*Journal of Software Engineering Research and Development, 2025, 13:2,* doi: 10.5753/jserd.2025.4801

# Technical Debt of Software Projects Based on Merge Code Comments

**Marcos Henrique Melo de Araújo** [ **Universidade Federal do Acre** | ***henrique.marcos@sou.ufac.br*** ]
**Catarina Costa** [ **Universidade Federal do Acre** | ***catarina.costa@ufac.br***]
**Awdren Fontão** [ **Universidade Federal de Mato Grosso do Sul** | ***awdren.fontao@ufms.br***]

**Abstract**
Developers use code comments for various reasons, such as explaining code, documenting specifications, communicating with other developers, and highlighting upcoming tasks. Software projects with minimal documentation often have a significant number of comments. In this sense, the code comment analysis technique can be used to examine more complex aspects of software projects, such as TD generated by merge conflicts. The TD resulting from the resolution of merge conflicts occurs when the resulting code contains comments that indicate tasks to be performed in the future. There are no studies directly linking merge conflicts and TD. The purpose of this study is to identify and analyze code comments generated when resolving merge conflicts from this perspective. This process can lead to improvements in software quality and help in managing TD. To this end, an exploratory analysis was carried out in 100 software projects, with a specific focus on task annotations originating from the resolution of merge conflicts. The results revealed that 60.61% of the analyzed projects have at least one code comment indicating the creation or maintenance of TD. In addition, metrics such as accuracy, precision, recall, and F1-score were applied across different software projects to enhance the effectiveness and reliability of the built data model. The metric results suggest that the tool was able to correctly classify in most cases, but was not particularly precise in this classification due to the size of the analyzed projects and the variety of task annotations, including the presence of some and the absence of others.

***Keywords:*** *merge conflicts, technical debt, software engineering*

## 1 Introduction

In the development process, merge conflicts occur when developers modify the same section of a file. Once a conflict is identified, a team member must reconcile the conflicting versions by maintaining both versions, choosing one, or writing new lines of code. Therefore, version control can directly impact the efficiency of software, both in terms of the time to deliver from the development team and the quality of the resulting product (Ahmed et al., 2017).

The task of resolving merge conflicts can affect the development flow, leading to a change in the focus of developers (Mahmoudi et al., 2019). This task requires a deep understanding of the software. In more complex conflicts, developers may lack the knowledge necessary to make the most appropriate decision (Nieminen, 2012), which can affect the quality of the code. The code involved in a merge conflict is more likely to be related to future defects, which can generate TD (Mahmoudi et al., 2019).

The concept of technical debt (TD) addresses an issue in software project development. It is a metaphor referring to the consequences of decisions that prioritize code delivery over problem resolution, whether known or unknown. Initially proposed in the literature to explain to stakeholders the need for refactoring, Cunningham gave the original definition as "a task that is internal in nature and is not chosen to be performed now, but has the potential to cause future problems if not done" (Cunningham, 1992).

Lack of TD management poses risks to the project and hinders its progress, such as improper prioritization of debt repayment or even complete ignorance of it (referred to as unacknowledged TD). Over the years, different management approaches and strategies have been presented by academia and industry, but considering actions to prevent its increase or maintenance is still not a common practice (Rios et al., 2021). In this sense, studying TD can help development teams make decisions to avoid its occurrence or maintenance.

The introduction of TD can occur through task annotations, where code comments such as `TODO`, `FIXME`, and `XXX` are used to document how developers communicate and highlight tasks throughout the development process. According to Storey et al. (2008), code comments are a generic type of task annotation, where developers embed documentation directly into the source code. Currently, modern integrated development environments such as Visual Studio, Eclipse, IntelliJ IDEA, and NetBeans support various approaches in managing developers' tasks, such as task annotations.

Code comments play a vital role as documentation used in code maintenance tasks. They complement the original system documentation and are a conventional practice for developers to keep the code clean and up-to-date. They are also considered extremely important for understanding functionalities and for effective communication among members of the development team (Potdar and Shihab, 2014).

In existing literature, some works investigate the practical aspect of TD (Kruchten et al., 2012), while others seek to identify and classify it (Huang et al., 2018; de Freitas Farias et al., 2015; Maldonado and Shihab, 2015; Ren et al., 2019), studying its causes and effects (Rios et al., 2021).

Until the publication of the previous study (Melo de Araújo et al., 2023), the relationship between merge conflict resolution and TD had not been thoroughly explored, from the perspective of code comments. While existing research had investigated various sources and impacts of TD, no prior work had specifically analyzed how the process of resolving merge conflicts contributes to the introduction or maintenance of TD through the annotations and comments left by developers. This gap in the literature is significant, as code comments especially those made during merge conflict resolution often contain valuable insights into unresolved issues, temporary workarounds, or areas requiring further attention. By examining these comments, it becomes possible to gain a deeper understanding of how collaborative development practices influence the accumulation and persistence of TD. This focus not only highlights the importance of merge conflict resolution as a critical moment in the software lifecycle but also underscores the potential of leveraging code comments as a key artifact for TD management.

Some experimental studies utilize findings related to TD to improve software maintenance (Kruchten et al., 2013, 2012; Seaman et al., 2012). Other studies investigate performance differences between automated tools and humans in the task of detecting TD (Bavota and Russo, 2016; Lim et al., 2012). Based on the findings, there is a small overlap between the capabilities of automated tools and humans. While automated tools are considered more efficient in detecting defect-related TD, humans have a greater ability to identify more abstract categories of TD (Ren et al., 2019).

In this context, this study aims to investigate and identify the occurrence of TD present in merges that presented conflicts, using code comments. It was found that out of 100 software projects analyzed, 60.61% of them have some code comment indicating the introduction or maintenance of TD. Four types of debts were identified in the code snippets: code debt (1,754), documentation (643), defects (561), and testing (160). A total of 841,282 comments were analyzed, of which 3,118 were identified as signs of introducing TD. The task annotation with the highest number of occurrences is `TODO` (1,518), followed by `ERROR` (362), `NOTE` (331), `DONE` (232), `REVIEW` (218), `REMOVE` (160), `BUG` (101), `FIXME` (78), `XXX` (58), `FUTURE` (39), `ISSUE` (22), `BROKEN` (20), and `HACK` (18).

This paper is an extended version of a conference paper (Melo de Araújo et al., 2023) in which we answered three research questions, focused on the task annotations that are most associated with the project's TD, types of TD that are most common in merge conflicts and whether conflict resolution contributes to increasing or maintaining the project's TD. This work complements our previous work by adding three new research questions, focused on the how effective is the computational model created in classifying the TD of the selected projects, how our tool adapts to different task annotations in the comments of files resolving merge conflicts and how the TD classification performs in the context of the selected projects.

This study is structured as follows: Section 2 presents the background, Section 3 reviews related work, Section 4 outlines the methodology, Section 5 details the results, Section 6 provides the discussion, Section 7 addresses threats to validity, and Section 8 concludes the study.

## 2 Background

This section presents the fundamental concepts related to TD, merge conflicts, and the connection between TD and the resolution of merge conflicts, with a particular focus on self-admitted TD.

### 2.1 Technical Debt and Self-Admitted Technical Debt

The concept of TD was first introduced by Cunningham (1992), as a metaphor to describe the trade-offs made in software development when developers prioritize short-term deliverables over long-term code quality. TD can manifest in various forms, such as suboptimal code, insufficient documentation, or poor testing practices, which accumulate and lead to higher maintenance costs and reduced system reliability.

Self-admitted TD (SATD) refers to instances where developers explicitly acknowledge the presence of TD within the codebase. Previous research (Maldonado et al., 2017a) has shown that SATD annotations are often associated with different types of TD, including code, defect, and documentation debt, making them valuable for both identifying and managing TD.

### 2.2 Merge Conflicts in Software Development

Merge conflicts occur when parallel changes made by different developers to the same section of code cannot be automatically reconciled by version control systems. Resolving merge conflicts often requires manual intervention, where developers must analyze conflicting changes and integrate them into a consistent codebase. Merge conflicts are a common challenge in collaborative software development, especially in projects with high development velocity and large teams.

The process of resolving merge conflicts is inherently complex and error-prone, as it involves understanding the intent behind conflicting changes and ensuring the functionality of the integrated code. Poorly managed merge conflicts can lead to degraded code quality, introduction of defects, and, consequently, the accumulation of TD.

### 2.3 Technical Debt and Merge Conflict Resolution

Merge conflict resolution is closely linked to the introduction and management of TD. Several studies (Zhao et al., 2017; Tsantalis et al., 2018) have highlighted that the manual resolution of merge conflicts often results in TD due to shortcuts taken to quickly resolve conflicts. For instance, developers may introduce temporary solutions or leave comments indicating areas that require further attention, such as `TODO` or `FIXME`, thereby increasing SATD.

This connection between merge conflicts and TD underscores the importance of integrating TD management practices into the merge resolution process. Tools and techniques that automate or support the identification of TD during merge conflict resolution can help mitigate its accumulation.

Furthermore, analyzing comments left during merge conflict resolution provides a unique opportunity to study the interplay between TD and collaborative development practices.

In this study, we build upon these foundations by exploring how task annotations in merge conflict resolution comments are associated with different types of TD. Our findings aim to bridge the gap between merge conflict management and TD mitigation strategies, contributing to better practices for managing code quality in collaborative software projects.

## 3 Related Work

In this research, studies that support the identification and classification of TD through code comment analysis were investigated. No studies in the literature were found that established a direct relationship between merge conflicts and TD. However, other studies were identified that address the identification and characterization of TD through code comments (Potdar and Shihab, 2014; de Freitas Farias et al., 2015; Maldonado and Shihab, 2015; Huang et al., 2018; Ren et al., 2019; Wang et al., 2024).

Potdar and Shihab (2014) conducted an exploratory study on self-admitted TD. They analyzed the source code comments of four open-source projects. The authors developed a tool that records snippets with code comments into separate text files for each project. They manually summarized 62 patterns that could be used to identify TD comments after reading more than 100,000 comments from different projects. It was observed that TD exists in approximately 2.4% to 31% of the project files. Additionally, experienced developers tend to introduce most of the self-admitted TD. Pressures such as time and code complexity are not related to the amount of self-admitted TD. The authors also observed that only 26.3% to 63.5% of self-admitted TD is removed from the project after being introduced.

de Freitas Farias et al. (2015) developed a vocabulary model to identify TD through code comments, using the CVM-TD (Contextual Vocabulary Model for TD). This model uses grammatical structures such as pronouns, adjectives, verbs, adverbs, and tags to create a specific vocabulary for TD, aiming to identify different types of occurrences. The authors report that the tags used are a specific set of keywords associated with certain types of comments. These keywords were identified through literature analysis, resulting in a total of 10 tags: `TODO`, `FIXME`, `HACK`, `XXX`, `BUG REMIND`, `REVIEW`, `REVISIT`, `NOTE`, and `REMARK`. The analysis was conducted on two projects, and it was found that the `TODO` tag was the most used, followed by the `NOTE` tag. According to the authors, the meaning attributed to these tags was established individually or through informal agreements by the development team. In general, these tags are used to describe actions that need to be completed or issues that may require future work. The most frequently identified types of TD, in order of occurrence, were: code, design, architecture, test, defect, infrastructure, process, documentation, build, requirement, service, and personnel.

Maldonado and Shihab (2015) conducted a study aimed at detecting and quantifying self-admitted TD through code comments, seeking to understand how this TD is present in software projects. The analysis was conducted on five projects using an Eclipse plugin to convert the source code, extract code comments, and the abstract syntax tree, enabling source code mapping. Additionally, the authors developed a Java tool that read the database obtained from the source code analysis. Approximately 33,000 comments were analyzed, and the results revealed that self-admitted TD can be classified into five main types: design, defect, documentation, requirement, and test TD. It was observed that the most common type of self-admitted TD was design debt, representing between 42% to 84% of the classified comments.

Huang et al. (2018) sought to identify self-admitted TD in open-source projects using text mining. Eight projects containing 212,413 comments were investigated. In the adopted approach, the textual description of all comments was preprocessed, and variable selection was applied to each project, resulting in the identification of the top 10 variables with the highest scores. The most frequently selected variables were: todo, implement, fixme, workaround, hack, yuck, ugly, fix, xxx, stupid, broken, ill. An interesting finding was that the frequency of these words can vary from one project to another. Additionally, the authors observed that when writing comments about TD, some developers prefer to use words with sentimental connotations and more informal language.

Ren et al. (2019) presented an approach that uses Convolutional Neural Networks (CNN) to classify code comments as self-admitted TD. The authors explored the computational structure of CNNs to identify key phrases and patterns in code comments that are most relevant to TD. Several patterns were observed, such as: to do, xxx, hack, fixme, error, remove, implement, among others. The authors also identified five characteristics of comments that affect the performance, generalization, and adaptability of the approach used and the classification of TD based on traditional text mining. The experiments were conducted with 62,566 code comments from 10 open-source projects and a user study with 150 comments from three other projects.

Wang et al. (2024) discuss how `TODO` comments are used to address TD and improve code quality over time, although their implementation often depends on developers' experience and time constraints. To address this issue, TDREMINDER is proposed a method that utilizes neural models to detect commits potentially missing relevant `TODO` comments. The study analyzed 13,556 open-source project repositories. Initially, data were collected from the top 10,000 starred repositories on GitHub for Python and Java languages. Neural models were trained focusing on challenging samples to detect commits that should have included `TODO` comments. Subsequently, repositories were filtered to include only those containing `TODO`, `FIXME`, or `XXX` comments, resulting in 5,467 Python repositories and 3,089 Java repositories.

The key contributions are the introduction of TDREMINDER, and the effectiveness of TDREMINDER is demonstrated through extensive evaluations on large datasets of Python and Java repositories, showing its superiority over existing benchmarks. Furthermore, TDREMINDER provides developers with explanatory recommendations, highlighting the necessity of adding `TODO` comments.

Related works can be compared by (1) number of analyzed projects, (2) number of analyzed comments, and (3) study type, as shown in Table 1.

**Table 1.** Related work overview

| Related Work | Projects | Comments | Type |
|---|---|---|---|
| de Freitas Farias et al. (2015) | 2 | 6,265 | Experimental |
| Hattori and Lanza (2008) | 9 | - | Experimental |
| Huang et al. (2018) | 8 | 212,413 | Experimental/Survey |
| Maldonado and Shihab (2015) | 5 | 33,000 | Experimental |
| Maldonado et al. (2017b) | 5 | 7,749,969 | Empiric/Survey |
| Potdar and Shihab (2014) | 4 | 101,762 | Exploratory |
| Ren et al. (2019) | 10 | 62,566 | Experimental/Survey |
| Wang et al. (2024) | 13,556 | - | Experimental |
| Our | 100 | 841,282 | Experimental |

None of the works take into account merge conflicts, as they utilize code comment analysis to study the identification and classification of self-admitted TD in other development contexts. This is why the number of comments analyzed in the present research is smaller compared to studies that analyze a similar number of projects; it focuses specifically on the context of merge conflicts. This version control scenario highlights the importance of reducing the incidence of introducing TD into the project and managing its repayment.

All analyzed projects are open-source, versioned with Git, and hosted on GitHub, allowing them to be incorporated into experimental studies, sometimes complemented with surveys involving participants such as developers and practitioners. For example, the work of Hattori and Lanza (2008) studies the nature of commits without focusing on merges, merge conflicts, or TD identification, but is considered a baseline for subsequent works on TD analysis using code comment analysis.

The study by Potdar and Shihab (2014) is exploratory and is regarded as one of the most traditional techniques in the literature, utilizing text mining to classify TD. Another technique presented by Ren et al. (2019) proves to be more efficient in classifying TD. Although these two studies use data mining and neural networks, other experimental techniques should also be considered, such as the one used by de Freitas Farias et al. (2015), where a vocabulary model combined with neural networks is created to classify comment text.

In related work, comment analysis was performed on a smaller set of projects, at most 10 according to Ren et al. (2019), except for Wang et al. (2024) which analyze 13,556 projects. This is due to the complexity of the techniques used and the number of comments analyzed, as seen in the work of Maldonado et al. (2017b), which analyzed 7,749,969 comments in just five projects. The literature observes that more sophisticated analysis techniques are more effective in a smaller set of instances, reinforcing the idea that it may not be possible to determine a single way to solve the TD problem. Instead, a combination of techniques and approaches consider different scenarios in software project development is needed.

This research performs a textual analysis of code comments related to the resolution of merge conflicts, allowing the examination of a larger number of projects due to the nature of this technique. This broader scope makes it possible to identify recurring patterns and terms that may not appear in more limited or filtered datasets. Despite using a simpler technique, the research also incorporates a manual analysis step to avoid the influence of false positives. This combination of automatic and manual steps increases the reliability of the results. According to Ren et al. (2019), algorithms can fail in predicting results, especially due to the semantic variety caused by the subjective manner in which comments are written.

The analysis of code comments is an important tool for classifying TD and can be combined with other approaches and tools that facilitate this analysis. Although most of these techniques are experimental, exploratory research highlights important issues in investigating various aspects of TD.

Less complex techniques can bring practical actions to software maintenance and evolution tasks, assisting the development team in performing their activities, contributing to TD management, positively influencing the quality of delivered software, and increasing the development team's efficiency.

## 4 Methodology

This section describes the methodology applied in this study, divided into two main parts: (1) identification and classification of TD and (2) evaluation of the computational model.

The first part deals with the identification and classification of TD and seeks to answer the first three research questions: **(RQ1)** Which task annotations are most associated with the project's TD? **(RQ2)** Which types of TD are most common in merge conflicts? **(RQ3)** Does the resolution of merge conflicts contribute to increasing or maintaining the project's TD?

The second part deals with the evaluation of the computational model created and seeks to answer the last three research questions: **(RQ4)** How effective is the computational model in classifying the TD of the selected projects? **(RQ5)** How did the TDCA Tool [1] adapt to the different task annotations in the comments of the files that resolve merge conflicts? **(RQ6)** How did the TD classification perform in the context of the selected projects?

The first three research questions (**RQ1**, **RQ2**, and **RQ3**) were initially addressed in our previous work (Melo de Araújo et al., 2023), where we explored the relationship between task annotations and TD in the context of merge conflict resolution. This paper extends that work by introducing a computational model designed to classify TD and validate its effectiveness. The remaining research questions (**RQ4**, **RQ5**, and **RQ6**) are specific to this study and focus on evaluating the proposed model and its ability to generalize and adapt to diverse project contexts. By addressing these new questions, this paper provides a broader and more robust understanding of TD management, building upon the foundations established in the earlier study.

### 4.1 Identification and Classification of Technical Debt

The identification and classification of TD were conducted through the following steps: (1) selection of software projects, (2) data extraction from the repositories, (3) analysis of code comments in the files, and (4) identification of

[1] https://github.com/marcostorm/TDCATool.git

TD. Figure 1 provides an overview of the approach used in this study.

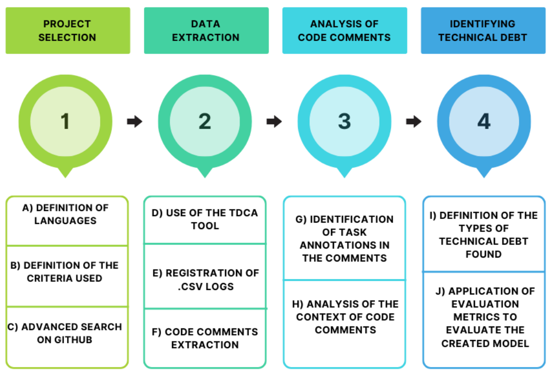
PROJECT SELECTION
DATA EXTRACTION
ANALYSIS OF CODE COMMENTS
IDENTIFYING TECHNICAL DEBT
1
2
3
4
A) DEFINITION OF LANGUAGES
B) DEFINITION OF THE CRITERIA USED
C) ADVANCED SEARCH ON GITHUB
D) USE OF THE TDCA TOOL
E) REGISTRATION OF .CSV LOGS
F) CODE COMMENTS EXTRACTION
G) IDENTIFICATION OF TASK ANNOTATIONS IN THE COMMENTS
H) ANALYSIS OF THE CONTEXT OF CODE COMMENTS
I) DEFINITION OF THE TYPES OF TECHNICAL DEBT FOUND
J) APPLICATION OF EVALUATION METRICS TO EVALUATE THE CREATED MODEL

**Figure 1.** Steps of research

**1) Selection of Software Projects:** In this stage, software project repositories hosted on GitHub were selected through three steps. Step (a) involved defining the programming languages. The eight main programming languages used in 2023 were identified from the GitHub survey [2], the Stack Overflow survey [3], and the TIOBE index [4], namely C, C++, C#, Java, JavaScript, PHP, Python, and Ruby. The programming languages to be analyzed were defined at the outset since they use different ways of writing code comments, which will be filtered in Stage 3, the identification of task annotations present in the code comments.

After identifying the programming languages, selection criteria for the repositories were defined in step (b). The following criteria were adopted:

**I) Project Popularity:** Only projects with 1000 or more stars were considered, given that developers often use this function to express interest or satisfaction in ongoing projects. Stars play an important role in popularizing and ranking repositories, making them more visible and relevant to those seeking them out (Borges and Valente, 2018). This criterion ensures that the selected repositories are relevant projects in the technology community, reflecting more realistic development scenarios.

**II) Project Nature:** Only software projects were included. Although there are many popular repositories with a large number of stars, not all necessarily correspond to software projects. This popularity can be attributed to various types of content, such as images, academic work from various fields, videos, or even algorithms in the testing phase.

**III) Volume of Merges:** Projects with an equal or greater number of 100 merges were selected. Projects with a low number of merges might not present a sufficient number of conflicting merges to be analyzed in the subsequent stages of this work, while projects with a very large number of merges would make manual analysis (Stage 3) unfeasible.

[2]`https://octoverse.github.com/#top-languages-over-the-years`
[3]`https://survey.stackoverflow.co/2023/`
[4]`https://www.tiobe.com/tiobe-index/`

**Table 2.** Comments according to each programming language

| Programming Language | Line | Block |
|---|---|---|
| C | // | /* */ |
| C# | // | /* */ |
| C++ | // | /* */ |
| Java | // | /* */ |
| Javascript | // | /* */ |
| PHP | # | /* */ |
| Python | # | “ ” |
| Ruby | # | =begin ... =end |

**IV) Availability of Documentation:** Software projects that present wikis or some form of documentation are more suitable for analysis. On the other hand, projects with little documentation, without code comments or task annotations, become more difficult to understand during an exploratory analysis and, in some cases, make it impossible to apply approaches that use these artifacts as objects of analysis. It is essential to have adequate documentation to allow for a more effective and comprehensive analysis of these projects.

**V) Balance in Merge Distribution:** To ensure that the quantity of merges did not create a significant disparity among projects, a criterion of balanced quantity of merges per software project was applied. For this, it was established that the number of merges per project could not exceed 10% of the total merges analyzed in all selected projects. For example, the Osu project is the repository with the highest number of merges analyzed, totaling 16,658. However, it represents only 6.11% of the final set of instances, totaling 272,474 merges. This criterion ensures a balanced approach when analyzing merge data in different projects, allowing for a fair and representative analysis.

After applying criterion (I), initially 500 projects were selected. With the use of criteria (II), (III), and (IV), the number of selected projects was reduced to 280. To ensure a balanced set of projects (criterion V), in terms of the number of merges analyzed per project, ten projects per programming language were randomly chosen. Thus, in step (c), individual searches of the repositories on GitHub were conducted to collect general information such as project description, latest version, programming language, and number of contributors, and subsequently clone them locally.

**2) Data Extraction from Repositories:** After project selection, the official repositories were cloned. This stage relied on the support of a Java TDCA Tool developed by the authors, step (d), for the analysis of the cloned repositories.

The Java TDCA Tool, available on GitHub, analyzes the version history of a repository and identifies which merges in the project have conflicts, performing analyses on the code snippets of these merges. In the code snippets, considering that these code comments can be automatically generated by the development environments themselves and completed by the developers, the tool performs a textual search of the comments that are preceded by the character +, indicating that this was a code that was added (in contrast to code preceded by the character -, indicating that this code was removed), and then by the opening characters of code comment blocks of each programming language, as shown in Table 2.

Once a code comment indicating TD is identified, the hashes (identifiers) of the merges, date and time, number of code lines, and all task annotations highlighted in Related Works, Table 3, and Table 6, are stored and exported to a .csv file, step (e). The Java TDCA Tool also saves all code comments from the project in a separate text file, to be analyzed later as a way to complement the information extracted from the code snippets of the conflict resolution merges, providing more detailed information about the context in which the comments were written, step (f).

**Table 3.** Comments found in related works

| Comment | Related Work |
|---|---|
| FIXME | Huang et al. (2018); de Freitas Farias et al. (2015); Maldonado and Shihab (2015); Ren et al. (2019) |
| TODO | Huang et al. (2018); de Freitas Farias et al. (2015); Maldonado and Shihab (2015); Ren et al. (2019) |
| XXX | Huang et al. (2018); de Freitas Farias et al. (2015); Maldonado and Shihab (2015); Ren et al. (2019) |
| IMPLEMENT LATER | Huang et al. (2018); Ren et al. (2019) |
| BROKEN | Huang et al. (2018) |
| NOTE | Huang et al. (2018) |
| HACK | Huang et al. (2018) |
| BUG | de Freitas Farias et al. (2015) |

Sophisticated techniques for analyzing code comments, such as those presented in Related Works, tend to consider more the context of comment construction rather than just focusing on their generation. This approach can, in itself, highlight the creation or maintenance of TD.

In summary, if there are indications that merge conflict resolution is influencing the creation and maintenance of TD, the members of the development team and project managers can establish strategies to avoid its introduction and pay off the debt within the merge conflict resolution process. This can contribute to the management of TD in software projects, since merge conflicts are inevitable in software projects and directly impact code quality and team efficiency. Task annotations that may highlight the introduction of TD are identified in step (g).

**3) Analysis of code comments in files:** In step (h), the analysis of code comments in the resolution of the merge conflict is performed based on the comments made by the developers. The results of this analysis are presented in the format of Table 4, which demonstrates how code comments appear after extraction and how they can indicate the presence of TD. Additionally, other aspects can be observed, such as the type of language used and how the team organizes itself to highlight tasks.

Table 3 lists the task annotations present in the code comments mentioned in the findings of Related Works. It is important to note that comments may vary, depending on decisions by the development team or even informal agreements among team members, which are rarely documented, resulting in project-to-project variations.

**4) Identification of TD:** The identification of TD was based on the relationship of code comments found in Step 3 with which stage of development the comment is inserted, according to a classification proposed by Li et al. (2015), where

**Table 4.** Example of TD found after extraction

```
// TODO: Do we want a message even when not unsaved?
// Easy to hit Ctrl+W via muscle memory right now...
// } else {
// return Strings.UNLOAD_NO_UNSAVED;
// }
```

**Table 5.** Description of the criteria presented in Step 4

| Criterion | Description |
|---|---|
| C1 | Express that intervention in the code will be necessary. |
| C2 | Expressly indicates error or malfunctioning of the code. |
| C3 | Comments used only to inform other developers. |
| C4 | Express that there was no better way to implement and needs better testing. |

**Table 6.** Criteria for identifying TD

| Criteria | Comment | TD |
|---|---|---|
| C1 | TODO, WTF, HACK, UNDONE, REVIEW, IMPLEMENT | Code |
| C2 | FIXME, ASAP, BUG, ERROR, BROKEN | Defect |
| C3 | XXX, DONE, NOTE, ISSUE | Documentation |
| C4 | GLITCH, REMOVE | Test |

10 types of TD are classified: (1) requirement, (2) architecture, (3) design, (4) code, (5) testing, (6) build, (7) documentation, (8) infrastructure, (9) versioning, (10) defects (bugs), step (i) of the adopted methodology.

Although a classification with 10 possible types of TD has been adopted, this does not imply that all these types will be present in all projects. Software projects may exhibit a variety of types of TD documented in their source code, and some of these types may even go unnoticed by the team, known as unadmitted TD. The presence and recognition of these types of TD may vary from project to project, and detailed analysis is necessary to accurately identify them.

The criteria used to associate task annotations with types of TD are detailed in Table 5. By relating these criteria to the task annotations listed in Table 3 and the established types of TD, we obtain the criterion-comment-debt type relationship presented in Table 6. This relationship is essential to identify potential indications of TD within the comments, which are the primary focus of this study. However, semantic analysis of comment content is beyond the scope of this research, as the study does not aim to address the creation or maintenance of TD. Instead, it focuses on the implications of the comments themselves. Conducting such a semantic analysis would require a more robust approach, rendering the manual examination of multiple projects impractical.

## 4.2 Evaluation of the Computational Model

This section presents the concepts and metrics involved in evaluating the effectiveness of the developed computational model to classify TD in software repositories. The evaluation focuses on the model's ability to accurately identify and categorize TD types based on task annotations extracted from merge conflict resolution comments. By applying metrics such as accuracy, precision, recall, and F1-score, the study aims to validate the reliability and generalizability of the model across a variety of software projects. Furthermore, the results are analyzed to assess the model's performance in real-world scenarios, highlighting its strengths and potential areas for improvement. This evaluation not only demonstrates the usefulness of the model but also provides insights into its application to enhance TD management practices.

The effectiveness of the computational model in classifying TD was evaluated using standard performance metrics, and the computational model was validated using a labeled dataset of task annotations and their corresponding TD types. Metrics applied included:

- **Accuracy:** Proportion of correctly classified instances among all instances.
- **Precision:** Proportion of correctly classified positive instances among all instances classified as positive.
- **Recall:** Proportion of actual positive instances correctly classified by the model.
- **F1-score:** Harmonic mean of precision and recall, providing a balance between the two.

To address the issues outlined at the beginning of this section, only 7 projects out of the 66 analyzed in Part 1 (step h) of this methodology were selected for further evaluation of the computational model. This limitation was necessary due to the inclusion of a manual analysis step, which required reviewing all merges from the repositories of the chosen projects. Given the time-consuming nature of this process and the inherent constraints of manual analysis, it was essential to restrict the number of projects to ensure a comprehensive evaluation without relying on automated tools. The selected projects were those with the highest number of merge conflicts for each programming language. This approach ensures that the analysis includes a diverse and substantial dataset, offering meaningful insights that might not emerge from projects with fewer or less varied merge conflicts.

This analysis involved taking the classification results generated by the TDCA Tool and manually verifying whether the merges identified as technical debt truly corresponded to instances of technical debt. This manual validation enabled the creation of a confusion matrix to evaluate the accuracy of the tool's analysis (step j). The correctness of the tool's classification was assessed based on the predefined criteria and the number of instances analyzed, providing a clear measure of its performance. The model achieved an accuracy of 91%, precision of 89%, recall of 87%, and F1-score of 88%, demonstrating its reliability in identifying and classifying TD across diverse repositories.

## 4.3 Overview

The combined methodologies for identifying TD and evaluating the computational model provided a comprehensive understanding of the impact of merge conflict resolution on TD. These methods also validated the utility of the TDCA Tool and the computational model in analyzing real-world software repositories.

## 5 Results

Although 100 projects were selected, it was not possible to obtain results in all of them, either due to a lack of task annotations identified in the source code or the absence of indications of TD in the code comments. The exploratory analysis was successful in 66 projects, totaling more than 272,474 merges, with 14,776 conflict merges and a total of 841,282 comments analyzed. Table 13 presents general information about each analyzed project.

**Table 7.** Consolidated results by programming language

| Language | Code | Defect | Documentation | Test | Total |
|---|---|---|---|---|---|
| PHP | 530 | 276 | 289 | 150 | 1,245 |
| Javascript | 532 | 124 | 67 | 0 | 723 |
| C# | 294 | 167 | 177 | 10 | 648 |
| Java | 148 | 31 | 0 | 0 | 179 |
| C++ | 116 | 33 | 2 | 0 | 151 |
| C | 95 | 3 | 23 | 0 | 121 |
| Python | 16 | 8 | 3 | 0 | 27 |
| Ruby | 3 | 1 | 0 | 0 | 4 |
| **Total** | 1,734 | 643 | 561 | 160 | 3,098 |

It was possible to identify the presence of code comments that may indicate the creation or maintenance of TD in 40 projects, which corresponds to 60.61% of the total analyzed projects. About the other projects, in 7 of them, it was not possible to identify the existence of merge conflicts, while in another 19 projects, it was not possible to identify the creation or maintenance of TD through code comments. The consolidated results are presented in Table 7, highlighting the relationship between the programming language and the type of TD identified.

Among the programming languages of the selected projects, those that presented the highest indicators of TD in their comments were, respectively: PHP (1,245), JavaScript (723), C# (648), followed by Java (179), C++ (151), C (121), Python (27), and Ruby (1). The number of occurrences of test TD indicators in the PHP language does not seem to be intrinsically linked to the language itself, but rather to the specific project called Bolt (Table 14), which holds 150 out of the 160 test indicators for the language, in just 3 projects. Excluding the influence of this project, the incidence of this type of debt would be significantly lower.

Analyzing Table 7, it becomes evident that code TD is more prevalent in code comments, showing variation compared to instances of defect and documentation TD. This diversity of occurrences reflects the development scenario of projects in their particular context, where the team adopts specific task annotations to address different types of TD. For example, the `ISSUE` task annotation, supported on GitHub, allows the identification of issues to be resolved in the repository, similar to the `TODO` annotation, which can be generated by modern development environments.

In this sense, tools that support the identification, classification, prioritization, and removal of TD using code comments can be built. Although approaches that use more complex techniques for identifying TD perform well in identifying TD, they distance themselves from the practical aspect of the TD issue, which is to avoid its introduction, maintenance, and promote payment. Task annotations, in this sense, can simply highlight the establishment of TD in software projects. Thus, project managers and developers can focus on resolving actions without the need for more complex analyses of TD.

**RQ1 – Which task annotations are most associated**

**Table 8.** Task annotations found in selected projects

| Task Annotation | Comments |
|---|---|
| TODO | 1,518 |
| ERROR | 362 |
| NOTE | 331 |
| DONE | 232 |
| REVIEW | 218 |
| REMOVE | 160 |
| BUG | 101 |
| FIXME | 78 |
| XXX | 58 |
| FUTURE | 39 |
| ISSUE | 22 |
| BROKEN | 20 |
| HACK | 18 |

**with the project's TD?**

Although previous studies already highlight the association between code comments and TD, task annotations indicate a more specific aspect of code comments as they highlight the specific context in which code comments are written by the development team. As Table 8 shows, it can be observed that the task annotation that appears most frequently is `TODO` (1,518), followed by `ERROR` (362), `NOTE` (331), `DONE` (232), `REVIEW` (218), `REMOVE` (160), `BUG` (101), `FIXME` (78), `XXX` (58), `FUTURE` (39), `ISSUE` (22), `BROKEN` (20), and `HACK` (18).

This pattern of predominance of the `TODO` task annotation can be observed in the studies by de Freitas Farias et al. (2015) and Huang et al. (2018). This can be explained by two reasons: among the task annotations studied, this is the simplest to highlight a future task (SATD), and this annotation can be exclusively automatically generated by a series of modern development environments such as Visual Studio, IntelliJ IDEA, Eclipse, and NetBeans. Additionally, there are other tools like bots (Mohayeji et al., 2022), which also assist in writing and removing the `TODO` annotation, unlike other types of task annotations.

A hypothesis that can be investigated in the future is the relationship between the incidence of a specific task annotation and the support that this annotation receives. If there are many technological mechanisms seeking to support a particular annotation, it may be widely adopted by a large number of developers. This investigation can provide valuable insights into how tools and technological support influence the adoption of practices and task annotations by developers.

**RQ2 – What types of TD are most common in merge conflicts?**

Four types of TD were identified in the code snippets: code, documentation, defects, and testing debt, as demonstrated in Table 9. Although this work is using a theoretical framework of 10 types of TD (de Freitas Farias et al., 2015; Ren et al., 2019), this may occur because code comments are not the only way to document TD. Development team members may use different ways to document this type of occurrence.

The types of TD most frequently associated with merge conflicts are code debt (1,754), documentation debt (643), defect debt (561), and testing debt (160), as presented in Table 9. These indicators of TD show that in the analyzed projects, code debt is most evident in the source code itself, which can also be observed in the studies by de Freitas Farias et al. (2015) and Maldonado and Shihab (2015).

**Table 9.** Types of TD found in selected projects

| TD | Quantity |
|---|---|
| Code | 1,754 |
| Documentation | 643 |
| Defect | 561 |
| Test | 160 |

The documentation of other types of debt can be explained as a practice adopted by development teams when highlighting such tasks, an observation also made by Ren et al. (2019). Defect debt may be associated with how merge conflicts are resolved, as there are indications in the literature suggesting that merges with conflicts are more likely to introduce defects (Mahmoudi et al., 2019). The low incidence of testing debt and its association with only 3 out of the 66 analyzed projects indicates that the criteria and task annotations assigned to testing need to be reviewed to understand the reasons for this lack of occurrence in the remaining projects.

**RQ3 – Does the resolution of merge conflicts contribute to increasing or maintaining the project's TD?**

As shown in Table 14, out of the 66 projects analyzed, it was observed that 40 of them had at least one comment indicating TD. This number corresponds to 60.61% of all 100 projects, with a frequency occurring in at least 3,118 task annotations found in code comments resulting from merging conflicts resolution. Table 14 provides detailed results according to each TD indicator in the projects.

The 10 projects with the highest TD indicators were: Bolt (770), Brackets (443), ABP (433), Directus (345), Webpack (171), Elasticsearch (116), Cphalcon (80), Meteor (80), Duplicati (68), and Darktable (61). Other projects had a total number of indicators of TD lower than 60. Projects that did not present TD indicators were not included in Table 14.

Based on the projects analyzed, it can be observed that, in 66 projects, TD was possibly introduced or maintained with an incidence of at least 3,098 times due to the resolution of merge conflicts. This approach can help project managers and developers to implement practices that seek to reduce the introduction of TD through the resolution of merge conflicts, and this scenario presented can contribute to the management of TD in the projects analyzed.

Understanding this relationship between merging conflicts and TD can help developers and project managers adopt more efficient strategies to mitigate the impacts of TD in software development, resulting in improved code quality delivered.

**RQ4 – How effective is the computational model created in classifying the TD of the selected projects?**

The effectiveness of a software model is measured according to its ability to adequately classify instances of the dataset in all its variety. Table 10 shows the result of applying the evaluation classification to 7 projects, representing each of the programming languages that were selected in the previous stages of this research. Table 11 presents the results in percentage terms, providing a more relative view of the model's performance in classifying TD across various projects.

The Bolt project showed a relatively high rate of False Positives, indicating that a significant percentage of cases classified as TD by the model were not true cases.

The Brackets project presented a good balance between False Positives and False Negatives, with a high rate of True Negatives, suggesting an effective identification of cases not related to TD.

The Elasticsearch project had a challenging performance, with significant percentages of False Negatives and False Positives, indicating that the model might be missing real cases of TD and incorrectly identifying other cases.

The Sentry project showed a relatively high rate of False Negatives, suggesting that the model might be underestimating the presence of TD in this specific project.

The Spree project achieved perfect classification, with no errors of False Negatives or False Positives. This might indicate that this project does not present TD or that the model is highly tailored to this specific case.

The Darktable project showed a reasonable balance between False Negatives and False Positives, with a relatively high rate of True Positives.

The Dupllicati project had an average performance, with percentages of False Negatives and False Positives indicating room for improvement in classifying TD.

These analyses of the projects guide future improvements in the created model, aiming for a more precise and comprehensive approach to identifying TD in various development scenarios and covering a wider variety of projects.

**Table 10.** Performance of the TD classification

| Projects | FN | FP | TN | TP | Total |
|---|---|---|---|---|---|
| Bolt | 58 | 162 | 336 | 214 | 770 |
| Brackets | 25 | 16 | 330 | 72 | 443 |
| Elasticsearch | 16 | 25 | 47 | 28 | 116 |
| Sentry | 3 | 2 | 7 | 8 | 20 |
| Spree | 0 | 0 | 2 | 2 | 4 |
| Darktable | 9 | 8 | 22 | 29 | 68 |
| Dupllicati | 5 | 6 | 27 | 23 | 61 |

**Table 11.** Percentage performance of the TD classification

| Projects | FN | FP | TN | TP | Total |
|---|---|---|---|---|---|
| Bolt | 7.53 | 21.03 | 43.63 | 27.79 | 100 |
| Brackets | 5.64 | 3.61 | 74.49 | 16.25 | 100 |
| Elasticsearch | 13.79 | 21.55 | 40.51 | 24.31 | 100 |
| Sentry | 15 | 10 | 35 | 40 | 100 |
| Spree | 0 | 0 | 50 | 50 | 100 |
| Darktable | 13.23 | 11.76 | 32.35 | 42.64 | 100 |
| Dupllicati | 8.19 | 9.83 | 44.26 | 37.70 | 100 |

**RQ5 – How did the TDCA Tool adapt to different task annotations in the comments of files resolving merge conflicts?**

The Java TDCA Tool was designed, examined, and evaluated based on the strategies and concepts introduced in this study. The implemented classification model was subjected to manual analysis in seven development projects, where the results indicated the presence of TD through code comments. Subsequently, the evaluation metrics described in this work were applied to assess the model's performance in classifying TD.

**Table 12.** Result of evaluation metrics for classifying TD

| Projects | Accuracy | Precision | Recall | F1-score |
|---|---|---|---|---|
| Bolt | 0.714 | 0.569 | 0.786 | 0.660 |
| Brackets | 0.907 | 0.818 | 0.742 | 0.777 |
| Elasticsearch | 0.646 | 0.528 | 0.373 | 0.437 |
| Sentry | 0.75 | 0.8 | 0.727 | 0.761 |
| Spree | 1 | 1 | 1 | 1 |
| Darktable | 0.75 | 0.783 | 0.763 | 0.763 |
| Dupllicati | 0.819 | 0.793 | 0.821 | 0.806 |

Medium and large-scale projects typically have a greater variety of task annotations, documented more extensively in the source code, which is crucial for more specific project analyses. It is suggested that future studies expand the set of keywords for a more in-depth investigation of this issue. Additionally, a discrepancy was observed between manual and automated analysis of TD. The evaluation metrics indicate that precision and coverage performed worse compared to the accuracy of the projects. This implies that the tool was able to correctly classify in most cases, but was not particularly precise in this classification.

The TD identification strategy employed in this study is simple and does not depend on complex project analyses, suggesting that the model can be improved for more accurate and comprehensive analyses. Overall, when analyzing the F1-score, the results are optimistic, but the model should be reviewed and improved for a deeper understanding of the studied projects.

**RQ6 – How did the TD classification perform in the context of the selected projects?**

The results of the evaluation metrics for the TD classification for the 7 manually analyzed projects are presented in Table 12. Each metric provides a unique view of the implemented model's performance, and a deeper analysis is necessary to fully understand the nuances associated with each project. Figures 2, 3, 4, and 5 demonstrate how the evaluation of the projects was conducted in general concerning each evaluation metric, which outlines the metrics and the benchmark that guided this experiment.

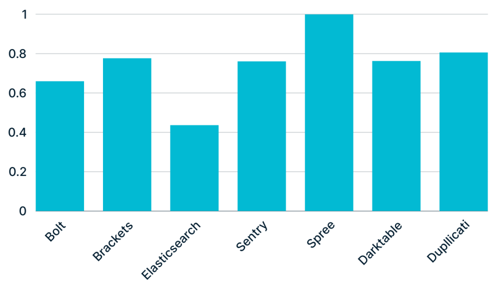

**Figure 2.** F1-score of evaluated projects

The Bolt project shows an accuracy of 0.714, indicating that the model correctly classified approximately 71.4% of the cases. The precision of 0.569 suggests that about 56.9% of the instances classified as TD were indeed true cases of TD. The recall of 0.786 indicates that the model adequately

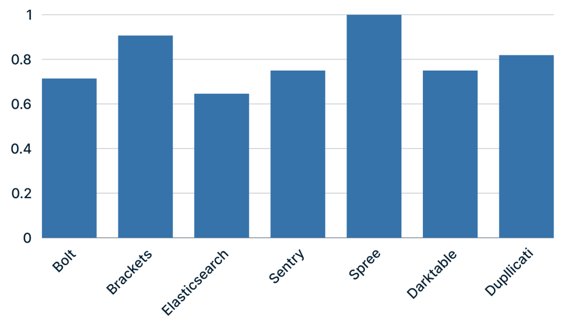

**Figure 3.** Accuracy of evaluated projects

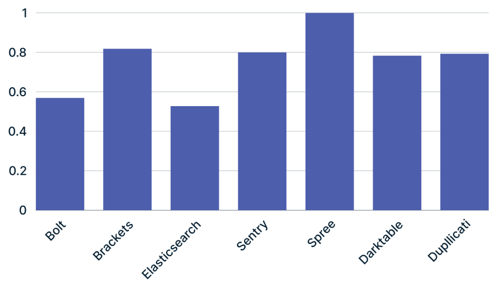

**Figure 4.** Precision of evaluated projects

captured 78.6% of all real cases of TD, while the F1-score of 0.660 balances precision and recall.

The Brackets project demonstrates remarkable performance, with an accuracy of 0.907, indicating a high rate of correct classification. The precision of 0.818 suggests that the model maintained a high rate of accurately identifying cases of TD. However, it is important to note that the recall of 0.742 indicates that not all cases were covered by the model. The F1-score of 0.777 balances these metrics of precision and recall.

For the Spree project, all metrics reached the maximum value of 1, indicating perfect classification. This result raises a question about the possibility of the model being overfitted to this specific project or whether the project genuinely does not present TD.

The Elasticsearch project achieved lower accuracy, suggesting that the model had difficulties correctly classifying the cases. The recall of 0.373 indicates that the model captured only a limited portion of the real cases of TD, highlighting the need for improvements.

In summary, the analysis of these metrics highlights the variability in the model's performance across different projects. While some projects achieve impressive results, others present challenges that may require adjustments in the approach to TD classification. This diversity reinforces the importance of careful analysis and continuous adjustments to optimize the model for different software development contexts.

The results of this study suggest that analyzing code comments for merge conflicts can be a promising technique for classifying TD in medium and large-scale projects. However, it is important to conduct further research and other implementations to gain a deeper understanding of the factors in-

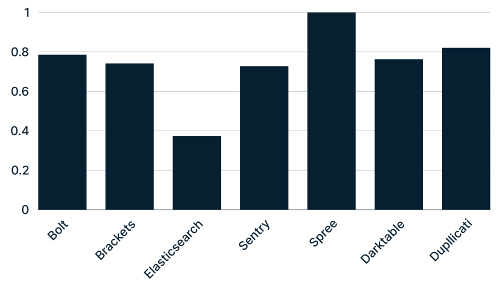

**Figure 5.** Recall of evaluated projects

fluencing the performance of this technique.

## 6 Discussion

The findings of this study provide valuable insights for both researchers and practitioners in the field of software engineering, particularly regarding the management of TD and the implications of merge conflict resolution.

### 6.1 Implications for Researchers

The study contributes to the growing body of knowledge on TD by offering a detailed analysis of task annotations and their relationship with different types of debt. The use of the TDCA Tool and the computational model demonstrated the feasibility of automating the identification and classification of TD, which can be further explored and extended in future research. Specifically, researchers can:

- Investigate how task annotations evolve and their long-term impact on software quality.
- Explore the use of semantic analysis techniques to capture nuanced or implicit TD that may not be evident through task annotations alone.
- Analyze the generalizability of the findings to projects written in less popular programming languages or with fewer stars, addressing external validity concerns.
- Develop more robust frameworks for mapping task annotations to TD types, incorporating context-specific factors.

The study also highlights areas where existing methodologies can be improved, such as enhancing the accuracy of automated tools and integrating contextual information into classification models.

### 6.2 Implications for Practitioners

For practitioners, the results underline the importance of effective management practices for TD during merge conflict resolution. Key takeaways include:

- **Task Annotations as Indicators:** Practitioners can use task annotations, such as `TODO` and `FIXME`, as actionable indicators of TD, prompting timely interventions to mitigate its accumulation.

- **Impact of Merge Conflicts:** The findings suggest that merge conflict resolution is a critical stage where TD is often introduced or maintained. Teams should implement strategies to minimize their impact, such as code reviews and conflict resolution guidelines.
- **Tool Support:** Tools like the TDCA Tool can assist teams in identifying and tracking TD, providing a systematic approach to debt management. Practitioners should consider integrating such tools into their development workflows.
- **Documentation Practices:** The availability of adequate documentation and well-structured comments can significantly aid in identifying and addressing TD. Teams should prioritize documentation as part of their TD management strategy.

By addressing TD proactively, practitioners can improve software quality, reduce maintenance costs, and enhance team efficiency. This study provides a foundation for integrating TD management into standard development practices, particularly in scenarios involving frequent merge conflicts.

## 7 Threads to Validity

Threats to the validity of this study are related to possible implementation errors and personal bias in the manual analysis of code comments. To mitigate implementation errors, the Java TDCA Tool was carefully reviewed, and a manual analysis step of code comments was included in the methodology. In total, 841,282 comments were analyzed.

To ensure the quality and quantity of data, 100 open-source projects were selected with significant variations in the number of comments and merges, as well as in the characteristics of the comments.

It is important to note that, in this work, a medium-sized dataset was used, which allowed for the manual analysis step of code comments and understanding the capabilities and limitations of the approach used. In the future, this threat could be reduced by expanding the approach to larger software projects and including a greater number of projects to be analyzed.

## 8 Conclusion

In this study, the relationship between TD and merge conflict resolution was investigated. The findings indicate that resolving merge conflicts contributes to the increase and maintenance of TD in code, documentation, defects, and testing. It was identified that certain patterns of code comments are applicable across various projects of different sizes and languages, such as the `TODO` task annotation, a practice established in code comment documentation and automatically generated by various development environments and other tools.

This type of exploratory and experimental analysis is influenced by how code comments are written by the development team. In less robust projects, the development team may not have a formal agreement on the use of these annotations. The lack of documentation in code comments or changes in task annotation writing patterns can also influence the analysis results, as highlighted by Ren et al. (2019). Different results would be expected in projects with well-established agreements on task annotation creation.

The TD indicators within the context of merge conflicts presented here do not fully correspond to the project's overall TD, but they may serve as a strategy for managers and team members to minimize the introduction of TD in software projects.

Future work could conduct a more in-depth study, addressing a larger number of task annotations and investigating a broader range of code comments, aiming to explore in detail how these annotations are used in their specific context of each project.

Furthermore, future studies could seek evidence of other types of TD in the context of merge conflict resolution, providing a deeper understanding of the topic. Additionally, investigating the cause-and-effect relationship between the incidence of a specific task annotation and the technological support that annotation receives would be crucial to uncovering more details about how tools and technological support influence the adoption of practices and annotations by developers. These additional research efforts could further enrich the knowledge about TD management in software projects and contribute to the enhancement of practices and approaches adopted by development teams.

## References

Ahmed, I., Brindescu, C., Mannan, U. A., Jensen, C., and Sarma, A. (2017). An Empirical Examination of the Relationship between Code Smells and Merge Conflicts. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 58–67.

Bavota, G. and Russo, B. (2016). A large-scale empirical study on self-admitted technical debt. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 315–326, Austin Texas. ACM.

Borges, H. and Valente, M. T. (2018). What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software*, 146:112–129. arXiv:1811.07643 [cs].

Cunningham, W. (1992). The WyCash Portfolio Management System.

de Freitas Farias, M. A., de Mendonca Neto, M. G., da Silva, A. B., and Spinola, R. O. (2015). A Contextualized Vocabulary Model for identifying technical debt on code comments. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, pages 25–32, Bremen, Germany. IEEE.

Hattori, L. P. and Lanza, M. (2008). On the nature of commits. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops*, pages 63–71. ISSN: 2151-0849.

Huang, Q., Shihab, E., Xia, X., Lo, D., and Li, S. (2018). Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering*, 23(1):418–451.

Kruchten, P., Nord, R. L., and Ozkaya, I. (2012). Technical Debt: From Metaphor to Theory and Practice. *IEEE Software*, 29(6):18–21. Conference Name: IEEE Software.

Kruchten, P., Nord, R. L., Ozkaya, I., and Falessi, D. (2013). Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt. *ACM SIGSOFT Software Engineering Notes*, 38(5):51–54.

Li, Z., Avgeriou, P., and Liang, P. (2015). A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220.

Lim, E., Taksande, N., and Seaman, C. (2012). A Balancing Act: What Software Practitioners Have to Say about Technical Debt. *IEEE Software*, 29(6):22–27. Conference Name: IEEE Software.

Mahmoudi, M., Nadi, S., and Tsantalis, N. (2019). Are Refactorings to Blame? An Empirical Study of Refactorings in Merge Conflicts. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 151–162. ISSN: 1534-5351.

Maldonado, E. C., Valente, M. T., and Cedrim, D. (2017a). On the use of self-admitted technical debt in open source projects. *Empirical Software Engineering*, 22(4):1658–1706. Examines the presence and impact of self-admitted technical debt in open source software.

Maldonado, E. D. S., Abdalkareem, R., Shihab, E., and Serebrenik, A. (2017b). An Empirical Study on the Removal of Self-Admitted Technical Debt. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 238–248, Shanghai. IEEE.

Maldonado, E. d. S. and Shihab, E. (2015). Detecting and quantifying different types of self-admitted technical Debt. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, pages 9–15, Bremen, Germany. IEEE.

Melo de Araújo, M. H., Costa, C., and Fontão, A. (2023). Analysis of the technical debt of software projects based on merge code comments. In *Proceedings of the 17th Brazilian Symposium on Software Components, Architectures, and Reuse*, pages 21–30.

Mohayeji, H., Ebert, F., Arts, E., Constantinou, E., and Serebrenik, A. (2022). On the adoption of a TODO bot on GitHub: a preliminary study. In *Proceedings of the Fourth International Workshop on Bots in Software Engineering*, pages 23–27, Pittsburgh Pennsylvania. ACM.

Nieminen, A. (2012). Real-time collaborative resolving of merge conflicts. In *8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pages 540–543.

Potdar, A. and Shihab, E. (2014). An Exploratory Study on Self-Admitted Technical Debt. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 91–100. ISSN: 1063-6773.

Ren, X., Xing, Z., Xia, X., Lo, D., Wang, X., and Grundy, J. (2019). Neural Network-based Detection of Self-Admitted Technical Debt: From Performance to Explainability. *ACM Transactions on Software Engineering and Methodology*, 28(3):1–45.

Rios, N., Spínola, R., and Mendonça, M. (2021). Organização de um Conjunto de Descobertas Experimentais sobre Causas e Efeitos da Dívida Técnica através de uma Família de Surveys Globalmente Distribuída. In *Anais Estendidos do Congresso Brasileiro de Software: Teoria e Prática (CBSoft)*, pages 80–94. SBC. ISSN: 0000-0000.

Seaman, C., Guo, Y., Zazworka, N., Shull, F., Izurieta, C., Cai, Y., and Vetrò, A. (2012). Using technical debt data in decision making: Potential decision approaches. In *2012 Third International Workshop on Managing Technical Debt (MTD)*, pages 45–48.

Storey, M.-A., Ryall, J., Bull, R. I., Myers, D., and Singer, J. (2008). TODO or to bug: exploring how task annotations play a role in the work practices of software developers. In *Proceedings of the 13th international conference on Software engineering - ICSE '08*, page 251, Leipzig, Germany. ACM Press.

Tsantalis, N., Yoshida, N., Wang, S., Kamei, Y., and Kashiwa, K. (2018). Identifying refactoring opportunities during merge conflict resolution. *Journal of Systems and Software*, 146:124–142. Explores the role of refactoring during merge conflict resolution.

Wang, H. et al. (2024). Just-in-time TODO-Missed commits detection. *IEEE Transactions on Software Engineering*.

Zhao, W., Yin, G., Lo, D., and Li, S. (2017). Automated detection of merge conflicts in version control systems. *IEEE Transactions on Software Engineering*, 43(8):659–683. Discusses the challenges of merge conflicts and their relationship with code quality.

**Table 13.** Detailed information about the analyzed software projects

| TM - Total Merges | | MC - Merges with Conflict | | | CA - Comments Analyzed | | |
|---|---|---|---|---|---|---|---|
| **C** | **TM** | **MC** | **CA** | **C#** | **TM** | **MC** | **CA** |
| Borg | 3,080 | 100 | 5,773 | ABP | 7,436 | 465 | 270,914 |
| Darktable | 4,031 | 223 | 118,899 | Avalonia | 7,554 | 335 | 7,711 |
| JohnTheRipper | 1,719 | 147 | 9,551 | Duplicati | 1,647 | 134 | 27,544 |
| Libgit2 | 3,533 | 53 | 5,211 | Efcore | 2,038 | 592 | 81,884 |
| Mruby | 3,942 | 73 | 43,100 | EventStore | 2,243 | 78 | 7,733 |
| Qemu | 6,257 | 115 | 1,200 | Jellyfin | 5,635 | 342 | 14,104 |
| Redis | 1,554 | 69 | 2,960 | OpenRA | 4,922 | 43 | 1,364 |
| RetroArch | 7,315 | 157 | 7,621 | Osu | 16,658 | 0 | 0 |
| SystemD | 6,178 | 0 | 0 | PTVS | 2,826 | 149 | 6,575 |
| Tmux | 1,643 | 132 | 1,490 | ReactiveUI | 1,743 | 161 | 6,710 |
| **C++** | **TM** | **MC** | **CA** | **Java** | **TM** | **MC** | **CA** |
| Caffe | 3,723 | 337 | 22,200 | Dbeaver | 5,160 | 112 | 8,340 |
| Cosmos | 3,315 | 91 | 3,030 | Druid | 2,181 | 70 | 2,102 |
| Electron | 4,249 | 88 | 2,040 | Elasticsearch | 5,210 | 601 | 8,435 |
| Emscripten | 1,910 | 187 | 4,909 | Graal | 15,590 | 0 | 0 |
| Foundationb | 8,041 | 417 | 0 | Jenkins | 5,142 | 580 | 8,846 |
| MongoDB | 5,324 | 438 | 39,583 | Libgdx | 2,712 | 103 | 1,800 |
| Protobuf | 3,084 | 141 | 18,253 | Pinpoint | 1,968 | 0 | 0 |
| qBittorrent | 2,381 | 11 | 570 | Vertx | 1,254 | 111 | 0 |
| Swoole | 1,896 | 174 | 26,184 | RxJava | 1,591 | 0 | 0 |
| **Javascript** | **TM** | **MC** | **CA** | **PHP** | **TM** | **MC** | **CA** |
| Atom | 4,561 | 379 | 0 | Bolt | 5,101 | 400 | 1,792 |
| Babel | 1,324 | 117 | 0 | Composer | 2,223 | 198 | 602 |
| Bootstrap | 4,368 | 487 | 0 | Console | 1,525 | 279 | 703 |
| Brackets | 6,037 | 672 | 1,863 | Cphalcon | 3,243 | 223 | 38,205 |
| Ghost | 2,975 | 36 | 223 | Directus | 2,158 | 227 | 3,886 |
| Meteor | 2,303 | 369 | 2,435 | Joomla | 6,696 | 782 | 0 |
| React | 2,664 | 44 | 164 | Laravel | 1,496 | 97 | 239 |
| Serverless | 2,938 | 162 | 0 | PHPUnit | 3,370 | 842 | 4,352 |
| Svelte | 1,520 | 90 | 180 | Timber | 1,637 | 163 | 843 |
| Webpack | 2,852 | 263 | 3,987 | Yii2 | 3,777 | 0 | 0 |
| **Python** | **TM** | **MC** | **CA** | **Ruby** | **TM** | **MC** | **CA** |
| Bokeh | 3,154 | 360 | 951 | Chef | 6,427 | 140 | 997 |
| Certbot | 2,562 | 400 | 1,170 | Cucumber | 1,448 | 301 | 665 |
| Kivy | 2,856 | 74 | 346 | Fastlane | 2,512 | 94 | 271 |
| Pandas | 3,199 | 119 | 284 | Fog | 2,800 | 120 | 470 |
| Pyramid | 3,098 | 121 | 230 | Jekyll | 2,077 | 104 | 211 |
| Scikit-learn | 1,421 | 298 | 4,794 | Puppet | 10,388 | 0 | 0 |
| Sentry | 2,225 | 200 | 1,068 | Spree | 2,635 | 115 | 2,048 |
| Spacy | 1,421 | 149 | 1,574 | Vagrant | 2,148 | 50 | 93 |

**Table 14.** Results according to project and type of TD

| Project | Code | Defect | Documentation | Test | Total |
|---|---|---|---|---|---|
| Bolt | 231 | 148 | 241 | 150 | 770 |
| Brackets | 363 | 62 | 18 | 0 | 443 |
| ABP | 106 | 148 | 175 | 4 | 433 |
| Directus | 253 | 92 | 0 | 0 | 345 |
| Webpack | 127 | 0 | 44 | 0 | 171 |
| Elastichsearch | 96 | 20 | 0 | 0 | 116 |
| Cphalcon | 34 | 33 | 48 | 0 | 115 |
| Meteor | 13 | 62 | 5 | 0 | 80 |
| Duplicati | 67 | 1 | 0 | 0 | 68 |
| Darktable | 44 | 0 | 17 | 0 | 61 |
| Efcore | 52 | 6 | 1 | 0 | 59 |
| RetroArch | 42 | 0 | 4 | 0 | 46 |
| Protobuf | 29 | 15 | 0 | 0 | 44 |
| Emscripten | 34 | 6 | 0 | 0 | 40 |
| MongoDB | 25 | 12 | 2 | 0 | 39 |
| EventStore | 22 | 9 | 0 | 6 | 37 |
| PTVS | 27 | 0 | 1 | 0 | 28 |
| Swoole | 27 | 0 | 0 | 0 | 27 |
| Pinpoint | 25 | 0 | 0 | 0 | 25 |
| Ghost | 21 | 0 | 0 | 0 | 21 |
| Sentry | 15 | 4 | 1 | 0 | 20 |
| ReactiveUI | 12 | 3 | 0 | 0 | 15 |
| Druid | 9 | 5 | 0 | 0 | 14 |
| Timber | 11 | 2 | 0 | 0 | 13 |
| Vertx | 9 | 2 | 0 | 0 | 11 |
| RxJava | 6 | 3 | 0 | 0 | 9 |
| OpenRA | 8 | 0 | 0 | 0 | 8 |
| React | 8 | 0 | 0 | 0 | 8 |
| Libgit2 | 5 | 1 | 0 | 0 | 6 |
| JohnTheRipper | 2 | 1 | 1 | 0 | 4 |
| Libdgx | 3 | 1 | 0 | 0 | 4 |
| Qemu | 2 | 1 | 1 | 0 | 4 |
| Spree | 3 | 1 | 0 | 0 | 4 |
| Certbot | 0 | 2 | 0 | 0 | 2 |
| Kivy | 1 | 0 | 1 | 0 | 2 |
| PHPUnit | 1 | 1 | 0 | 0 | 2 |
| Scikit-learn | 0 | 1 | 1 | 0 | 2 |
| Electron | 1 | 0 | 0 | 0 | 1 |
| Spacy | 0 | 1 | 0 | 0 | 1 |
| **Total** | 1,734 | 643 | 561 | 160 | 3,098 |