD keywords. Li et al. (2023a) also introduced an automated SATD identification approach from multiple sources, such as source code comments, commit messages, pull requests, and issue tracking systems, leveraging a multitask learning technique. Guo et al. (2019) suggested a simple heuristic approach for SATD identification, demonstrating that it performs similarly or better than existing methods with a high overlap in correct identification, and recommended that future SATD identification studies utilize this as an easy-to-implement baseline.

Additionally, researchers explored the repayment of SATD. Maldonado et al. (2017) evaluated the repayment of SATD in five open-source projects and observed that the majority of SATD is eventually eliminated, mostly by those who introduced it. They found that it takes 18 to 172 days to remove SATD comments on average. Zampetti et al. (2018) also studied the removal of SATD in five Java open-source projects and discovered that 20% to 50% of SATD is unintentionally removed, and only 8% of debt removal is documented in commit messages.

2.2 Repayment Effort for Technical Debt

Numerous studies have investigated the effort of paying back technical debt. Xiao et al. (2016) introduced a novel approach for the automatic detection, quantification, and modeling of architectural debt in software systems. The authors quantified effort by measuring the number of lines of code modified and committed to fix bugs. Their evaluation carried out on seven large-scale open-source projects revealed that their approach effectively uncovers how architectural issues evolve into technical debt over time. Martini et al. (2018) conducted a case study within a large company, establishing a comprehensive framework for the semi-automated identification and estimation of architectural debt. In their effort estimation, they considered factors such as the number of files, lines of code, changes in all files, and McCabe's and Halstead's complexity metrics. Nugroho et al. (2011) proposed an approach to quantify debt in terms of fixing technical quality issues and the extra cost spent on maintenance. Specifically, they estimated the repayment effort by calculating the rework fraction and rebuild value: the rework fraction represents the percentage of lines of code requiring modification to enhance software quality; the rebuild value estimates the effort (in man-months) necessary to rebuild a system using a particular technology.

Furthermore, two studies have empirically investigated the repayment effort of SATD. Mensah et al. (2018) analyzed SATD items to identify instances of "vital few" (bug-prone) tasks and "trivial many" (less bug-prone) tasks. They used the number of commented lines of code as a measure of effort estimation for SATD. The results indicated that highly prioritized (vital few) SATD tasks required a rework effort of modifying 10 to 25 commented LOC per source file. Wehaibi et al. (2016) explored whether SATD changes require more effort to be repaid than non-SATD changes. They identified SATD and non-SATD changes using 62 SATD keywords (Potdar and Shihab, 2014) from five open-source projects, namely Chromium, Hadoop, Spark, Cassandra, and Tomcat. They then compared the difficulty of SATD and non-SATD changes using four measures: the total number of modified lines, the number of modified files, the number of modified directories, and change entropy. Their results suggested that SATD changes were more challenging than non-SATD changes across all four measures of difficulty.

Our study **differs** from the aforementioned studies on SATD repayment effort in several aspects. First, we employ state-of-the-art machine learning techniques to identify SATD items, as opposed to relying on the list of 62 SATD keywords from earlier work (Potdar and Shihab, 2014). Second, we extract SATD-related changes corresponding to commits identified as SATD items. Specifically, after employing machine learning techniques to pinpoint commits with SATD items, we further analyze the related code changes to measure the repayment effort involved. Third, we examine four distinct types of SATD items, rather than treating them as a single, homogeneous group, allowing for more nuanced insights into different types of SATD. Fourth, we integrate a broader range of metrics to offer a more comprehensive assessment of repayment effort, enhancing our understanding of the SATD repayment. Fifth, our analysis covers over 1,000 repositories, representing a significant expansion compared to the five repositories investigated in previous studies. Finally, and most notably, we introduce the first-ever approach for predicting SATD repayment effort based on SATD textual information, as well as identifying keywords associated with varying levels of repayment effort. This innovative approach offers valuable insights and guidance for developers managing SATD.

3 Study Design

The goal of this study, formulated according to the Goal-Question-Metric (van Solingen et al., 2002) template is to "***analyze*** *source code and commit messages* ***for the purpose of*** *investigating the effort required to repay different types of SATD and non-SATD items, automatically estimating this repayment effort, and identifying keywords associated with varying levels of repayment effort* ***from the point of view of*** *software engineers* ***in the context of*** *open-source software.*" This goal is refined into four research questions (RQs):

– **RQ1:** *Are there differences in repayment effort between SATD and non-SATD items?*
  Rationale: Previous study (Li et al., 2022b) has found that it generally takes longer to resolve SATD items compared to non-SATD items; however, it is unclear whether this is due to the more complex changes required for SATD items or their lower priority. Understanding the differences in SATD repayment compared to fixing non-SATD is crucial for comprehending why it takes longer to repay SATD items and how best to address them. For example, if we discover that SATD items require significantly more complex changes than non-SATD items, this can increase awareness among development teams that addressing SATD items may demand more resources and effort than initially anticipated. As a result, organizations may opt to allocate a larger budget or increase the number of developers tasked with addressing SATD items, thereby promoting more efficient debt repayment. Conversely, if SATD repayment is deemed too costly for rapid delivery, organizations may choose to prioritize other tasks instead.
– **RQ2:** *Are there differences in repayment effort among different types of SATD items?*
  Rationale: SATD can be categorized into several types, such as code debt, design debt, test debt, documentation debt, requirement debt, and architecture debt (Alves et al., 2014). Addressing different types of SATD may entail varying degrees of complexity and challenges for software developers. For instance, resolving requirement debt could be more complex than dealing with code debt. By investigating this research question, we aim to understand the differences in repayment effort among various types of SATD items. This insight can assist developers in prioritizing specific types of SATD items throughout the software development process. For example,

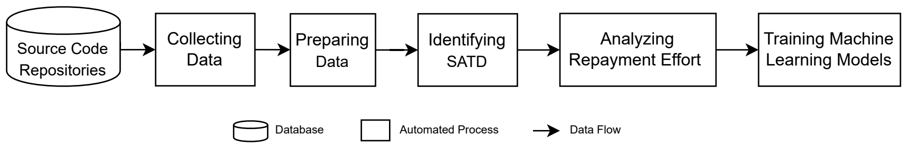

Fig. 1 The framework of our study.

if we determine that addressing code debt is less demanding than tackling design debt, developers can prioritize code debt resolution, recognizing that it will have a comparatively smaller impact on the development process.

– **RQ3:** *Can we accurately predict the effort required for SATD repayment based on the SATD text?*
  Rationale: Effectively prioritizing SATD items based on their repayment effort can help developers allocate resources effectively. Nevertheless, estimating the effort required for SATD repayment based on textual descriptions remains a challenging task, and no approaches for this task currently exist. To address this challenge, we explore the potential of machine learning techniques to predict the effort of the required code changes, based on the textual content of SATD items. By investigating the effectiveness of different approaches in this context, we aim to offer insights into the feasibility of employing automated techniques for estimating SATD repayment effort. This would enable practitioners to utilize these tools for better SATD management and decision-making regarding which SATD items to tackle first.
– **RQ4:** *What keywords are associated with varying levels of repayment effort when repaying SATD?*
  Rationale: Gaining an understanding of the specific keywords linked to different levels of effort for repaying SATD items can offer valuable insights into the complexity and difficulty associated with addressing SATD items. Identifying such keywords might enable the creation of targeted guidelines or best practices for addressing specific types of SATD items, which could lead to more efficient prioritization and resource allocation in the software development process. For example, such a guideline based on identified keywords might recommend addressing low-effort SATD items during regular development sprints or when developers have some spare time, as the presence of specific keywords may indicate low repayment effort. Additionally, best practices could be developed to encourage developers to use clear and accurate keywords when reporting SATD items, ensuring that the true complexity and effort associated with addressing the technical debt are well-understood and considered during the prioritization process.

To address our research questions, we use the approach outlined in Fig. 1. First, we *collect data* in terms of commit messages and their corresponding code changes from repositories. An example[2] of a commit message along with

[2] `https://github.com/apache/ant/commit/ecf83d2`

its associated code changes is presented in Table 1. Next, we *prepare data* to ensure consistency and eliminate irrelevant information. Subsequently, we *identify SATD* items using the commit messages and *analyze the repayment effort* based on the associated code changes. Finally, we *train machine learning models* to answer our research questions. The subsequent subsections provide detailed information about each of these steps.

Table 1 Example of commit messages and their corresponding code changes

Commit message: **implement TODO of configurable buffer size.**

```
@@ -131,6 +133,27 @@ public synchronized void waitFor()
+ /**
+ * Set the size in bytes of the read buffer.
+ * @param bufferSize the buffer size to use.
+ * @throws IllegalStateException if the StreamPumper is
+ * already running.
+ */
+ public synchronized void setBufferSize(int bufferSize) {
+   if (started) {
+       throw new IllegalStateException(
+           "Cannot set buffer size on a running StreamPumper");
+   }
+   this.bufferSize = bufferSize;
+ }
```

3.1 Collecting Data

To address our research questions, we focused on analyzing Apache Java projects, owing to their high quality, widespread usage, mature community support, and public availability for access and use. To gather the necessary data, we obtained all available Apache Java repositories[3] on Feb 1, 2023. Our search resulted in a total of 1,060 Apache Java repositories, all of which were included in our study. The collection process was carried out using an automated script to ensure consistency and accuracy in the selection of repositories. The script, along with the collected dataset, is made available in the replication package[1] to facilitate future research and validation of our findings. The collected data includes all available commits and their associated commit messages and code changes from each of the 1,060 Apache Java repositories.

[3] `https://github.com/orgs/apache/repositories`

3.2 Preparing Data

The data collected from the 1,060 Apache repositories resulted in 2,568,728 commits. First, we removed merge and rollback commits from the dataset, consistently with previous similar studies (Jiang et al., 2017; Liu et al., 2019), as these commits do not provide new information and tend to be substantially larger than other types of commits. Merge commits represent a point where two branches of code are combined, while rollback commits are used to undo changes made to a codebase. Next, we eliminated all commits containing non-English characters, resulting in a dataset of 2,382,877 commits for subsequent analysis.

3.3 Identifying SATD

To identify SATD items in commit messages, we employed a machine learning model that was trained in our previous work (Li et al., 2023a). Their study proposed a CNN and multitask learning approach to identify four types of SATD, namely code/design debt, requirement debt, documentation debt, and test debt, from four software artifacts with an average F1-score of 0.611. As our study aims to identify SATD from commit messages, which is a software artifact supported by the approach proposed by Li *et al.*, we use their approach to identify the different types of SATD items from commit messages.

3.4 Analyzing Repayment Effort

We then analyze the code changes linked to the commit messages which indicate SATD to calculate the repayment effort of the SATD items. In this study, we assess the repayment effort of SATD from two distinct perspectives: 1) the effort required to address SATD items per se, and 2) the effort involved in handling ripple effects of the SATD items on the rest of the system. These two perspectives are essential to understanding the full implications of repaying technical debt, as they capture both the direct and indirect consequences of addressing SATD items. The first perspective has been the primary focus of existing literature on SATD repayment (Mensah et al., 2018; Wehaibi et al., 2016). The second perspective is equally important, as addressing technical debt often results in unforeseen consequences on other parts of the software (Brown et al., 2010). To capture these two aspects, we employ a comprehensive set of nine different metrics, as detailed below:

1. **Effort Required to Address SATD Items**: Inspired by the work of Wehaibi *et al.* (Wehaibi et al., 2016), we employ five metrics to provide a quantitative assessment of the effort required to address SATD items. These metrics include: 1) lines of code added (LA): the number of new lines of code added during changes; 2) lines of code deleted (LD): the number of lines of code removed during changes; 3) files added (FA): the number

of new files created and added during changes; 4) files modified (FM): the number of files changed; 5) files deleted (FD): the number of files removed during changes. For instance, a high number of lines of code added typically indicates a more complex or substantial change to the software system. We employ JGit[4] to analyze repositories and calculate these five metrics.

2. **Effort Involved in Handling Ripple Effects**: Drawing inspiration from the work on the significance level of code changes by Fluri *et al.* (Fluri and Gall, 2006; Fluri et al., 2007), we utilize four metrics to provide a qualitative assessment of the effort involved in managing ripple effects within the system. The significance level expresses how strongly a change may impact other source code entities (Fluri et al., 2007). To measure this, we adopt the significance level classification tool[5] provided by Fluri *et al.* (Fluri and Gall, 2006; Fluri et al., 2007), which is based on tree edit operations performed on the abstract syntax tree. These metrics include: 1) low significance level code changes (LCC): the number of changes with a low impact on the functionality or performance of the software system; 2) medium significance level code changes (MCC): the number of changes with a moderate impact on the functionality or performance of the software system; 3) high significance level code changes (HCC): the number of changes with a significant impact on the functionality or performance of the software system; 4) crucial significance level code changes (CCC): the number of changes that are critical or essential to the functionality or performance of the software system. For instance, code changes in a method body are generally assigned LCC or MCC, while alterations to a class interface are typically designated as HCC or CCC.

3.5 Training Machine Learning Models

3.5.1 Machine Learning Models and Baseline:

To predict the effort required for SATD repayment based on the SATD text (RQ3), we select appropriate machine learning models. Although no existing approach specifically targets predicting the repayment effort of SATD, TextCNN and BERT have demonstrated effectiveness in feature extraction for SATD studies (Ren et al., 2019; Li et al., 2023a,b). Therefore, we choose TextCNN and BERT as our primary machine learning models for predicting the repayment effort of SATD. We evaluate their performance by comparing them to several classic machine learning approaches and one naive baseline. The methods used in this study are detailed below:

– **TextCNN**: TextCNN (Text Convolutional Neural Network) is an effective text classification algorithm proposed by Kim (Kim, 2014). By employing convolutional layers, TextCNN captures local features and patterns within

[4] `https://www.eclipse.org/jgit`
[5] `https://bitbucket.org/sealuzh/tools-changedistiller`

text data. Its ability to efficiently learn discriminative features from text has led to its adoption in numerous SATD identification studies (Ren et al., 2019; Li et al., 2022a, 2023a). In this study, we use optimal hyperparameters identified in previous work (Li et al., 2023a) to maximize model performance. Specifically, we set the word-embedding dimension at 300, apply filters with five different window sizes (1,2,3,4,5), and use 200 filters for each window size.
– **BERT**: BERT (Bidirectional Encoder Representations from Transformers) is a pre-trained transformer-based language model developed by Devlin *et al.* (Devlin et al., 2018) that has shown state-of-the-art performance in various natural language processing tasks, including text classification. Given its success in capturing contextual information from text and its proven effectiveness in handling complex language understanding tasks, BERT is well-suited for predicting the repayment effort of SATD based on textual information. In this study, we fine-tune BERT base models to leverage their deep understanding of language and context to accurately predict the repayment effort based on the SATD textual information.
– **Classic Machine Learning Approaches (LR, RF, SVR)**: Since predicting repayment effort is a regression task, we compare deep learning approaches with Linear Regression (LR) (Chatterjee and Hadi, 2006), Random Forest (RF) (Ho, 1995), and Support Vector Regression (SVR) (Drucker et al., 1996). The input data is vectorized using TF-IDF, and the models are trained using Sklearn implementation with default settings.
– **Naive Baseline**: To provide a benchmark for the performance of our models, we include a naive baseline. This baseline produces results by generating predictions according to the mean and standard deviation of the training data set distribution.

We implement machine learning models using the PyTorch library and train them on NVIDIA Tesla A100 GPUs. For evaluation, we split the dataset into training, validation, and test sets, adhering to a standard 80/10/10 split.

3.5.2 Evaluation Metrics

In this study, we measure the performance of our machine learning models using the Root Mean Square Error (RMSE) metric. RMSE is a widely adopted metric in regression analysis, employed to assess the accuracy of predicted values in comparison to the actual values. The RMSE is calculated by taking the square root of the average of the squared differences between the predicted and actual values: $\sqrt{\frac{1}{n}\sum_{i=1}^{n}(actual_i - predicted_i)^2}$, where n is the number of data points, $actual_i$ is the actual value of the ith data point, and $predicted_i$ is the predicted value of the ith data point. The lower the RMSE value, the better the performance of the model, indicating a smaller gap between the predicted and ground truth values.

3.5.3 Keyword Extraction

To address RQ4, we employ a keyword extraction approach to identify keywords associated with varying levels of repayment effort. This approach is based on the work of Ren *et al.* (Ren et al., 2019), which uses the backtracking technique on TextCNN to extract n-gram keywords. First, we feed the text data to the CNN model, and the most important features are selected based on their weights. Next, the corresponding filters are located by backtracking the selected features. Finally, we locate the n-gram keywords in the input text based on the filter position information. Specifically, we use the CNN models trained in RQ3 to extract and summarize unigram to five-gram SATD keywords.

4 Results

4.1 RQ1: Are there differences in repayment effort between SATD and non-SATD items?

To evaluate the effort required for direct resolution of SATD, we present the mean values of lines added (LA), lines deleted (LD), and total lines changed in Table 2. The highest values are highlighted in bold, and the lowest values are underlined for easier comparison. While the total lines changed for SATD items are slightly greater than non-SATD items, a noticeable difference exists in the lines added (LA) and lines deleted (LD) for SATD and non-SATD items. Specifically, SATD changes have lower lines added (LA) and higher lines deleted (LD) compared to non-SATD changes.

Table 2 The average number of lines added (LA) and deleted (LD).

Type	LA	LD	Total
SATD	<u>70.4</u>	**64.6**	**135.0**
Non-SATD	**88.0**	<u>41.1</u>	<u>129.1</u>

For assessing the effort in direct resolution of SATD in terms of file changes, we display the results for files added (FA), files deleted (FD), files modified (FM), and the total number of files changed in Table 3. It can be observed that the total number of files changed for SATD repayment is marginally lower

Table 3 The average number of files added (FA), deleted (FD), and modified (FM).

Type	FA	FD	FM	Total
SATD	<u>0.35</u>	**0.34**	<u>3.65</u>	<u>4.34</u>
Non-SATD	**0.61**	<u>0.21</u>	**3.74**	**4.56**

than non-SATD changes. Furthermore, the results show that SATD changes have a lower number of files added (FA) and a higher number of files deleted (FD) compared to non-SATD changes, while the differences in files modified (FM) are relatively minor.

To evaluate the effort involved in handling ripple effects, Table 4 presents the mean values of low (LCC), medium (MCC), high (HCC), and crucial (CCC) significance level code changes. The results reveal a significantly higher number of code changes across all significance levels for SATD repayment compared to non-SATD changes. Notably, SATD repayment changes involve a substantially higher number of medium-significance code changes (MCC) compared to non-SATD changes. Our statistical analysis confirms that the differences between SATD and non-SATD changes are statistically significant with respect to LA, LD, FA, FD, FM, and different levels of code changes, with a p-value less than 0.05, using the Mann-Whitney test (Mann and Whitney, 1947).

Table 4 The average number of low (LCC), medium (MCC), high (HCC), and crucial (CCC) significance level code changes.

Type	LCC	MCC	HCC	CCC	Total
SATD	**6.28**	**4.34**	**0.67**	**0.77**	**12.06**
Non-SATD	5.41	2.98	0.39	0.49	9.27

Although SATD and non-SATD items require ***similar*** *levels of effort for direct resolution, SATD items demand* ***higher*** *effort in managing the associated ripple effects.*

4.2 RQ2: Are there differences in repayment effort among different types of SATD items?

We assess the direct resolution effort for various SATD types by presenting the mean values of lines added (LA), lines deleted (LD), and total lines changed for each SATD type and non-SATD items in Table 5. Documentation debt changes demonstrate significantly lower levels of lines added (LA) and lines deleted (LD) compared to non-SATD changes. In contrast, requirement and test debt SATD changes exhibit a substantial increase in the number of lines added (LA) compared to non-SATD changes. Code/design debt changes show a lower number of lines added (LA) and a higher number of lines deleted (LD) compared to non-SATD changes.

Table 6 summarizes the results of files added (FA), files deleted (FD), files modified (FM), and the total number of files changed for each SATD

Table 5 The average number of lines added (LA) and deleted (LD).

Type	LA	LD	Total
Code/Design Debt	70.4	**74.8**	145.2
Documentation Debt	51.8	25.8	77.6
Requirement Debt	**126.0**	47.7	**173.7**
Test Debt	116.0	47.6	163.6
Non-SATD	88.0	41.1	129.1

type and non-SATD items, further assessing the direct resolution effort. We can observe that documentation debt changes result in significantly fewer file updates, whereas changes associated with requirement and test debt lead to a substantial increase in the number of files added compared to non-SATD changes; this is similar to the results in Table 5. Moreover, we observe a similar trend for code/design debt changes, with fewer files added (FA) and more files deleted (FD) than non-SATD changes. Interestingly, our findings indicate that test debt changes have the least number of files modified (FM) compared to other types of SATD and non-SATD changes.

Table 6 The average number of files added (FA), deleted (FD), and modified (FM).

Type	FA	FD	FM	Total
Code/Design Debt	0.34	**0.41**	**3.86**	**4.61**
Documentation Debt	0.24	0.10	3.02	3.36
Requirement Debt	**0.83**	0.19	3.28	4.30
Test Debt	**0.83**	0.24	2.43	3.50
Non-SATD	0.61	0.21	3.74	4.56

To evaluate the effort involved in handling ripple effects, Table 7 summarizes the number of code changes of low (LCC), medium (MCC), high (HCC), and crucial (CCC) significance levels, for each type of SATD and non-SATD changes. As we can see, documentation debt changes result in the lowest number of code changes across different significance levels. In contrast, SATD changes associated with requirement debt exhibit the highest number of code changes across all significance levels, significantly exceeding non-SATD changes. Code/design and test debt changes display a similar trend, following requirement debt in terms of the number of code changes across different significance levels, which are also significantly higher than non-SATD changes.

Various types of SATD items demonstrate ***distinct levels*** *of repayment effort.* ***Documentation debt*** *presents the lowest repayment effort, which*

Table 7 The average number of low (LCC), medium (MCC), high (HCC), and crucial (CCC) significance level code changes.

Type	LCC	MCC	HCC	CCC	Total
Code/Design Debt	7.19	5.13	0.79	0.92	14.03
Documentation Debt	1.28	0.66	0.14	0.12	2.20
Requirement Debt	**9.76**	**5.83**	**0.87**	**1.02**	**17.48**
Test Debt	8.51	4.08	0.49	0.56	13.64
Non-SATD	5.41	2.98	0.39	0.49	9.27

is even lower than that of non-SATD items, while ***requirement debt*** *leads to the highest repayment effort among all SATD types.*

4.3 RQ3: Can we accurately predict the effort required for SATD repayment based on the SATD text?

To assess the efficacy of various machine learning approaches in predicting the required code changes to repay SATD, we compared two deep learning methods (i.e., BERT and TextCNN) and three classical machine learning methods (i.e., RF, LR, and SVR) against a baseline approach (i.e., naive). Due to the substantial variability in the predicted values, we applied a log transformation to the target variable and standardized the target. This transformation produced a more normal distribution, which subsequently improved the learning capabilities of the evaluated machine learning models.

Table 8 Comparison of RMSE performance: machine learning models versus baseline approach for predicting lines added (LA), lines deleted (LD), and total lines of code changed (LT).

Approach	LA	LD	LT	Avg.	IOB
BERT	**0.58**	**0.63**	**0.61**	**0.61**	**56.7%**
TextCNN	0.59	0.65	0.62	0.62	56.0%
RF	0.85	0.91	0.84	0.87	38.3%
LR	1.04	1.10	1.06	1.07	24.1%
SVR	0.71	0.76	0.75	0.74	47.5%
Naive	1.41	1.41	1.41	1.41	-

The results presented in Table 8 provide a comprehensive comparison of different approaches in predicting the lines added (LA), lines deleted (LD), and total lines changed (LT) required for SATD repayment based on the SATD text. The *Improvements over Baseline* (IOB) are also provided in the table. It is noted that the best results are highlighted in bold. As shown in Table 8, the BERT-based approach achieves the lowest RMSE values for LA (0.58), LD (0.63), and LT (0.61), with an average RMSE of 0.61, closely followed by TextCNN with an average RMSE of 0.62. The BERT-based approach yields a 56.7% improvement over the baseline method. Furthermore, the average RMSE scores obtained by the classical machine learning approaches range from 0.74 to 1.07, which are significantly lower than the naive baseline (1.41) but higher than the BERT approach.

Table 9 Comparison of RMSE performance: machine learning models versus baseline approach for predicting files added (FA), deleted (FD), modified files (FM), and the total number of files affected (FT).

Approach	FA	FD	FM	FT	Avg.	IOB
BERT	**0.70**	**0.69**	**0.69**	**0.68**	**0.69**	**50.7%**
TextCNN	**0.70**	**0.69**	0.70	0.69	0.70	50.4%
RF	0.91	0.90	0.94	0.94	0.92	33.3%
LR	1.19	1.23	1.17	1.16	1.18	15.9%
SVR	0.83	0.84	0.80	0.79	0.81	42.2%
Naive	1.41	1.41	1.41	1.41	1.41	-

We also predicted the number of files that need to be added (FA), deleted (FD), and modified (FM), as well as the total number of affected files (FT). The RMSE of the different approaches is displayed in Table 9. The deep learning approach (BERT) consistently demonstrates the lowest RMSE across all predicted values, with values of 0.70 for FA, 0.69 for FD, 0.69 for FM, and 0.68 for FT, resulting in an average RMSE of 0.69. This performance signifies a 50.7% improvement over the baseline method. Similarly, the performance of TextCNN is slightly worse than BERT (0.70 vs 0.69). Moreover, the average RMSE scores obtained by the classical machine learning approaches range from 0.81 to 0.92, which are substantially lower than the naive baseline.

In addition to predicting the changed lines of code and number of files required to repay SATD, we investigated the application of machine learning approaches to predict the number of code changes of the various significance levels, based on the SATD text. Table 10 presents the RMSE performance of the approaches in predicting different significance levels of code changes required for SATD repayment. As shown in Table 10, TextCNN achieved the best performance in predicting different significance levels of code changes, with an average RMSE of 0.67, constituting a 51.7% improvement over the baseline approach. Specifically, the model exhibits the best performance in predicting LCC with an RMSE of 0.62, MCC with an RMSE of 0.72, HCC with an RMSE of 0.67, and CCC with an RMSE of 0.71. Notably, the BERT-based approach is surpassed by TextCNN in this task by a small margin.

Table 10 Comparison of RMSE performance: machine learning models versus baseline approach for predicting different levels of code changes.

Approach	LCC	MCC	HCC	CCC	Avg.	IOB
BERT	**0.60**	**0.68**	0.74	0.72	0.69	51.0%
TextCNN	0.62	0.72	**0.67**	**0.71**	**0.67**	**51.7%**
RF	0.91	0.92	0.94	0.93	0.92	34.3%
LR	1.06	1.10	1.17	1.15	1.12	20.5%
SVR	0.73	0.74	0.82	0.79	0.77	45.3%
Naive	1.41	1.41	1.41	1.41	1.41	-

Additionally, the three classical machine learning models exhibit better RMSE performance than the baseline approach, with values ranging from 0.77 to 1.12.

Machine learning approaches, specifically the BERT and TextCNN, can be ***effective in predicting the effort required for SATD repayment*** *based on the SATD text, outperforming the naive baseline by a large margin.*

4.4 RQ4: What keywords are associated with varying levels of repayment effort when repaying SATD?

We begin by summarizing the keywords linked to the effort required for addressing SATD items. Using the deconvolution technique (refer to Section 3.5.3), we identify and present the top keywords associated with low and high lines of code changed (including LA and LD) as well as low and high numbers of files changed (including FA, FD, and FM) in Table 11. Unique keywords are highlighted in bold. Our analysis reveals that when the lines of code and number of files modified are low, the keywords generally relate to typos, error message updates, warning message updates, or code comments. We also observe that SATD items involving unused imports, logging, workarounds, or debugging consistently require low repayment effort, measured in terms of lines of code and number of files. In contrast, SATD items related to code cleanup (e.g., *code cleanup*, *formatting*, and *rename*), tests, documentation (e.g., *documentation* and *license header*), and requirements (e.g., *work in progress*, *improvement*, and *support for*) generally demand more lines of code to repay. Interestingly, keywords for high numbers of files modified differ from those for high lines of code. While keywords related to high numbers of files modified also pertain to code cleanup (e.g., *code cleanup*, *naming*, and *tidy up*), they involve changes to interfaces and classes as well (e.g., *interface*, *class*, and *extension point*).

To summarize keywords connected to the effort in handling ripple effects, we provide an overview of the keywords identified for code changes with varying levels of significance (i.e., low, medium, high, and crucial) during SATD

Table 11 Top keywords associated with low or high levels of SATD repayment effort with respect to the number of lines of code modified and the number of files modified.

Low # Lines	High # Lines	Low # Files	High # Files
typo	code cleanup	typo	header
unused import	**formatting**	unused import	**interface**
error message	**more tests**	comment	code cleanup
comment	**documentation**	**warning**	**annotation**
logging	**work in progress**	debug	naming
javadoc	**improvement**	**workaround**	**class**
minor	rename	**proper**	**tidy up**
update	**support for**	**variable**	**files**
debug	header	error message	**extension point**

Table 12 Top keywords that are associated with varying levels of significant changes for repaying SATD.

LCC	MCC	HCC	CCC
logging	handling	unused code	unused code
exception	logging	interface	interface
handling	simplify	API	refactoring
test	**logic**	implementation	API
output	**catch**	code cleanup	support
cast	output	refactoring	**deprecated code**
simplify	code cleanup	support	implementation
findbugs	leak	**checkstyle errors**	**constructor**
leak	implementation	**redundant**	**endpoints**

repayment. Table 12 provides a comprehensive overview of our findings, with unique keywords (exclusive to one significance level) highlighted in bold for easy reference. The keywords show that changes with low or medium significance levels primarily focus on exception handling, logging, tests, logic improvement, and fixing leak issues. On the other hand, changes with high or crucial significance levels predominantly involve cleaning up unused code, modifying interfaces, refactoring code, and implementing new requirements. Some keywords are shared between different levels of significance, as certain tasks in software development are fundamental and ubiquitous across all complexity levels.

Different types of SATD repayment efforts are associated with distinct keywords. ***Low-effort repayments*** *typically involve typos, error messages, and code comments, while* ***high-effort repayments*** *often require code cleanup, modifying interfaces, and implementing requirements.*

5 Discussion

5.1 Repayment Effort in SATD and Non-SATD Changes

Our findings show that SATD repayment changes involve a marginally greater number of total lines changed compared to non-SATD changes, while displaying a slightly lower total number of files changed. These results contradict the prior study by Wehaibi *et al.* (Wehaibi et al., 2016), which suggested that SATD changes have significantly higher values in terms of both the total number of lines and files changed. One possible explanation for this discrepancy is that the previous study employed 62 SATD keywords (Potdar and Shihab, 2014), which might have identified more severe and error-prone SATD items. Another potential factor is the difference in SATD identification methods: our study identifies SATD from commit messages, while the previous study (Wehaibi et al., 2016) used code comments, possibly leading to contrasting results. We recommend that future research further examines the disparities in repayment effort between SATD and non-SATD items to gain a deeper understanding of the factors influencing these differences. Comparisons of various SATD identification techniques and the severity levels of the identified items could offer valuable insights into the causes of these discrepancies.

Furthermore, we observed significant differences in the lines added (LA) and lines deleted (LD) between the two types of changes. In particular, SATD repayment changes involve fewer lines added (LA) and more lines deleted (LD) compared to non-SATD changes. This pattern extends to files, with SATD changes showing a lower number of files added (FA) and a higher number of files deleted (FD) compared to non-SATD changes. A possible explanation for these differences is that non-SATD changes are more likely to involve feature implementation, which typically adds more lines and files while deleting fewer. In contrast, SATD repayment changes may involve more refactoring, which requires modifying and deleting lines of code and files rather than adding new ones.

Regarding the significance level of code changes, our findings demonstrate that SATD repayment changes involve a substantially higher number of code changes across various significance levels compared to non-SATD, suggesting that repaying SATD creates more ripple effects. This indicates that addressing SATD may require more extensive and diverse code modifications, impacting multiple aspects of the codebase. A possible explanation for this observation is that SATD items may be more deeply embedded within the code, necessitating a greater degree of intervention to resolve. For practitioners, this emphasizes the importance of early detection and resolution of SATD items in order to minimize the impact of these ripple effects and maintain code quality.

5.2 Repayment Effort among Different Types of SATD Items

Our analysis of various types of SATD items uncovers significant differences in repayment effort among them. Repaying documentation debt shows the lowest levels of lines added (LA) and lines deleted (LD) compared to other SATD and non-SATD changes. This finding implies that repaying documentation debt might be less effort-intensive, as it often requires updating comments and documentation without substantially altering the codebase. On the other hand, SATD repayment related to requirement and test debt display a significant increase in the number of lines of code added (LA) compared to non-SATD changes. This observation suggests that repaying requirement and test debt may involve more extensive codebase modifications, as developers need to implement new features or enhance existing ones to meet evolving requirements and ensure sufficient test coverage. For code/design debt changes, our results indicate that they exhibit a lower number of lines added (LA) and a higher number of lines deleted (LD) compared to non-SATD changes. This pattern suggests that addressing code/design debt may involve increased refactoring and optimization effort, as developers focus on improving code structure and eliminating unnecessary or problematic code elements.

Considering the variations in repayment effort among distinct types of SATD, we advise researchers and practitioners to treat each SATD type distinctly rather than considering SATD as a monolithic entity. Resolving documentation debt may require less effort, while addressing requirement debt may demand more repayment effort. This tailored approach enables a nuanced understanding of specific challenges and requirements associated with each SATD type, leading to more effective management strategies. Previous research primarily focused on examining SATD as a whole (Sierra et al., 2019), resulting in limited attention to individual types. By considering different SATD types, researchers and practitioners can better understand, prioritize, and develop more effective strategies to address each type. Consequently, it is crucial to devise approaches for supporting other SATD types, such as architecture or build debt, which are difficult to detect automatically. A detailed understanding of repayment efforts for various SATD types allows researchers to create advanced tools to support SATD management.

5.3 Predicting SATD Repayment Effort Based on SATD Text

Our evaluation of machine learning approaches for predicting the effort required for SATD repayment based on SATD text reveals that the deep learning methods (BERT and TextCNN) outperform classical machine learning methods regarding RMSE performance. The superior performance of deep learning methods can be attributed to their ability to capture complex patterns and relationships within the SATD text, enabling more accurate predictions of repayment effort. These results highlight the potential of using machine learning approaches, especially deep learning methods, to assist developers in estimating the effort necessary to repay SATD. By understanding the estimated repayment effort for each SATD item, software developers and project managers can make informed decisions about how to allocate team resources effectively.

However, it is important to note that the performance of the models in predicting the effort required for SATD repayment is not perfect, and there is room for improvement (further decreasing the RMSE). Additionally, our work predicts SATD repayment effort based on SATD text in commit messages, and it remains uncertain whether the trained machine learning approach can be applied to predict repayment effort using SATD documented in other sources, such as code comments. Consequently, we encourage researchers to explore the applicability of the proposed machine learning approach for predicting SATD repayment effort across different types of SATD documentation sources, including code comments, issue trackers, and pull requests. Researchers could also investigate the possibility of combining multiple SATD documentation sources to enhance the accuracy and reliability of repayment effort prediction. Moreover, future research can examine how incorporating human expertise and domain knowledge can complement machine learning models for predicting SATD repayment effort. This might involve developing hybrid approaches that integrate automated techniques with expert judgment to yield more accurate and actionable insights for managing technical debt.

5.4 Keywords Associated With Varying Levels of Repayment Effort

We summarized the top keywords associated with different levels of repayment effort in Section 4.4. The primary advantage of using these keywords lies in their interpretability and ease of use. By closely monitoring the presence of these keywords, developers can gain a quick understanding of the potential repayment effort associated with the respective SATD items. This awareness can help facilitate the identification and prioritization of SATD items to be addressed, especially in cases where the deep learning model predictions are unavailable. Moreover, integrating these keywords into automated tools can further enhance the support provided to software developers in prioritizing SATD. By highlighting the keywords in SATD items, such tools can help developers better estimate the repayment effort of SATD items, enabling them to make more informed decisions when addressing technical debt. Combining the deep learning model's predictions with keyword-based insights provides a comprehensive understanding of the repayment effort associated with various SATD items.

6 Threats to Validity

In this section, we discuss the potential threats to the validity of our study, focusing on construct validity, reliability, and external validity.

6.1 Threats to Construct Validity

In our study, we used lines of code, files changed, and significance levels of code changes as proxies to measure repayment effort. These metrics may not capture all aspects of the effort involved in SATD repayment. We partially mitigated this threat by employing widely-accepted metrics to quantify the repayment effort. Additionally, there is a possibility that the approach used to identify SATD items may not capture all types of SATD, which may impact the results of our study. Indeed, we used a machine learning model trained on commit messages to identify SATD items, and it may miss some SATD types that are not typically described in commit messages. We mitigated this threat, at least partially, by using a well-established model from previous research, which has demonstrated high accuracy in identifying various types of SATD.

6.2 Threats to Reliability

Reliability refers to the consistency and stability of the results obtained across different instances of the study. In our research, we employed various machine learning models, which inherently possess some level of randomness due to their training process. To mitigate this threat, we reported the average performance across multiple runs of the models. Moreover, we made our collected dataset publicly available in the replication package[1], allowing for replication and verification of the results.

6.3 Threats to External Validity

External validity is concerned with the generalizability of our findings to other contexts or populations. Our study focused on a specific dataset containing Java open-source projects. Although these projects vary in size and complexity, they might not be representative of all types of software projects. Moreover, the generalizability of our findings to other programming languages and domains remains uncertain. To improve external validity, future studies should consider investigating SATD repayment effort in different languages, domains, and project types, as well as incorporating more diverse sources of data, such as issue trackers and code review comments.

7 Conclusion

In this study, we investigated the effort required to repay SATD items and non-SATD changes. Our research aimed to provide a better understanding of the differences in repayment effort between SATD and non-SATD items, the variations among different types of SATD items, the feasibility of predicting repayment effort based on the textual content of SATD items, and the keywords that are associated with different levels of repayment effort.

References

Alves NS, Ribeiro LF, Caires V, Mendes TS, Spínola RO (2014) Towards an ontology of terms on technical debt. In: 2014 Sixth International Workshop on Managing Technical Debt, IEEE, pp 1–7

Avgeriou P, Kruchten P, Ozkaya I, Seaman C (2016) Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162). Dagstuhl Reports 6(4):110–138, DOI 10.4230/DagRep.6.4.110

Brown N, Cai Y, Guo Y, Kazman R, Kim M, Kruchten P, Lim E, MacCormack A, Nord R, Ozkaya I, et al. (2010) Managing technical debt in software-reliant systems. In: Proceedings of the FSE/SDP workshop on Future of software engineering research, pp 47–52

Chatterjee S, Hadi AS (2006) Regression analysis by example. John Wiley & Sons

Devlin J, Chang MW, Lee K, Toutanova K (2018) Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:181004805

Drucker H, Burges CJ, Kaufman L, Smola A, Vapnik V (1996) Support vector regression machines. Advances in neural information processing systems 9

Fluri B, Gall HC (2006) Classifying change types for qualifying change couplings. In: 14th IEEE International Conference on Program Comprehension (ICPC'06), IEEE, pp 35–45

Fluri B, Wursch M, PInzger M, Gall H (2007) Change distilling: Tree differencing for fine-grained source code change extraction. IEEE Transactions on software engineering 33(11):725–743

Guo Z, Liu S, Liu J, Li Y, Chen L, Lu H, Zhou Y, Xu B (2019) Mat: A simple yet strong baseline for identifying self-admitted technical debt. arXiv preprint arXiv:191013238

Guo Z, Liu S, Liu J, Li Y, Chen L, Lu H, Zhou Y (2021) How far have we progressed in identifying self-admitted technical debts? a comprehensive empirical study. ACM Transactions on Software Engineering and Methodology (TOSEM) 30(4):1–56

Ho TK (1995) Random decision forests. In: Proceedings of 3rd international conference on document analysis and recognition, IEEE, vol 1, pp 278–282

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

Kim Y (2014) Convolutional neural networks for sentence classification. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp 1746–1751

Li Y, Soliman M, Avgeriou P (2020) Identification and Remediation of Self-Admitted Technical Debt in Issue Trackers. Proceedings - 46th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2020 pp 495–503, DOI 10.1109/SEAA51224.2020.00083, 2007.01568

Li Y, Soliman M, Avgeriou P (2022a) Identifying self-admitted technical debt in issue tracking systems using machine learning. Empirical Software Engineering 27(131), DOI 10.1007/s10664-022-10128-3

Li Y, Soliman M, Avgeriou P, Somers L (2022b) Self-admitted technical debt in the embedded systems industry: An exploratory case study. IEEE Transactions on Software Engineering pp 1–22, DOI 10.1109/TSE.2022.3224378

Li Y, Soliman M, Avgeriou P (2023a) Automatic identification of self-admitted technical debt from four different sources. Empirical Software Engineering 28(65), DOI 10.1007/s10664-023-10297-9

Li Y, Soliman M, Avgeriou P (2023b) Automatically identifying relations between self-admitted technical debt across different sources. arXiv preprint arXiv:230307079

Liu Q, Liu Z, Zhu H, Fan H, Du B, Qian Y (2019) Generating commit messages from diffs using pointer-generator network. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), IEEE, pp 299–309

Maldonado EdS, Shihab E (2015) Detecting and quantifying different types of self-admitted technical debt. In: 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD), IEEE, pp 9–15

Maldonado EdS, Abdalkareem R, Shihab E, Serebrenik A (2017) An empirical study on the removal of self-admitted technical debt. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 238–248

Mann HB, Whitney DR (1947) On a test of whether one of two random variables is stochastically larger than the other. The annals of mathematical statistics pp 50–60

Martini A, Sikander E, Madlani N (2018) A semi-automated framework for the identification and estimation of architectural technical debt: A comparative case-study on the modularization of a software component. Information and Software Technology 93:264–279

Mensah S, Keung J, Svajlenko J, Bennin KE, Mi Q (2018) On the value of a prioritization scheme for resolving self-admitted technical debt. Journal of Systems and Software 135:37–54

Nugroho A, Visser J, Kuipers T (2011) An empirical model of technical debt and interest. In: Proceedings of the 2nd workshop on managing technical debt, pp 1–8

Potdar A, Shihab E (2014) An exploratory study on self-admitted technical debt. In: 2014 IEEE International Conference on Software Maintenance and Evolution, IEEE, pp 91–100

Ren X, Xing Z, Xia X, Lo D, Wang X, Grundy J (2019) Neural network-based detection of self-admitted technical debt: From performance to explainability. ACM Transactions on Software Engineering and Methodology (TOSEM) 28(3):1–45

Sierra G, Shihab E, Kamei Y (2019) A survey of self-admitted technical debt. Journal of Systems and Software 152:70–82, DOI https://doi.org/10.1016/j.jss.2019.02.056

van Solingen R, Basili V, Caldiera G, Rombach HD (2002) Goal Question Metric (GQM) approach. In: Encyclopedia of Software Eng., John Wiley & Sons, Inc., Hoboken, NJ, USA, pp 528–532

Wehaibi S, Shihab E, Guerrouj L (2016) Examining the impact of self-admitted technical debt on software quality. In: 2016 IEEE 23Rd international conference on software analysis, evolution, and reengineering (SANER), IEEE, vol 1, pp 179–188

Xiao L, Cai Y, Kazman R, Mo R, Feng Q (2016) Identifying and quantifying architectural debt. In: Proceedings of the 38th international conference on software engineering, pp 488–498

Zampetti F, Serebrenik A, Di Penta M (2018) Was self-admitted technical debt removal a real removal? an in-depth perspective. In: 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), IEEE, pp 526–536

Zampetti F, Fucci G, Serebrenik A, Di Penta M (2021) Self-admitted technical debt practices: a comparison between industry and open-source. Empirical Software Engineering 26(6):1–32