Self-Admitted Technical Debt in the Embedded Systems Industry

Self-Admitted Technical Debt in the Embedded Systems Industry: An Exploratory Case Study

Yikun Li, Mohamed Soliman, Paris Avgeriou, and Lou Somers

Abstract—Technical debt denotes shortcuts taken during software development, mostly for the sake of expedience. When such shortcuts are admitted explicitly by developers (e.g., writing a TODO/Fixme comment), they are termed as *Self-Admitted Technical Debt* or *SATD*. There has been a fair amount of work studying SATD management in Open Source projects, but SATD in industry is relatively unexplored. At the same time, there is no work focusing on developers' perspectives towards SATD and its management. To address this, we conducted an exploratory case study in cooperation with an industrial partner to study how they think of SATD and how they manage it. Specifically, we collected data by identifying and characterizing SATD in different sources (issues, source code comments, and commits) and carried out a series of interviews with 12 software practitioners. The results show: 1) the core characteristics of SATD in industrial projects; 2) developers' attitudes towards identified SATD and statistics; 3) triggers for practitioners to introduce and repay SATD; 4) relations between SATD in different sources; 5) practices used to manage SATD; 6) challenges and tooling ideas for SATD management.

Index Terms—Technical debt, self-admitted technical debt, mining software repositories, source code comment, issue tracking system, commit, empirical study

1 Introduction

Technical debt (TD) refers to compromising the long-term maintainability and evolvability of software systems by selecting sub-optimal solutions, in order to achieve short-term goals [1]. When software developers incur technical debt, they sometimes explicitly admit it; for example, software developers may write *TODO* or *Fixme* in a source code comment, indicating a sub-optimal solution in that part of the code. Potdar and Shihab [2] called these textual statements *Self-Admitted Technical Debt* (SATD). SATD can be found in several sources such as source code comments [2], issues in issue tracking systems [3], [4], and commit messages [5].

The SATD that can be identified in such sources is complementary to the technical debt that can be identified in source code through static analysis. This is because, certain types of technical debt cannot be identified by analyzing source code. For example, *partially implemented requirements* is a type of *requirement debt* [4] that can be identified from source code comments or issue tracking systems but not from running source code analysis tools: *"TODO: This class only has partial Undo support (basically just those members that had it as part of a previous implementation) [from Apache ArgoUML[1]]."* Therefore it is important to identify and further manage SATD, in addition to the more traditional approach of managing technical debt in source code.

There has been a fair amount of work investigating the identification [6], [7], measurement [8], prioritization [9], and repayment [10], [11] of SATD. However, to the best of our knowledge, all previous studies (but one, namely [5]) identified SATD in open-source projects; we actually know little about SATD in industrial projects. Moreover, none of the previous studies has surveyed software developers about SATD, in order to capture their perspectives towards SATD management, and tooling support for different sources. Without involving software developers to investigate SATD, researchers risk developing theories or approaches, which do not align with the needs and practices of software engineers.

To address these shortcomings, we conducted an exploratory case study in collaboration with an industrial partner to investigate how SATD is managed and how this can be supported. We collected data in two steps. First, we identified and characterized SATD in projects within that company from three sources: issues, source code comments, and commits. This step took place by using pre-trained machine learning models [12]. Second, we carried out a series of interviews with 12 software practitioners from that organization to understand their perception of what SATD really is, how it is managed, and how this management can be potentially improved. The contributions and main findings of this study are summarized as follows:

- Yikun Li, Mohamed Soliman, and Paris Avgeriou are with the Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence, University of Groningen, 9712 Groningen, CP, The Netherlands. E-mail: {yikun.li, m.a.m.soliman, p.avgeriou}@rug.nl.
- Lou Somers is with the Department of Mathematics and Computer Science, Eindhoven University of Technology, 5612 Eindhoven, AZ, The Netherlands. E-mail: l.j.a.m.somers@tue.nl.

Manuscript received 7 February 2022; revised 17 November 2022; accepted 18 November 2022. Date of publication 23 November 2022; date of current version 18 April 2023.
This work was supported by ITEA3 and RVO under Grant 17038 VISDOM (https://visdom-project.github.io/website).
(Corresponding author: Yikun Li.)
Recommended for acceptance by Y. Cai.
Digital Object Identifier no. 10.1109/TSE.2022.3224378

[1] https://github.com/argouml-tigris-org/argouml/blob/d5cd45cb4409c6f50747a3a2671219111b443c48/src/argouml-app/src/org/argouml/notation/NotationSettings.java#L108

- *Characterizing SATD in industrial projects.*The results indicate that most technical debt is admitted in issues, followed by source code comments and commit messages. Non-SATD issues take a significantly shorter time to close, compared to SATD issues.
- *Reporting developers' attitudes towards identified SATD and statistics.*Most interviewees acknowledged the identified SATD. However, they do need more information to assess the importance of individual SATD items.
- *Reporting relations between SATD from different sources.* We found that SATD in code comments and issues is referenced in the other sources, while SATD in commits is not referenced in other sources.
- *Reporting triggers on SATD introduction and repayment.* The results show developers have different reasons to introduce and pay back SATD, depending on the data source (code comments, issues, commits).
- *Reporting practices used to manage SATD.*We summarize and report practices that are used to assist in SATD prioritization and repayment.
- *Reporting tooling support for SATD management.*We report tool features that developers suggested as useful for SATD identification, traceability, prioritization, and repayment.

The rest of this paper is organized as follows. Section 2 discusses related work. The case study design is elaborated in Section 3. Section 4 presents the results, and Section 5 discusses the implications of these results on researchers and practitioners. Finally, threats to validity are evaluated in Section 6 and conclusions and future work are drawn in Section 7.

2 Related Work

To facilitate comparison to our work, we split the related work into two parts: work associated with SATD in Open-Source Software and work associated with SATD in industrial settings.

2.1 SATD in Open-Source Software

Potdar and Shihab [2] were the first to study SATD in source code comments. They analyzed four open-source projects and identified SATD in them. They found that 2.4% to 31% of source files contain SATD comments and only 26.3% to 63.5% of SATD are removed after introduction. Moreover, the results of Potdar and Shihab show that experienced developers tend to introduce more SATD compared to inexperienced developers. Building on this work, Maldonado and Shihab [13] focused on the types of SATD in open-source projects. They analyzed 33 K code comments from five projects and categorized SATD into five categories: design, requirement, defect, documentation, and test debt. The results indicated that design debt is the most common type of SATD, as 42% to 84% of classified SATD is design debt.

Subsequently, there was a significant focus on automatic SATD identification. Maldonado et al. [6] manually classified source code comments into different types of SATD from ten open-source projects and utilized the maximum entropy classifier to automatically identify design debt and requirement debt. Similarly, Huang et al. [14] used the feature selection method to select the most important features and adopted the ensemble learning technique to leverage different machine learning approaches to accurately identify SATD, again from source code comments. Furthermore, different machine learning approaches were applied to achieve higher predictive performance for SATD identification. Specifically, Ren et al. [7] proposed a Convolutional Neural Network-based approach to accurately identify SATD from source code comments. Wang et al. [15] proposed an attention-based neural network to automatically detect SATD. In addition to using source code comments, few studies focused on identifying SATD from other sources. Dai and Kruchten [3] manually analyzed 8 K issue tickets and used the Naive Bayes method to classify SATD issues and non-SATD issues. In our previous work [16], we examined 23 K issue sections and proposed a Convolutional Neural Network-based approach to identify SATD from issue tracking systems.

In addition to SATD identification, there has been work related to the measurement, prioritization, and repayment of SATD. Kamei et al. [8] explored ways to measure the interest of SATD and suggested using LOC and Fan-In measures. The results indicated that 42.2% to 44.2% of SATD incurs positive interest (i.e., technical debt costs more to repay in the future), while 8.1% to 13.8% of SATD incurs negative interest (i.e., technical debt costs less to pay back in the future). Mensah et al. [9] introduced a SATD prioritization scheme which consists of identification, examination, and rework effort estimation. The results showed that a rework effort of modifying 10 to 25 commented LOC per SATD source file is required for highly prioritized SATD tasks. Besides, Maldonado et al. [10] analyzed five open-source projects to investigate the repayment of SATD. The results indicated that most of SATD is removed eventually and the payback is mostly done by those that incurred the SATD in the first place. They also found that SATD lingers in the code for approximately 18 to 172 days. In a similar study, Zampetti et al. [11] looked into how SATD is resolved in five open-source projects. They found that between 20% to 50% of SATD comments are removed by accident, and 8% of SATD repayment is documented in commit messages.

Compared to all aforementioned studies, our study has the following differences: a) we utilized machine learning models to identify and characterize SATD; b) we performed this analysis on a number of different sources, instead of only one; c) we work in an industrial setting instead of open-source systems; d) we explored developers' perspectives towards both the nature of SATD and its management.

2.2 SATD in Industrial Settings

SATD in industrial settings is relatively unexplored; there is only one work that studied SATD in industrial settings and compared it with open-source settings [5]. Specifically, Zampetti et al. surveyed 52 industrial developers and 49 open-source project contributors. They focused on technical debt admitted in source code comments and found that technical debt annotation practices and the typical content of SATD comments are similar in industrial and open-source settings. Furthermore, the results showed that admitted technical debt in industrial projects is implicitly discouraged by the fear of taking on responsibilities. The results indicated that technical debt is also admitted in other sources, including, among others, commit messages, pull requests, and issue trackers.

In contrast to Zampetti et al. [5], who only investigated source code comments, we focus on analyzing SATD from multiple sources (i.e., source code comments, commit messages, and issue tracking systems). In addition, we use machine learning techniques to identify SATD from different sources in industrial settings, demonstrate the characteristics of SATD, and present the interviewees' attitudes towards the identified SATD and statistics. Finally, we also study the process of managing SATD, as well as how to improve it from the point of view of software practitioners.

3 Study Design

3.1 Objective and Research Questions

The goal of this study, formulated according to the Goal-Question-Metric [17] template is to "*analyze* self-admitted technical debt in source code comments, issue tracking systems, and commit messages *for the purpose of* understanding and improvement *with respect to* the nature and management process of self-admitted technical debt in practice *from the point of view of* software engineers *in the context of* the embedded systems industry." To be more precise, we aim at understanding both the nature of SATD per se and the process of managing it, as well as improving this process. Consequently, we formulate three main research questions (RQs) that are further refined into sub-questions. In Section 4 we will not answer the main RQs directly, but only indirectly through answering the sub-questions.

- *RQ1: What is the nature of SATD in industry?*
    - *RQ1.1:What are the types, amounts, resolution time, and sources of SATD items in industrial settings?*

      Rationale: There are significant differences between open-source projects and industrial projects concerning project management, tooling, as well as collaboration and communication [18], [19]. Thus, developers may admit technical debt differently in the two cases. Meanwhile, as mentioned in Section 2, all (but one, namely [5]) previous 'studies on SATD have focused on open-source projects. Thus, determining the types of SATD (e.g., requirements, design, code debt), amount of SATD (i.e., number and percentages of SATD items), and the sources of SATD (e.g., issue tracker or source code comments) in industrial projects, and comparing them with open-source projects could help researchers understand what SATD looks like in practice, and practitioners to better manage SATD in both cases.
    - *RQ1.2: How is automatically identified SATD regarded by professional software engineers?*

      Rationale: The identification of SATD can be automated, e.g., by using machine learning techniques [7], [16]. However, as far as software engineers are concerned, the identified SATD items could be obsolete, inaccurate, irrelevant, or inconsistent with the code. We aim at understanding how far software engineers consider that the SATD items are indeed important and relevant for their system. We also want to understand whether software engineers agree with the main statistics of the identified SATD (e.g., percentage of backlog items and the lifetime of SATD items). This can help us understand the strengths and weaknesses of automated SATD identification.
    - *RQ1.3: What are the relations between SATD in different sources?*

      Rationale: The different sources where SATD is documented (e.g., source code, issues, commits), are implicitly or explicitly related to each other. Thus, developers sometimes choose to document the same technical debt in more than one source. For example, when developers encounter SATD in code comments and they consider this debt as important, they might document it in issue tracking systems for more exposure and visibility. In these cases, we are interested in understanding the connections between the SATD items in these different sources. This can assist in improving the traceability of SATD in different sources.
- *RQ2: How is SATD being managed in industry?*
    - *RQ2.1: When is technical debt (not) admitted in source code comments, issue tracking systems, and commit messages?*

      *Rationale:*It is important to understand the reasons for documenting or not documenting technical debt in different sources. This can help researchers in coming up with guidelines and practices for SATD documentation. Furthermore, this could help develop tools to assist in documenting SATD.
    - *RQ2.2: What are the pros and cons of admitting technical debt in different sources?*

      Rationale: Each source has its advantages and disadvantages in terms of documenting technical debt. For example, technical debt that is admitted in code comments, allows developers reading those comments to also examine the problem in the adjacent code. On the downside, SATD in code comments has limited visibility for team leads and project managers and typically does not get added to the backlog. Understanding the pros and cons of different sources for documenting SATD could help developers make better use of different sources to document technical debt.
    - *RQ2.3: What are the triggers to pay back or not pay back SATD?*

      Rationale: Developers are more likely to pay back SATD in certain cases. We plan to investigate the developers' motivation both for repayment and for deciding to leave SATD in the system. This could help researchers understand the reasons for SATD repayment and develop a tool to assist practitioners in paying back TD.

- *RQ2.4: What practices are used to support SATD management in industrial settings?*

  Rationale: While, a number of studies have investigated TD management in industry, we know very little about managing self-admitted TD. This RQ can help in understanding the current practices of SATD management in industrial settings. For example, developers could group similar SATD to facilitate the SATD repayment. Software practitioners may be able to use some of these practices in their own context, while researchers may investigate ways of supporting them.

- *RQ3: How can we improve SATD management?*
  - *RQ3.1: What challenges do software practitioners face when managing SATD?*

    Rationale: Understanding the challenges of SATD management can help researchers to provide support for addressing those challenges. For example, prioritization of technical debt items is a typical challenge in any kind of technical debt, including self-admitted. If we obtain an in-depth understanding of why it is difficult to prioritize SATD in specific, we are in a better position to propose practices, tools, or guidelines to address this challenge.

  - *RQ3.2: What features should tools have to effectively manage SATD?*

    *Rationale:*As mentioned in Section 2, there are tools supporting SATD identification. However, we currently lack tools to assist in other activities of SATD management such as prioritization or repayment [20]. Answering this question can support the development of new tools or the improvement of existing ones that could help practitioners better and easier manage SATD.

Fig. 1 presents the overall framework of our approach to answering the research questions. The two major processes (i.e., data collection and data analysis) are elaborated in the following sub-sections.

3.2 Cases and Units of Analysis

This case study is designed as a single embedded case study [21]. Our case is a large software company in the *embedded systems industry* that chooses to remain anonymous. The software development in this company adopts *Scrum* development practices.

Because we focus on understanding SATD and its management process, as well as improving the latter, we collected data from two types of units. The first type of unit is software artifacts, including source code, commits, and issue tracking systems. It is noted that the studied projects mainly use C++ and XML files. There are, on average, 20 software engineers working on the analyzed projects. We identified and analyzed the nature of SATD from these software artifacts. The second type of unit is software engineers that participated in the development of a specific project. More specifically, each engineer represents a single unit. We conducted interviews with software engineers to derive their opinions on the aforementioned SATD nature, as well as to understand and improve SATD management. Details about the background of the practitioners are presented in Table 2.

3.3 Data Collection

As seen in Fig. 1, data is collected through *analysis of work artifacts* and *interviews*, which are third- and first degree data collection methods respectively, according to Lethbridge et al. [22]. These two methods are explained in the following subsections in detail.

3.3.1 Analysis of Work Artifacts

In order to answer the research questions, we choose a large-scale industrial project which contains eight sub-projects. More specifically, the selected project has over 475 K lines of comments, 21 K commits, and 130 K files (including documentation, test files, configuration files, etc.). Regarding issues, we collected 78 K issues from the issue tracking system. We note that, all embedded software in the selected company uses the same issue tracking system; thus, the collected issues come from all embedded software while comments and commits are from the selected project where we had access. Because we focus on three different types of work artifacts, namely source code comments, issue tracking systems, and commits, we obtained data from these different sources separately. For source code comments, we

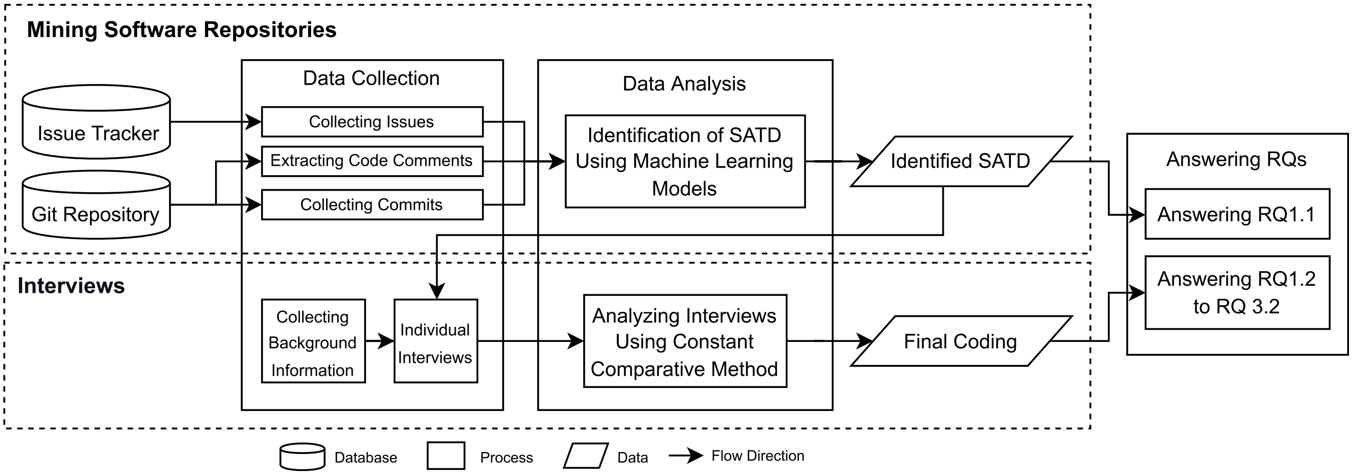


Fig. 1. The framework of our approach.

TABLE 1
Questions for Individual Interviews

Question	Related RQs
Do you acknowledge the identified SATD items from the different sources?	RQ1.2
Do you consider identified SATD items to be important?	RQ1.2
What do you think of different types of SATD?	RQ1.2
What do you think about the average time to close different types of SATD issues and non-SATD issues?	RQ1.2
Do these SATD items and statistics give you new information or insights about this project?	RQ1.2
What do you think are the relations between TD documented in different sources?	RQ1.3
Do you record TD in your project?	RQ2.1, RQ2.2
Which TD items do you usually record and which do not?	RQ2.1, RQ2.2
Where do you typically record TD?	RQ2.1, RQ2.2
Do you have any constraints on recording TD?	RQ2.1, RQ2.2
Which types of TD do you usually record?	RQ2.1, RQ2.2
What do you think are the differences between TD documented in different sources?	RQ2.1, RQ2.2
How do you decide on resolving one of the recorded TD items?	RQ2.3
How do you manage the recorded TD items in practice?	RQ2.4
How do you prioritize documented TD items in practice?	RQ2.4
What strategies do you follow to pay back documented TD?	RQ2.4
What challenges do you encounter when you manage recorded TD?	RQ3.1
What features would you like to have from an ideal tool to manage SATD?	RQ3.2

created a script to first retrieve all code changes in git and then extract all source code comments using the *Comment-Parser* tool.[2] We manually verified the correctness of the extracted comments with this tool before collecting the data. For issue tracking systems, we extracted all issue descriptions for analysis using the API of the issue tracker used by the company (Microsoft TFS). Lastly, for commits, we obtained all commit descriptions in git. The scripts for collecting data are included in the replication package.[3] We analyzed the latest version of the selected industrial project on July 7th, 2021.

3.3.2 Interviews

We have conducted semi-structured interviews to collect data from practitioners and answer the research questions. Semi-structured interviews were selected as they are an effective approach to exploring participants' thoughts and experiences in depth [23]. Regarding the interviewee selection, we aimed at recruiting participants that have different roles in the organization and have extensive experience in dealing with technical debt; these characteristics would allow us to explore the SATD management process and its tooling support from different perspectives (see Section 3.1). Thus, the contact person at the company selected and invited the interviewees based on these characteristics from approximately 60 embedded software engineers. These invitations were accepted by 12 practitioners.

TABLE 2
Background Information of Interview Participants

Interviewee ID	Role in the Company	Years of Experience
I_1	Architect	22
I_2	Architect	19
I_3	Architect	20
I_4	Software developer	22
I_5	Software developer	22
I_6	Software developer	20
I_7	Software developer	32
I_8	Software developer	17
I_9	Team lead	18
I_{10}	Software developer	34
I_{11}	Team lead	32
I_{12}	Project manager	24

Before the interviews, we extracted work artifacts from the selected project as aforementioned and identified SATD from them, as illustrated in Fig. 1. Then we selected a sample of 15 SATD items, to show to the practitioner in order to help them gain a basic understanding of what automatically identified SATD looks like and prepare for the interviews. The sample of 15 SATD items was selected from the set of identified SATD items in the previous step based on the proportion of different types of SATD, and consisted of 5 items from source code comments, 5 items from commit messages, and 5 items from issues. For example, two of the presented SATD items were: *"stupid code, why isn't this part of [function name]?"* and *"adding sanity check on timing."* Besides, we provided practitioners with an introductory document on SATD (including a definition and examples of SATD as well as the high-level goal of this study).

Practitioners were then interviewed one by one by at least two of the authors via a web-based platform. We asked practitioners to answer questions relating to their background, namely their role in the company and years of experience. This background information is presented in Table 2. After the background information collection step, we asked interviewees some introductory questions (e.g., What is your understanding of technical debt? Can you tell me some examples of technical debt?). These "warm-up" questions encouraged interviewees to think about their own experiences with technical debt so that they can answer the rest of the questions based on those experiences. During the interviews, we provided statistics on SATD (such as numbers and percentages of different types of SATD from different sources) in the selected project and the sample of identified SATD items. Practitioners were asked to think about the SATD examples and statistics before answering interview questions. Finally, the main part of the interview consisted of several questions aimed at answering the Research Questions

[2] https://github.com/jeanralphaviles/comment_parser
[3] https://github.com/yikun-li/satd-in-industry

TABLE 3
Examples of Different Types of SATD

Debt Type	Example
Code/Design debt	*"Perl protocol handler could be more robust against unrecognised types"* - [from Thrift-issue] *"Need to add better handling for hz instance cleanup."* - [from Camel-issue]
Test debt	*"TODO: need more tests* - [from JMeter-code-comment] *"Tweaks tests to be a bit more robust"* - [from TrafficServer-commit]
Doc. debt	*"FIXME: Document difference between warn and warning"* - [from JRuby-code-comment] *"we need to add it to the wiki page"* - [from Camel-issue]
Req. debt	*"TODO: add a dynamic context...* - [from Heron-code-comment] *"Union is not supported yet... I might be adding that capability quite soon."* - [from Samza-pull]

(as shown in Table 1); these were developed by following the interview guidelines of Seidman [24]. We asked the practitioners to talk about their ideas and opinions freely without restrictions. During the interviews, we also asked follow-up questions to delve into their experiences and understanding. Each interview took approximately 30 minutes. After obtaining permission from interviewees, interviews were recorded to be transcribed for analysis.

3.4 Data Analysis

3.4.1 Analysis of Work Artifacts

To identify SATD from work artifacts, we first collect data from the selected projects, as discussed in Section 3.3. Subsequently, we followed the results of our previous work [12] to identify SATD from different sources using a deep learning approach. This work is the only one focusing on accurately capturing SATD from different sources; specifically, the trained deep learning model achieved an f1-score (i.e., the harmonic mean of precision and recall) of 0.666, 0.644, 0.557 when identifying SATD from source code comments, commit messages, and issue tracking systems respectively. Moreover, the machine learning model can identify four types of SATD, namely code/design debt, requirement debt, documentation debt, and test debt. Examples of each type of SATD are presented in Table 3.

3.4.2 Interviews

To analyze the interviews, we first transcribed all interview recordings. It is noted that one of the interviewees did not grant us permission to record the interview, so this interview was transcribed on the fly during the meeting. Then, we followed an iterative qualitative data analysis process according to the *Constant Comparative* method of *Grounded Theory* [25], [26]. Specifically, the analysis process is composed of three main steps. The first step is *open coding*, which breaks the transcript text down to discrete textual segments, which are subsequently coded (i.e., labeled). When reading the interview transcripts, we continuously added new codes or changed current codes when necessary. The scope of codes varies, as it could be a phrase, a sentence, or a paragraph. Second, we applied *selective coding*, by constantly comparing different codes and annotations, and then merging similar codes. Third, we worked on the *theoretical coding* to establish conceptual relations between codes.

To ensure the agreement on codes, the first and second authors independently performed the *Constant Comparative* analysis process, discussed, and compared the generated codes to eliminate bias. Any disagreements between the two authors were subsequently resolved.

We used a professional qualitative analysis tool (ATLAS.ti[4]) to analyze the interview data. The analysis results and interview protocol are available in the replication package[3].

TABLE 4
Number of Different Types of SATD Items From Different Sources

Debt Type	Source			Total
	Comment	Issue	Commit	
Code/Design debt	3,139	9,318	2,236	14,693
Req. debt	602	702	119	1,423
Doc. debt	225	1,350	199	1,774
Test debt	63	540	93	696
All SATD	4,029	11,910	2,647	18,586

4 Results

4.1 (RQ1.1) What Are the Types, Amounts, Resolution Time, and Sources of SATD Items in Industrial Settings?

Table 4 presents the number of different types of SATD from different sources. We can observe that *most of the identified SATD is* code/design debt*(79.1%)*, followed by *documentation debt* and *requirement debt* (9.5% and 7.7% respectively). The least amount of identified SATD is *test debt* (3.7%).

As mentioned in Section 3.3, the issues come from all of the embedded software, while comments and commits are only from one (large) project. Thus, we cannot compare the absolute numbers of SATD items directly between sources. Thus, we look into the percentages of items across the different sources that contain SATD of different types (see Table 5). It is noted that, in this and subsequent tables, the highest values are highlighted in bold, while the lowest values are underlined. Specifically, we calculate the percentages of different types of SATD by dividing the number of SATD items of a specific type from a specific source by the number of items from this source. We observe that *the percentage of issues or commits being SATD issues or commits is significantly greater than source code comments* (16.3% and 12.7% versus 2.6%). Finally, the percentage of issues being SATD is slightly greater than commits.

Fig. 2 presents the number of cumulative technical debt admitted in different sources over time. As can be seen, software developers keep documenting technical debt in different sources. At the beginning of the studied period (before

[4] https://atlasti.com

TABLE 5
Percentages of Different Types of SATD Items From Different Sources

Debt Type	Source			Total
	Comment	Issue	Commit	
Code/Design debt	2.0%	**12.8%**	10.7%	5.9%
Req. debt	0.4%	**1.0%**	0.6%	0.6%
Doc. debt	0.2%	**1.9%**	1.0%	0.7%
Test debt	0.0%	**0.7%**	0.4%	0.3%
All SATD	2.6%	**16.3%**	12.7%	-

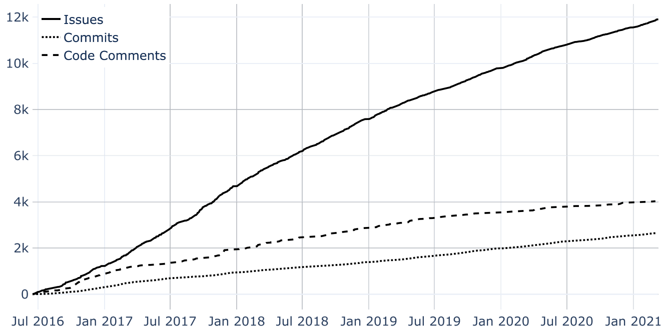


Fig. 2. Cumulative number of SATD items in different sources over time.

January 2017), the number of SATD in source code comments is comparable with the number of SATD in issues. Afterward, the rate of admitting technical debt in issues increases compared to source code comments.

Because issue tracking systems provide additional information (e.g., issue type, issue status, issue closed time) that is related to SATD introduction and repayment, we further investigate SATD in issue tracking systems. Specifically, we investigate the time required to resolve issues (with and without SATD), and the types of issues (e.g., backlog item or bug) with SATD. These are presented in the rest of this sub-section.

Table 6 presents the average time to close different types of issues and the percentage of different types of issues that are closed. As we can see, the average time to close different types of issues varies: *the average time to close non-SATD issues is shorter (47.2 days) compared to different types of SATD issues*; test debt issues take the longest average time to close (80.7 days). Moreover, non-SATD issues achieve the highest closed rate (75.5%), while requirement debt issues have the lowest closed rate (60.8%).

Specifically, to evaluate the significance level and the effect size of closing time between different types of issues, we choose Mann-Whitney test [27] and Cliff's delta [28]. The Mann-Whitney test is used to determine if two groups are significantly different from each other and is widely used in software engineering studies [4], [14]. The results are demonstrated in Table 7, while the p-value is highlighted when it is less than 0.05, which indicates the result has statistical significance. We can notice that, *in contrast to previous research findings [29], there are significant differences between the closing time of non-SATD issues and different types of SATD issues* (p-values are 1.1e-20, 1.3e-4, 9.9e-4, and 3.3e-7 respectively). Moreover, according to the Cliff's delta (i.e., 0.12, 0.24, 0.20, and 0.19 respectively), we can observe that the effect sizes[5] between them are all categorized as *small* [28]. Furthermore, closing time differences between test debt issues and code/design debt issues or documentation debt issues are statistically significant (p-values are 0.014 and 0.020). However, their effect sizes are *negligible* based on the Cliff's delta (i.e., 0.07 and 0.01). As specified by the effect size in Table 7, we find that the closing time differences between non-SATD issues and different types of SATD issues are greater than the closing time between different types of SATD issues. Additionally, we compare the average time to close different types of issues in Table 8. As we can see, except for *test issues*, all other types of issues align with the finding that SATD items take longer to solve. The reason for *test issues* not following this trend, might be due to the insufficient number of *test issues* in comparison to other types of issues (247 versus 885/4822/1333/4492).

Next, we study the occurrence of the types of SATD items (e.g., design or test debt) in the different types of

TABLE 6
Average Time to Close Issues and Percentage of Closed Issues

Type	Avg. Time to Close (d)	Pct. of Closed (%)
Code/Design debt	62.5	71.3
Req. debt	70.2	60.8
Doc. debt	60.4	72.0
Test debt	**80.7**	67.0
Non-SATD	47.2	**75.5**

TABLE 7
Comparison of Average Time to Close Issues Between Different Types of SATD and Non-SATD Issues

Pairwise Comparison		p-value	Cliff's Delta
Code/Design debt	& Non-SATD	**1.1e-20**	0.12 (small)
Req. debt		**1.3e-4**	0.24 (small)
Doc. debt		**9.9e-4**	0.20 (small)
Test debt		**3.3e-7**	0.19 (small)
Code/Design debt	& Test debt	**0.014**	0.07
Req. debt		0.361	-0.06
Doc. debt		**0.020**	-0.01
Non-SATD		**3.3e-7**	-0.19 (small)
Code/Design debt	& Doc. debt	0.641	0.07
Req. debt		0.160	-0.06
Test debt		**0.020**	0.01
Non-SATD		**9.9e-4**	-0.20 (small)
Code/Design debt	& Req. debt	0.247	0.12 (small)
Doc. debt		0.160	0.06
Test debt		0.361	0.06
Non-SATD		**1.3e-4**	-0.24 (small)
Req. debt	& Code/Design debt	0.247	-0.12 (small)
Doc. debt		0.641	-0.07
Test debt		**0.014**	-0.07
Non-SATD		**1.1e-20**	-0.12 (small)

[5] Effect sizes are marked as *small* ($0.11 \leq d < 0.28$), *medium* ($0.28 \leq d < 0.43$), and *large* ($0.43 \leq d$) based on suggested benchmarks [28].

TABLE 8
Average Time to Close Different Types of Issues

Type	Issue Type				
	Feature	Backlog Item	Bug	Task	Test
SATD	196.6	91.7	75.3	25.5	897.6
Non-SATD	186.6	75.9	68.0	19.4	1110.0

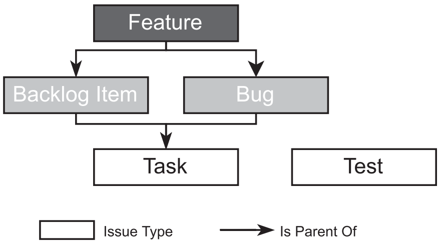


Fig. 3. Hierarchy of issue types.

TABLE 9
Number of Different Types of SATD Items in Different Types of Issues

Debt Type	Issue Type				
	Feature	Backlog Item	Bug	Task	Test
Code/Design debt	695	**3,770**	1,204	3,355	220
Req. debt	58	**350**	37	211	10
Doc. debt	85	512	46	**701**	0
Test debt	47	190	46	**250**	17
All SATD	885	**4,822**	1,333	4,492	247

TABLE 10
Percentage of Different Types of SATD Items in Different Types of Issues

Debt Type	Issue Type				
	Feature	Backlog Item	Bug	Task	Test
Code/Design debt	16.7%	**19.2%**	12.4%	9.2%	12.9%
Req. debt	1.4%	**1.8%**	0.4%	0.6%	0.6%
Doc. debt	2.0%	**2.6%**	0.5%	1.9%	0.0%
Test debt	**1.1%**	1.0%	0.5%	0.6%	1.0%
All	21.2%	**24.5%**	13.8%	12.4%	14.5%

issues (e.g., backlog or bug). Issue tracking systems typically provide a function that helps developers categorize and track the progress of specific types of work [30]. In the studied case company, the issue tracking system (Microsoft TFS) similarly supports specifying different types of issues. The most common issue types used by the case company, are *feature*, *backlog item*, *task*, *bug*, and *test*. The hierarchy of issue types is illustrated in Fig. 3. The studied organization uses these five issue types as defined by Microsoft [31]: *feature* is the highest-level type of work, it is associated with a specific product feature, and it is the parent of *backlog item* and *bug*. *Backlog item* is used to track development work, while *bug* is for tracking code defects. Moreover, *task* is used to track fine-grained work, i.e., it is a child of both *backlog item* and *bug*. Additionally, test-related issue types are used independently of other types. Because there are three test-related types, namely *test case*, *test plan*, and *test suite*, we group them together under the category of *test*. Table 9 presents the number of different types of SATD in accordance with these different issue types. As we can see, most of SATD is identified as *backlog item* and *task* (4,822 and 4,492 respectively). This indicates that *backlog item* and *task* are the two most popular issue types to admit technical debt.

Because the total numbers of the different types of issues vary, it is unclear which issue type has the highest percentage of SATD issues. To address this, we show the percentage of different types of SATD in accordance with different issue types in Table 10 and Fig. 4. We can notice that although *backlog item* and *task* have similar number of SATD issues (4,822 versus 4,492) in Table 9, *backlog item* has a significantly higher percentage of SATD issues compared to *task* (24.5% versus 12.4%). This means that backlog item*has the highest percentage and number of SATD issues among all issue types; in other words, it is the most used issue type for admitting technical debt*. This is in line with the definition of technical debt [1]: while defects and poorly/partially implemented features are symptoms of technical debt, pure technical debt items concern issues that directly affect the maintenance and evolution of a software system.

Additionally, as can be seen in Fig. 4, *feature* and *backlog item* have a higher chance to contain code/design debt (16.7% and 19.2% compared to the average of 12.8%), while *task* only has 9.2% of items being code/design debt (which is lower than average). Moreover, the percentages of requirement debt for *feature* and *backlog item* are also higher than other types of issues (1.4% and 1.8%, respectively). Finally, issues with the tags of *bug* and *test* are less likely to have documentation debt (0.5% and 0% compared to the average percentage of 1.9%).

4.2 (RQ1.2) How Is Automatically Identified SATD Regarded by Professional Software Engineers?

We report here the opinions of the interviewees on identified SATD and corresponding statistics produced by the automated SATD analysis. First, we present the attitude towards SATD identified from different sources (two examples of identified SATD are shown in Section 3):

- *Attitude towards SATD identified from code comments.* We found that eight out of ten interviewees that commented on this, confirmed that SATD identified from code comments is indeed technical debt from their perspective: *“yeah, those are the typical things [technical debt] that we enter in the code indeed.”*The other two interview participants also identified the vast majority of the discussed SATD items but were not very sure about one or two items: *“the first one I would say difficult, it could also be a matter of taste; [...] the last one is the same as the first one, really depends on the situation.”*
- *Attitude towards SATD identified from issues.*Six out of seven interviewees acknowledged SATD identified from issues as debt: *“I expect them to be part of the backlog list, but I cannot explain to you one by one; I think they are technical debt.”*Meanwhile, one interviewee found it difficult to judge whether it is SATD or not.

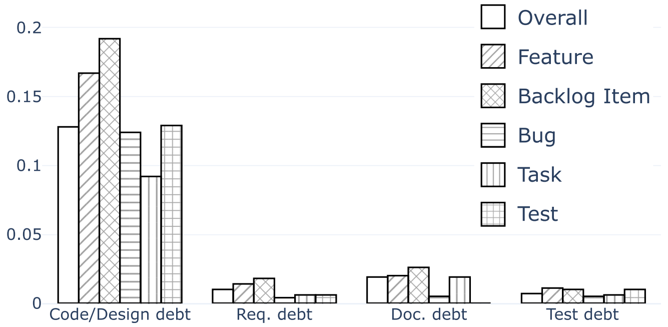


Fig. 4. Percentage of different types of SATD against different issue types.

- *Attitude towards SATD identified from commits.*Seven out of eight participants confirmed that SATD identified from commits is technical debt from their point of view. Interestingly, four out of these seven participants pointed out that SATD in commits concerns documentation of paying back technical debt instead of incurring technical debt: *"do you recognize technical debt in commits? yeah, but I think these are [documented] when somebody solves the technical debt in commit messages."*There is one interviewee that did not acknowledge SATD in commits: *"we always added text block in commits but not technical debt in commit messages."*

Second, we discuss the participants' opinions about the importance of the identified SATD items. Five out of nine interviewees mentioned that they need more information to determine the importance: *"you have to know the implementation to have some insights on how severe such a thing is and how much work it will be to solve it; it [importance] is not immediately clear from the TODO itself."*. Besides, two out of nine participants believed that some of the items are not important, while the others need more information to evaluate: *"the first one that I would say it's something that isn't really going to be resolved [...] the third one - it looks like it depends a bit on the on the functionality; is this really important or not, which is difficult to determine."* Meanwhile, the rest two of the nine interviewees pointed out that none of the identified SATD are important: *"it does not have urgency to be solved."*

Third, we describe the attitudes towards average time to close issues (see Tables 6 and 7). Seven out of nine participants expected the same as the figure they were shown: *"I think that is correct, which is about the balancing I told you between new functionality and technical debt."* They also mentioned that the reason behind this phenomenon is that they are under big pressure to implement new features or fix bugs instead of improving the quality of the code by solving SATD: *"I know [this] project is in challenging phase; they are high pressured to reach the time-to-market, [so] we are also under pressure to have shortcuts and do not redesigns [unless we are] told necessary by the developers."* One interviewee agreed that technical debt items take a longer time to be resolved compared to non-debt items, but he also pointed out that he expected documentation debt to take the longest time to be solved among all types of debt: *"I did not expect test debt takes that long; I would have expected the documentation debt to be there the biggest one."* Besides, one participant had different expectations than the results: *"I think it's a matter of calculation; it's the other way around [compared to the expectation]."*

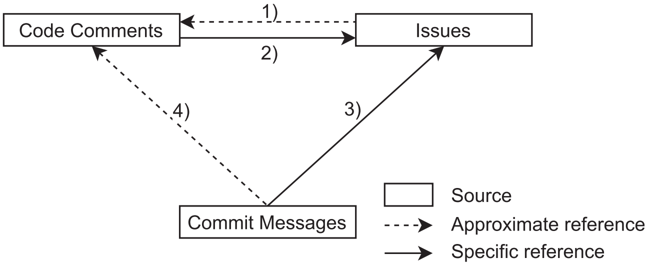


Fig. 5. Relations between SATD items in different sources.

Fourth, we report the thoughts of participants towards all the presented statistics (see Tables 4, 5, 6, 7, 9 and 10 and Fig. 2) and identified SATD items (some examples are shown in Section 3):

- *Giving insights on how technical debt is managed.*Five interviewees indicated that statistics help them understand how technical debt is managed in their projects: *"if you look at the statistics, I think that's the objective view of how things are managed."*Furthermore, two of the participants considered it useful that the statistics provide information about the different types of SATD and the average time spent on SATD items and non-debt items.
- *Showing what are the focus points.*One participant mentioned that statistics also show what the team emphasizes during SATD management: *"do you think statistics is useful? yeah, [...] I would say [I know] what is focused on."*
- *Increasing the awareness of SATD.*Five interviews revealed that statistics and identified SATD help developers be conscious of technical debt in the projects: *"it could always help to make us aware of technical debt."*
- *Providing insights on future improvement.*One interviewee stated that statistics could help developers become better and achieve higher productivity: *"it gives some insight on how can you improve and be more efficient in your work."*

4.3 (RQ1.3) What Are the Relations Between SATD in Different Sources?

The relations between SATD items in different sources, as derived from the data, are summarized and presented in Fig. 5. It is evident that technical debt admitted in code comments and issues is referenced in the other two sources, while technical debt admitted in commits is not mentioned in other sources. Because each issue has a unique issue ID, developers can refer to that ID when referencing a SATD issue: *"in most cases, we try to add the issue ID within the comments."* We call this reference a *specific reference*. On the other hand, since there is no unique identifier for each source code comment, it is impossible to reference specific comments. Thus, such references are usually *approximate references*. We describe the relations in detail as they are numbered in Fig. 5:

1) *Technical debt admitted in code comments is referenced in issues.*Two interviewees mentioned that it is not common to reference SATD code comments within issues: *"in the issues, nine out of ten times, [developers] never write down which actually line or file [is] related."* However, one of them also pointed out that developers sometimes note down in issues the approximate location of technical debt which has been documented in code comments: *"[developers] only specify a certain piece of code where the problem resides."*
2) *Technical debt admitted in issues is referenced in code comments.*The links from issues to code comments are mentioned by four interview participants. They tend to add issue IDs (unique identifiers) in the code comments to establish clear links: *"sometimes you add a link to the issue in the code."*
3) *Technical debt admitted in issues is referenced in commit messages.*Three interviewees mentioned that SATD in issues is also referenced in commits: *"in the commit, we are able to tag the issue item, and then the link between commits and issues is made automatically."*This gives developers a better understanding of the changes in the commits: *"I do not know if it was a single line commit message, which is vague, short, and without explanation [...] we need to have more information in the commit or a link to the issues."*
4) *Technical debt admitted in code comments is referenced in commit messages.*One interviewee indicated that the repayment of SATD in code comments might be documented in commit messages: *"there could be a link if you have the previous one in comments, when somebody solved it, probably in the commit message you might record the resolve of it."*

4.4 (RQ2.1) When Is Technical Debt (Not) Admitted in Source Code Comments, Issue Tracking Systems, and Commit Messages?

During the interviews, we established that software engineers tend to admit technical debt in different sources for different reasons. For each source (i.e., source code comments, issue tracking systems, and commit messages), we report several cases why technical debt is being admitted. We start first with the *source code comments:*

- *Scale of technical debt is small.*Four interviewees mentioned that developers tend to document small technical debt items in source code comments: *"if it [technical debt] is too small, just admit it in the code comments."*Regarding what *small* technical debt actually means, as an interviewee stated: *"if you look at things in source code, they are typically smaller; those things are just magic number or making this as parameter..."*
- *Solving technical debt brings little or no benefit.*When solving technical debt yields small or negligible benefit, developers tend to document it in comments: *"if we will not gain the advantage over anyway, then probably something will be noted in the software [...] a comment will be added."*
- *Deciding not to fix the technical debt.*Two participants pointed out that if developers reach an agreement on not fixing the technical debt, they usually just document it in code comments: *"if we already decide not to fix this technical debt, then probably it will remain as comments."*
- *Helping other developers to become familiar with technical debt related to code and its rationale.*Five interviewees mentioned that it is important to document technical debt and its rationale to help other developers become aware of problematic code and the reasons behind it: *"the comments in software to make sure that when people are facing troubles and having a look at the software again that they know about the facts that have been made to some different choices and which could result in a problem."*
- *It concerns requirement debt.*Three interviews revealed that if the technical debt is of the type requirement debt, such as partially implemented requirements, developers prefer to document it in source code comments: *"because we have not finished yet, it [code comment] is typically written down while developing the feature."*

Second, for *issue tracking systems*, technical debt is documented in the following cases:

- *Scale of technical debt is big.*Six interview participants indicated that developers always document large-scale technical debt in issue trackers: *"If you look at issue tracker... you have to fix this entire piece of code, that's a bigger span, while in code it's basically for the next line."*
- *Technical debt is part of the future plan.*Five interviewees pointed out that developers always document technical debt in issues when they actually plan to fix them in the future: *"I think that we create issues for them [technical debt] to make sure that they will become part of the future plans."*
- *Features only supported by the issue tracker.*Issue tracking systems provide features that are not provided by code comments or commits, such as uploading attachments and assigning severity levels for issues. An interviewee mentioned that he always summarizes technical debt and designs in a word document on a daily basis, then uploads it as an attachment when creating a new issue.
- *Track technical debt repayment in the engineering phase.* Developers in this case study refer to software development in later iterations with *engineering phase*, which is different from the early phase of development. When developers want to track what changes will be made to solve technical debt in the engineering phase, they create issues: *"I think in engineering phase [when I] have to clean something up, I will definitely make issues, so you can always see what has been done; in the early phase, it's just about what works are expected."*
- *Duplicate existing technical debt admitted in code comments.*Three software engineers believed that existing technical debt in comments should also be admitted in issues to facilitate their tracking: *"if there are still TODOs in the code, there should also be an issue that something still needs to be done."*

Third, developers document technical debt in *commit messages*, in the following cases:

- *Commits introducing Technical Debt.*One interviewee pointed out that, if commits include workarounds that are typical of technical debt, they usually document those problems, as well as the related issue keys in the commit messages: *"we have certain commits which indicate that we have to make shortcuts or we have implemented a temporary situation."*
- *Commits related to technical debt repayment.*Three interviews disclosed that if commits are about technical debt repayment, they always document it in commit messages: *"these are when somebody solved technical debt in these commit messages."*

Finally, we also summarize the cases when technical debt is ignored or not documented in artifacts:

- *Developers are under pressure and forget to document technical debt.*Three participants pointed out that if the pressure is very high, developers usually focus on other work and postpone technical debt documentation; in most cases, it is eventually forgotten: *"if the pressure is really high, [...] you will do that [technical debt] tomorrow, and tomorrow has another thing that got forgotten."*
- *Certain types of technical debt are ignored.*According to four interviews, some developers do not consider certain types of technical debt to be important and choose not to document them. Specifically, interviewees believed that developers pay less attention to documenting test debt: *"I think we don't have that many technical debt items for missing test cases; I think you more or less know about them, but no real documentation about test cases and actual implementation."*
- *Scale of technical debt is small.*When the scale of technical debt is small, developers may decide not to document it at all because of its low impact: *"what are the reasons not documenting technical debt? [it depends on] how big is the technical debt, if just a small thing, it's probably not."*
- *Technical debt in legacy code.*Two interviewees reported that developers are aware of the limitations of technical debt in old parts of the system and choose not to document it, because they know it will not be fixed anyway: *"[if] the architecture is already fifteen years old [...], you know what the limitations are, you can still write technical debt to make it better, but you know it will not be fixed anyway."*
- *Short life of technical debt.*We noticed that when developers think the technical debt will be solved in the near future, they might choose not to document it (mentioned by three interviewees): *"I know that for the old release, we make a quick workaround, but we don't mark [it] as technical debt because we make the actual good solution in our mainline immediately."*
- *Direction is unclear in early phases.*Because of uncertainties in the early phases of projects, software engineers may choose not to document it: *"At the beginning of the project, it can go anywhere, so if you put a lot of effort in explaining why something is done, it takes lots of time."*
- *The responsibility of other developers.*In some cases, developers are in charge of certain parts of software development or documentation update. When other developers encounter technical debt, they prefer to let the responsible person document it: *"if it's someone else's documentation, I might mention it to someone. I usually do not create an issue for that."*
- *Treating technical debt as common knowledge.*One participant mentioned that technical debt is not documented when it is known by everyone in the team: *"[technical debt] is not documented [...] [when we] have accepted [it], and treat [it] as a common knowledge of the team; the team members know that issue is there, or inconvenience is there."*

4.5 (RQ2.2) What Are the Pros and Cons of Admitting Technical Debt in Different Sources?

The pros and cons of documenting technical debt in *source code comments* are summarized below:

- *Pointing to problems in the code.*Five interview participants pointed out that technical debt documented in code comments could help developers understand the existing technical debt in code and potentially solve it: *"make sure that people are looking at the software [reading source code], they will be familiar with the fact that there is technical debt in code."*
- *Long lifetime of code.*One interview participant indicated that code is a very stable artifact compared to others. In contrast to other artifacts, comments in source code will not disappear in the future: *"I have seen tools coming and gone; five years back we [switched the issue tracker] [...], [but] I have code older than five years, maybe ten years old, so I don't know the change request anymore from seven years back and the rationale; the only thing I have is just the source code."*
- */ Limited visibility.*On the one hand, documenting technical debt in code comments causes less disturbance to other developers: *"too many detailed tasks [in issues] does not help which could bother teammates [...] just admit them in the code comments, [...] because you intend to solve it soon anyway."*On the other hand, it could restrict the visibility of technical debt, resulting in paying less attention to it: *"they are not visible anymore, only if you run into that."*
- *Resolving it depends on the initiative of developers.*Three interviewees reported that it highly depends on software developers to solve or leave technical debt admitted in code comments: *"you need be lucky that someone will be working on this to get a solution."* Thus, documenting technical debt in the source code can act both as an advantage (if it gets resolved) and a disadvantage (if it is ignored).

Subsequently, we list the main pros and cons of admitting technical debt in *issues:*

- */ Visible to the whole team.*Six interviewees mentioned that technical debt admitted in issues has the advantage of being visible to everyone in the team, helping developers to keep track of it: *"issue tracker is used for recording important technical debt which is shared in the team."*

- *Issue trackers provide features not supported by other artifacts.*Two interviewees revealed that issue tracking systems provide several features that support technical debt management. Specifically, issue trackers can give developers an overview of all documented technical debt: *"it's a good thing that technical debt is mostly recorded in the issue tracker because this gives an overview."*It also supports uploading technical debt information as attachments, assigning the issue type, and assigning the issue severity. Finally, it supports keeping track of what has been done about the technical debt: *"I will make an issue, so you can always see what has been done."*
- *Support planning to resolve technical debt.*Six interviewees reported that technical debt documented in issues will be a part of the future plan and be resolved eventually. This is because, in addition to developers, team managers also participate in the management of SATD in issues: *"[as the team lead] once they are in issues, they are in my list of choosing priorities, that I can deal with it."*

Finally, the pros and cons of recording technical debt in *commits* are presented below:

- *Providing explanation for TD changes.*Two interviewees mentioned that commits are important to explain what TD changes are made to the repository, such as introducing TD, modifying TD, and repaying TD: *"commit messages should include what has changed and what has been done, also for changes to technical debt."*
- / *Limited visibility.*Similar to code comments, technical debt admitted in commits has limited visibility. From the viewpoint of the team lead, it is not visible to him: *"if they are in comments or commits, they're not on my desk."*
- *Resolving it depends on the initiative of developers.*There is no guarantee that technical debt admitted in commits will be resolved. It depends on the developers to solve it or leave it. The team lead only manages technical debt documented in issues: *"I don't manage technical debt in code comments and commits at all, that's really depending on the engineer's responsibility."*

4.6 (RQ2.3) What Are the Triggers to Pay Back or Not Pay Back SATD?

According to the interview responses, software developers tend to repay SATD in the following cases:

- *SATD is involved in upcoming changes.*Based on six interviews, developers always choose to repay SATD when changes are going to take place in the same part of the system. This is because technical debt could make the changes more difficult: *"for instance the parameterize thingy, if I was doing a change which actually needs that or in the same area, I would take that along because that would really help me if I solve it."*
- *SATD is related to bugs.*In another case, three interviewees reported that when they find SATD connected to bugs, they will solve the technical debt: *"we really have to start solving [the technical debt] that keeps bugs popping up with that same piece of code, [for example if] you have these bugs popping up [while] you see test debt, [it happens because you] don't have test cases in that area."*
- *SATD is experienced by stakeholders.*One interviewee indicated that SATD observed by stakeholders is more important: *"I will focus first on technical debt that is experienced by stakeholders."*
- *SATD hinders other tasks.*Two participants pointed out that they need to repay SATD when it prevents them from keeping making progress: *"if they are hindered by [technical debt], then it's important to focus on the bad choices."*
- *Small SATD that can be solved easily.*Based on three interviews, we found that when developers encounter SATD and think the debt can be paid back easily, they prefer to solve it straight away: *"if there is a small [technical debt], there is time left in your sprint then you could pick up such a small item."*
- *The same SATD keeps annoying developers.*Two responses indicated that when developers encounter a technical debt item, which is repeatedly of concern to them, they will take some time to solve it: *"if you hit the same technical debt item and it annoys you enough, then it will be solved."*
- *Certain types of SATD are valued more than others.*Some developers believe that certain types of SATD are more important than others, and they choose to repay them with higher priority. More specifically, two interviewees stated that they prioritize test debt: *"I would prioritize test debt; that's critical on the code quality."*On the other hand, two participants mentioned they always give design debt higher priority: *"you also see preferences more to the design debt to documentation debt."*
- *Too much SATD in the area.*Five participants pointed out that when too much SATD is accumulated in a specific part of the system, it gets to be paid earlier rather than later: *"if there is a lot of technical debt in those modules, you might want to pick up earlier, cause if there is some TD there, maybe something wrong in the design..."*
- *Location of SATD is special.*One interview participant mentioned that SATD in different parts of the project is treated differently. They always give high priority to SATD in certain modules: *"it is up to the part of the project if I make a shortcut that should not be in; I am aware of it and will resolve it."*
- *Potential risk of SATD is high.*According to three interviews, when the potential risk of SATD is very high, SATD should be worked on: *"I think you should work on the most important things, the highest risk things."*
- *Have sufficient time.*Three interviewees indicated that when they have sufficient time, they will take some time to solve SATD: *"when do I decide to solve technical debt apart? when I have time in the program."*
- *Software craftsmanship.*Two interview participants reported that some developers have the attitude of striving to deliver software of high quality: *"[if] I have craftsmanship, I deliver software as the input and believe the software needs to be correct and needs to be maintainable."*

Meanwhile, in the following cases, SATD is ignored and left unresolved:

- *Repaying SATD brings small benefit.*Three interviewees mentioned that if the software works, repaying SATD yields only a small benefit. Thus, developers tend to not pay back the debt: *"if it already works, why make it better? someone pays for it."*
- *Repaying SATD takes too much effort.*Four participants had concerns about the considerable effort required to pay back SATD: *"we don't want to invest in it, due to [...] too much effort."*
- *Potential risks of paying back SATD.*Four interviewees expressed the concern that it could be risky to repay SATD, as it might break existing functionalities: *"there is some regression risk involved; it should be simple but sometimes takes a long time to finish."*
- *Certain types of SATD are ignored during repayment.*As one of the participants mentioned: *"the one writing [documentation] typically isn't the user of it"*; some developers simply give documentation debt low priority: *"I don't like writing documentation, so I really try to postpone writing the document."*Besides, two participants stated that some developers do not see test debt as important and choose not to repay it: *"[some] engineers don't see writing tests really helps them because you have implemented something; it runs on a machine and it works."*Moreover, another interviewee pointed out that architecture debt is sometimes ignored in the maintenance phase: *"architecture debt is addressed differently than design and test debt because it more prevents production; architecture debt also depends on the development phase; if it goes to maintenance phase from earlier phases of building a new product, we often keep the debt as long as it doesn't break functionality."*
- *Learning effect for the SATD creator.*Another interviewee believed that the debt creator should solve it, to be able to learn from it: *"it is important to close the feedback loop; if others resolve it [technical debt], the people who created it will never learn from it."*
- *Inactive code.*Two interviewees reported that when the code is inactive and there are no changes planned for it, related technical debt does not have priority to be paid back: *"If there are no changes or features, or plans for this, maybe [it] will not be used anymore, then that's not so important."*
- *Careless developers.*Lastly, one interview revealed that irresponsible developers also lead to SATD unsolved: *"some people say we should solve it, and then they don't stick to it."*

4.7 (RQ2.4) What Practices Are Used to Support SATD Management in Industrial Settings?

In the following, we summarize practices used to assist in SATD management. First, we describe practices that help prioritize SATD using different criteria:

- *Custom list.*Four interviewees indicated that they maintain a list of SATD with an order of priority. Specifically, they usually put the high-priority SATD on the top of the list for quicker repayment: *"[we] try to prioritize the list, and the most important items are on the top that needs to be solved first."*
- *Severity level of tickets.*Issue tracking systems always support setting the priority for each issue ticket (e.g., block, minor, and trivial) [32]. One interview participant mentioned they also use the issue tracker's built-in function to set the priority of each issue: *"most items are already categorized with a severity."*
- *Type of tickets.*Five interviewees mentioned that issue types have an impact on the priorities of issue tickets. They choose different issue types when creating issues with different priorities. For example, the interviewees considered that *bugs* have higher priority than *backlog items: "I would say I had a task if it is for short term if you intend to solve it within this sprint, if [...] you create a backlog item, [it could just] disappear, so it would be better to write the bug then at least you have a process to handle these."*
- *Referencing issue keys.*One participant indicated that when adding a reference to an issue ticket, the SATD in code comments will get higher priority: *"if that comment references an issue, it will automatically get more priority."*

Second, we report two common practices to efficiently pay back SATD:

- *Grouping related technical debt items.*Two participants indicated they usually group related technical debt items and investigate them together: *"we group them [technical debt] together; that's we say, those four or five items are in the same area, let's now take a look at them together, to be more effective."*
- *Grouping technical debt and development tasks.*Two interviews revealed that developers also group technical debt and development tasks (e.g., fixing bugs, creating new features, and adding tests). Then they solve them jointly: *"when we take technical debt we also resolve other things, which is more efficient."*

4.8 (RQ3.1) What Challenges Do Software Practitioners Face When Managing SATD?

In the following, we summarize the challenges for SATD management:

- *Convincing developers not to introduce SATD, when not necessary.*Three interviewees indicated that some technical debt can be paid back easily, so it should not be incurred in the first place: *"many of these [comments] seem to be fixed in five minutes, I think they shouldn't write these comments; they do not look like effort-intensive."*.
- *Prioritizing SATD.*Five interviewees pointed out that it is hard to determine priorities of SATD and other works: *"the biggest challenge is setting the priorities [...] the challenge is always what's the best to do, a piece of functionality or technical debt?"*
- *Getting resources to pay back SATD.*Based on two interviews, we found that getting resources for debt repayment remains challenging: *"to get technical debt on the agenda is a difficult task [...] there's always an argument to not work [on them]."*

- *Dealing with undocumented technical debt.*Three interview participants mentioned the difficulty of dealing with undocumented debt: *"we struggle with the ones [technical debt] we are not aware of or somebody identified without clearly communicated as being technical debt."* One interviewee specified that it is especially challenging when dealing with technical debt in old parts of the system without documentation: *"what challenges did you face when dealing with technical debt? dealing with legacy code in general [...] re-engineering the code or design sometimes is difficult [...] it could be a lot of helpful if there is some code documentation."*
- *No guideline for SATD documentation.*Four interviewees pointed out the problem of not having concrete guidelines for SATD documentation: *"we don't have any agreements on when you have technical debt and want to add some comments within the software, then please use certain tags."*
- *No guideline for SATD repayment.*Besides, two interviewees reported that there is no guideline for repaying SATD: *"there is no complete guideline; if you have to solve this, then you should do this, this, and this."*
- *Dealing with consequences of SATD.*Two interviewees found that it is challenging to deal with an increasing list of SATD items, as it causes SATD items to be ignored or not to document new technical debt: *"sometimes the technical debt items get out of sight because the list is becoming too long and you forgot about it; I am not sure how to deal with that growing list of technical debt items."*Meanwhile, another participant stated the harmfulness of accumulating technical debt without proper management: *"in another project, it was horrible; the big redesigns block the whole development, which also affects the trust of the software."*

4.9 (RQ3.2) What Features Should Tools Have to Effectively Manage SATD?

In the following, we report the tool features that developers thought were useful for SATD management. We categorize features into four groups: SATD identification, SATD traceability, SATD prioritization, and SATD repayment. SATD identification-related features are reported as follows:

- *Automated SATD identification.*Seven interviewees mentioned that it would be useful to be able to automatically identify SATD from different sources: *"I think [the tool should support] the identification of technical debt, scanning code or issues."*The interviewees suggested the following ways to present the identified SATD:
  - *Show the list of identified SATD.*Two interviews indicated that identified SATD items should be listed: *"if the tool could make some kind of printouts of technical debt items in your source code, then I can imagine that I will sit together with some engineers, walk through the list, find the most important, and solve them."*
  - *Show the quantity of SATD in the system.*One participant suggested showing the number of SATD items in the dashboard to increase the visibility of SATD in code: *"this dashboard will show you how much TODO in our code, so that is visible for the whole team."*
  - *Show the evolution of SATD quantity over time.* Another participant mentioned that it would be useful to have a function showing the number of SATD over time to know when they introduce more SATD or less SATD: *"[the tool should show] total amount technical debt in the system evolving during the development of the system, [so we can know] do we have more technical debt in the early phase."*
  - *Show the quantity of SATD in different modules.*As mentioned in Section 4.6, if too much SATD is accumulated in a part of the system or in some specific modules, developers would give the debt higher priority. Thus, they would like the tool to show the quantity of SATD in different modules: *"what would you like to see? [I want to have] some insights into which module has a lot of technical debt."*
- *Automated differentiation between fixed SATD and unfixed SATD.*As stated in Section 4.4, the identified SATD may be either repaid or not. There needs to be a distinction between them; as one participant stated: *"I am curious only about the open ones [unsolved debt]."* The participants mentioned the following means to visualize the distinction between solved and unsolved SATD:
  - *Show the period between SATD introduction and repayment.*One interviewee mentioned that he wanted to know the repayment time of SATD: *"[the tool should show] how much time is in between when we decided to introduce technical debt and when will things be solved."*
  - *Show how long unsolved SATD survives.*Another interviewee indicated that the tool should show the survival time of SATD: *"[the tool should show] how long technical debt is there."*
  - *Show the timeline of fixed and not fixed SATD items.* Another interviewee was interested in the point in time when they decide to either pay the TD back or not: *"[the tool should show] what is the moment we solve most technical debt? when do we decide to leave technical debt in the system and stop working on them?"*

Next, SATD traceability-related features are presented:

- *Automated tracing between SATD in different sources.*In Section 4.3, we observed some relations between SATD in different sources. But, some of these relations (e.g., technical debt admitted in code comments referenced in issues) are rarely documented. Thus, two interviewees think it would be very helpful if the tool could build traces automatically between SATD in different sources: *"linking back and forth would really help in getting an overview about technical debt things."*
- *Automated tracing between SATD and code.*Two participants mentioned that it would be useful to know the location of SATD in the code: *"I would like to know which area of code the technical debt is located at."*

- *Automated tracing between SATD and related development tasks.*As described in Section 4.6, when SATD is involved in upcoming changes, it is usually prioritized. Thus, developers are interested in which to-do items (e.g., fixing bugs, creating new features, or adding tests) are related to the SATD: *"[the tool should find] related work to it."*

Subsequently, we report the suggested features related to SATD prioritization:

- *Automated SATD prioritization.*Two interviewees wanted to automatically prioritize SATD: *"I'm looking to which part of technical debt could be left and which part of technical debt really needs to pay attention to."*
- *Automated identification of SATD risk.*Three participants mentioned that SATD risk identification should be supported by the tool: *"I am looking at [...] what is the pain? do I need to solve it? what are the consequences [of] not solving it?"*
- *Automated estimation of benefits to solve SATD.*Two interviewees indicated that estimating the benefits of solving SATD (e.g., how much technical debt interest will be saved) could be one of the tool's functions: *"need to know what the benefit of it [solving SATD], what is gained by it."*
- *Automated estimation of cost to solve SATD.*Based on four interviews, developers mentioned that automated estimation of SATD repayment cost (also referred to as the principal of technical debt) is useful: *"the other thing is how much effort does it take to get rid of this technical debt."*

Finally, there is one feature related to SATD repayment:

- *Automated SATD solution suggestion.*Two interview participants asked for the tool to provide some potential solutions (e.g., refactorings) for paying back SATD: *"the other tool could [provide] [...] possible routes of solution."*

5 Discussion

According to the study design (see Section 3), we formulated three main research questions to investigate the nature of SATD, the SATD management activities, and SATD management improvement. Thus, we organize the discussion into three parts: the discussion about the nature of SATD, the discussion about SATD management activities, and the discussion about SATD management improvement.

5.1 Nature of SATD

As mentioned in Section 3, there are significant differences between open-source and industrial projects. It is important to know these differences in order to understand how to better manage SATD in the two cases: what works for each case, what works for both, and what can be reused from one to the other. Thus, we compare the characteristics of SATD in industrial and open-source projects.

We first compare the percentages of different types of SATD in industrial projects (IP) and open-source projects (OSP). The comparison is presented in Table 11. It is noted that data from industrial projects are calculated based on Table 4, while the open-source data are obtained from 103 open-source projects from our previous work [12]. Specifically, these 103 open-source projects are from the Apache ecosystem. They are of high quality and well-maintained by mature communities. Observing the table, we can find that the majority of SATD is *code/design debt* in both industrial and open-source projects, ranging from 77.9% to 84.4% and 73.2% to 80.6% respectively. Moreover, we notice that the second most prevalent types of SATD from different sources are consistent when comparing industrial and open-source projects. Specifically, *requirement debt* is the second most popular SATD type from code comments in both kinds of projects, while *documentation debt* is the second most prevalent type of SATD from issues and commit messages in both kinds.

TABLE 11
Comparison Between Percentages of Different Types of SATD Items in Industrial Projects (IP) and Open-Source Projects (OSP)

	Source					
Debt Type	Comment		Issue		Commit	
	OSP	IP	OSP	IP	OSP	IP
Code/Design debt	80.6%	77.9%	80.0%	78.2%	73.2%	84.4%
Req. debt	12.0%	14.9%	1.0%	5.9%	1.1%	4.5%
Doc. debt	4.3%	5.6%	11.1%	11.3%	19.3%	7.5%
Test debt	3.2%	1.6%	7.7%	4.5%	6.4%	3.5%

Implication 1: The majority of SATD in both industrial and open-source projects is code/design debt. The second most prevalent types of SATD from different sources (requirement debt or documentation debt) are also the same in the two settings. Researchers could further investigate whether the types of SATD are similar in the two settings.

Regarding differences between the two settings, in Table 11, we observe the following: 1) the percentages of *requirement debt* from different sources in industrial projects are significantly higher in comparison with open-source projects; 2) the percentages of *test debt* of industrial projects are lower than open-source projects in the different sources. For the differences in *requirement debt*, we conjecture that this might result from the relatively high level of difficulty in changing embedded systems [33] and the high pressure of the studied projects as mentioned in Section 4.2: *"I know [this] project is in challenging phase; they are high pressured to reach the time-to-market, [so] we are also under pressure to have shortcuts and do not redesigns [unless we are] told necessary by the developers."* Similarly, Zampetti et al. [5] found that industrial developers reported releasing software under more pressure compared to open-source developers.

Moreover, the lack of self-admitting test debt in industry is consistent with our findings in Section 4.4 that *certain types of technical debt are ignored* as some developers mentioned: *"I think we don't have that many technical debt items for missing test cases; I think you more or less know about them, but no real documentation about test cases and actual implementation."* However, the reasons behind this phenomenon need further investigation. Additionally, according to the differences in percentage between different types of SATD, as well as the interviews, we advise

TABLE 12
Comparison Between SATD Percentages in Different Sources in Industrial Projects (IP) and Open-Source Projects (OSP)

Source	Percentage of SATD			
	OSP	IP	Percentage Diff.	Chi-Square Test
Comment	5.2%	2.6%	-50%	$\chi^2(1) = 2150.66\ p < 0.00001$
Issue	13.0%	16.3%	+25.4%	$\chi^2(1) = 666.83\ p < 0.00001$
Commit	11.3%	12.7%	+12.4%	$\chi^2(1) = 37.33\ p < 0.00001$

practitioners to create thresholds based on the percentages of different SATD types to evaluate the quality of SATD management. For example, if there is significantly less test debt documented than the threshold and the code analysis tool shows low coverage, this might refer more to the reluctance of developers to admit test debt rather than low test debt.

> *Implication 2: The percentage of requirement debt is higher while the percentage of test debt is lower in the studied industrial projects in comparison with open-source projects. The differences between percentages of different types of SATD between different projects should be further studied to potentially create thresholds for evaluating the quality of SATD management.*

Subsequently, we compare the percentages of SATD (irrespectively of type) in industrial projects and open-source projects, as shown in Table 12. Specifically, the data from industrial projects are obtained from Table 5, while the data from open-source projects are still acquired from 103 open-source projects from our previous study [12]. We perform chi-square tests to compare the number of SATD items from different sources in open-source and industrial projects. As can be seen in Table 12, the percentages of technical debt admitted in industrial and open-source projects are comparable. Interestingly, the percentage of SATD from code comments in industrial projects is just half of the percentage of open-source projects (2.6% versus 5.2%); this is statistically significant (p-value < 0.00001). Thus, less technical debt is admitted in code comments in industrial projects compared to open-source projects. In a recent study, Zampetti et al. [5] investigated the preferences for documenting technical debt in source code comments between industrial developers and open-source developers. They surveyed 101 software developers and found open-source developers tend to admit more technical debt in code comments in comparison to industrial developers. Our results confirm these findings.

> *Implication 3: The overall percentage of technical debt admitted in industrial and open-source projects are comparable. There is, however, less technical debt admitted in code comments in industrial projects compared to open-source projects (2.6% versus 5.2%).*

Furthermore, we can see that significantly more technical debt is admitted in issues and commits (+25.4% and +12.4% respectively) in the industrial projects compared to the open-source ones in Table 12. This is likely related to the practices for SATD management that are followed in the two types of projects. Within our industrial partner, we established a preference for documenting SATD in issues for better tracking. However, there might be other factors that have an impact on technical debt documentation. Thus, the evidence shows that more technical debt is documented in issues and commits within the studied industrial projects compared to open source projects. However, these differences are not as large as the differences in source code comments (25.4% and 12.4% versus 50%). Ultimately developers have the need to document SATD somewhere: industrial developers seem to prefer issues and commits while OS developers have a preference for source code comments. Researchers could further investigate if this is true more generally with further studies in industry.

> *Implication 4: There is more technical debt admitted in issues and commits in industrial projects compared to open-source projects (+25.4% and +12.4%). However, the differences are not as large as the differences in source code comments. Researchers can investigate the differences in other projects and look deeper into the causes of these differences.*

Subsequently, we compare our results regarding the issue closing times and issue closing percentages with previous work. There are two studies that investigated the closing time of issues. The first one was conducted by Bellomo et al. [29], who hypothesized that technical debt issues take a longer time to resolve than non-technical debt issues. However, they found that the average open days of issues vary greatly, and their results did not support their hypothesis. In the other study by Xavier et al. [34], they found that the median time to close technical debt issues is longer than other issues (16.7 days versus 4.0 days). In this paper, we investigated the same research question in industrial settings (see Section 4.1). The results shown in Tables 6 and 7 indicate that technical debt issues take a longer time to resolve compared to non-technical debt issues (with statistical significance). This supports the hypothesis proposed by the previous study [29]. It is noted that our study focuses on SATD in industrial settings. This hypothesis still needs to be tested in open-source settings. Furthermore, our study confirmed that the reason why technical debt issues take a longer time to close and have a lower closing rate is that developers are under pressure to implement new features or fix bugs instead of paying back technical debt (see Section 4.2).

> *Implication 5: The hypothesis that technical debt issues take a longer time to resolve than non-technical debt issues is supported by our study in industrial settings. Researchers could further examine this hypothesis in open-source settings and compare the results.*

Moreover, our study investigated the time to close issues and the closing rate of issues with different types of SATD and non-SATD. The results show that certain types of SATD take a significantly longer time to resolve than certain other types of SATD. Specifically, observing Tables 6 and 7, we notice that *requirement debt* and *test debt* issues take a longer time to close compared to the other two types of SATD (70.2 and 80.7 days versus 62.5 and 60.4 days). Moreover, they also have the first and second lowest closing rates compared to others (60.8% and 67.0% versus 71.3% and 72.0%). Furthermore, we find that *test debt* issues take a significantly longer time to resolve than *code/design debt* and *documentation debt* issues (*p*-value is 0.014 and 0.020 respectively). Overall, we observe that *requirement debt* and *test debt* might have a lower priority compared to *code/design debt* and *documentation debt*. This finding is in agreement with the findings of Ebert and Jones [35] which showed that requirements and tests are the major cost drivers in embedded-software development (requirements engineering and testing take most of the effort). As mentioned in Section 4.6, *"repaying SATD takes too much effort"* is one of the triggers not to pay back SATD. Thus, we conjecture that *requirement debt* and *test debt* issues take a longer time to close because they require more effort to repay than other types. The detailed reasons behind the observations still need to be further investigated and validated. Additionally, researchers could use our trained machine learning model to identify SATD issues in other projects and calculate the time spent to close different types of SATD issues, to investigate the priority of different types of SATD in different projects and compare results.

Implication 6: Requirement debt and test debt take longer time to be solved compared to code/design debt and documentation debt in the studied industrial projects. This observation still needs to be investigated and validated in other projects. Researchers could also use our SATD analysis approach to study the priority of different types of SATD in open-source projects.

In Section 4.2, we observed that most of the interview participants acknowledged the relevance and importance of the automatically-identified SATD items. Because we utilized publicly available datasets [6], [12], [16] to train machine learning models to identify SATD from the industrial projects, the results show that our models can be used to accurately identify SATD in industrial projects. We thus encourage researchers to use our approach to study SATD in industrial settings. To facilitate this, we share our scripts and trained machine learning model in the replication package[3]. Moreover, we noticed that most of the participants (five out of nine) indicated that it is difficult to determine the importance of SATD based solely on SATD statements. We suggest that researchers find the best method for presenting SATD, e.g., showing the SATD statements together with other contextual information to provide a broader picture.

Implication 7: Most of the interviewees acknowledged the automatically identified SATD. Researchers could use our trained machine learning model to further investigate SATD in other industrial projects.

In Section 4.3, we observed that it is common for source code comments or commit messages to reference related technical debt items in issues; the opposite is not common. Moreover, developers believed it could be very useful if related SATD is linked together. In the current state of the art, only the relation between code comments and commits has been used to study the repayment of SATD [11], [36], [37]. Thus, we argue that researchers and practitioners need to study the relations between SATD in different sources and build tools that aid in establishing traces between SATD in different sources and properly visualizing them.

Implication 8: Researchers and practitioners could further investigate the relations between SATD in different sources and build tools to automate and visualize traceability between SATD in different sources.

5.2 SATD Management Activities

In Section 4.4 we saw seven distinct cases that cause technical debt to be ignored and stay undocumented. Such implicit technical debt can have grave consequences for the development team. Researchers could look into this and propose solutions to avoid missing documentation for important technical debt. Moreover, as pointed out in Section 4.8, there are no concrete guidelines on technical debt documentation. Our work extended the scope of SATD documentation to three sources, namely code comments, commit messages, and issue trackers. Thus, we suggest that researchers and practitioners create comprehensive guidelines and develop tools to help technical debt documentation for different use cases in different sources based on our findings. Furthermore, the pros and cons of documenting technical debt in different sources, as described in Section 4.5, can also be of assistance in creating the aforementioned guidelines and tools, by building on the pros and working to avoid the cons.

Implication 9: Researchers and practitioners could create guidelines and build tools to assist in technical debt documentation in different sources.

In Section 4.6, we reported the triggers for paying back and not paying back SATD. However, the interviewees' opinions on *certain types of SATD* are contradictory. More specifically, some developers saw *test debt* as a trigger to repay SATD, while some others believed it should be ignored. This is caused by different participants' opinions about the importance of *test debt*. We encourage researchers and practitioners to create guidelines customized for specific organizations and teams about the importance and ways of repaying different types of SATD. The triggers identified in our study can act as an input for such guidelines. Moreover, researchers could also study how to eliminate the effects of SATD accumulation caused by triggers for not paying back technical debt. Finally, researchers could investigate the triggers in open-source projects, compare them, and create comprehensive lists of triggers for both open-source and industrial projects.

Implication 10: Researchers could investigate triggers for repaying and not repaying SATD, and create guidelines and tools for SATD prioritization based on those triggers. Besides, strategies to mitigate the effects of SATD accumulation caused by triggers for not paying back SATD need to be studied. Moreover, researchers could investigate triggers in open-source projects and compare results.

TABLE 13
Comparison Between Suggested Features and State-of-the-Art

Suggested Features		Related Papers
Automated SATD Identification From	Code Comments	[6], [7], [12], [14], [15], [38], [39], [40], [41], [42], [43], [44], [45], [46], [47], [48], [49], [50], [51], [52], [53], [54], [55], [56], [57], [58], [59], [60], [61], [62], [63], [64], [65], [66]
	Issue Trackers	[3], [12], [16]
	Commit Messages	[12]
	Pull Requests	[12]
Automated Differentiation Between Fixed and Unfixed SATD		-
Automated Tracing Between SATD	in Different Sources	[11], [12], [36], [37]
	and Code	-
	and Related Development Tasks	-
Automated SATD Prioritization		[9], [67], [68]
Automated Identification of SATD Risk		-
Automated Estimation of Benefits to Solve SATD		[8]
Automated Estimation of Cost to Solve SATD		[9], [69]
Automated SATD Solution Suggestion		[70]

In Section 4.7, we reported the practices used to support SATD prioritization. These practices are not acknowledged by all the interviewees.; for example, in contrast to the practice of using issue types (e.g., *bug*, *task*, and *backlog item*) to set priorities of SATD, one participant did not find significant differences between issue types. This is due to the organization not using such issue types in a standardized manner. Researchers should study and propose such practices for SATD prioritization that can be standardized across organizations.

Implication 11: Researchers could study and use the practices to support SATD prioritization by embedding them in tools or processes.

In Section 4.7, we also report on two strategies for efficiently paying back SATD. We found that adopting such strategies heavily depends on developers' personal opinions and discussions with their colleagues. Researchers could investigate how much effort (i.e., technical debt interest) is saved by using these strategies and how to automatically group SATD and other tasks for higher repayment efficiency.

Implication 12: Researchers could investigate the efficiency of SATD repayment strategies and build tools to help developers utilize these strategies.

5.3 SATD Management Improvement

In Section 4.8, we list the challenges faced by interviewees when dealing with SATD. We suggest that researchers examine the impact of the listed challenges, and propose strategies and tools to tackle them. Some of the challenges can be addressed directly within the development team, e.g., convincing developers not to introduce technical debt when not necessary. Practitioners could discuss these challenges in their team and decide which they can tackle and how.

Implication 13: Researchers can evaluate the impact of challenges, and propose strategies and tools to tackle them. Practitioners can also review them and discuss which ones they can address.

The participating developers have come up with various features they would like to see in SATD management tools (see Section 4.9). This begs the question of whether the current research work can already offer some of these features. To offer a preliminary answer, we checked research publications in Google Scholar, using the search string *"self-admitted technical debt"*. This resulted in 72 papers that deal with SATD; we then read their abstract and full text to filter out papers that are irrelevant to the required features (see Section 4.9). The resulting set of 45 related papers is listed in Table 13. As we can see, the majority of these papers focus on automatically identifying SATD from source code comments, while there has been no work investigating *automated differentiation between fixed and unfixed SATD*, *automated tracing between SATD and code or related development tasks*, and *automated identification of SATD risk*. For the other proposed features, some related studies exist, but these are not fully supported yet or require better support. For example, there is a study relevant to *automated SATD solution suggestion* [70], but it is only able to suggest one of six predefined SATD repayment strategies (e.g., changing API calls or changing return statements). Furthermore, the usefulness of each feature and the difficulties of implementing each feature are different. We recommend that researchers and practitioners evaluate the added value of each proposed feature, implement tools including the most important features, and test the effectiveness of such tools.

Implication 14: Most research works in SATD management tools focus on automatically identifying SATD from source code comments. Researchers should investigate other features required by the interviewees, such as automated differentiation between fixed and unfixed SATD, automated tracing between SATD and code or related development tasks, and automated identification of SATD risk.

6 Threats to Validity

6.1 Construct Validity

Threats to construct validity concern the correctness of operational measures for the studied subjects. One of the threats to construct validity in the study concerns the potentially different interpretations of discussed topics between interviewees and researchers. Because we focus on SATD in this study and most of the interviewees were not familiar with this concept, we tried to avoid misunderstanding this term by 1) asking for their understanding of technical debt; 2) asking them to give some examples of it to make sure they have the correct comprehension; 3) reminding them, during the interviews, that we focus on technical debt that is documented in different sources to avoid confusion. The responses we received from the interviewees regarding how they understand technical debt and the examples of technical debt they gave, attest to a correct understanding of the concept by all interviewees. For instance, one of the interviewees gave the following examples of technical debt: *"for me, technical debt can be multiple things; it can be a design that works now but might be problematic in the future. It also covers shortcuts you take in the code. When you have a bug, you create a quick workaround, so your customers can continue, which you know actually [...] needs a solid fix so that in the future it will remain intact. I think technical debt also builds automatically if certain techniques are no longer maintained, so you need to switch to a new one because you cannot upgrade it..."*.

Another threat to construct validity is related to the possible reluctance of interviewees to express negative opinions on their organization or admit mistakes made in the past. To minimize this threat, we emphasized that we are bound by a confidentiality agreement, and no sensitive or personal information would be revealed after the interviews.

6.2 Reliability

Reliability is concerned with the bias that researchers may induce in data collection and analysis. One threat to reliability could be different results obtained from work artifacts' analysis. Specifically, when extracting source code comments from studied projects, because the projects could use multiple programming languages, defining and using simple heuristic rules might not be able to extract all comments from different programming languages. To reliably and accurately extract code comments, instead of defining such heuristic rules ourselves, we chose to use a third-party library (*CommnetParser*), which supports multiple programming languages, such as C++, Go, Python, Java, XML, and Ruby. The studied projects mainly use C++ and XML files. We manually verified the correctness of the extracted comments with this library before collecting the data.

Another threat to reliability could be the impact of researchers' opinions on interviewees. To mitigate this threat, all authors followed a specific protocol for the interviews which is included in the replication package[3]. Besides, at least two authors attended each interview, to ensure that one interviewer did not bias the questions asked.

Furthermore, another threat to reliability could come from the selection of the 15 SATD items for interviews. As we can see in Table 11, the percentages of requirement, documentation, and test debt are relatively low (always below 15%). If we randomly collected five SATD items from each source, certain types of SATD items would likely be missing in the 15 SATD samples. This could result in the sample misrepresenting the SATD types. To mitigate this risk, we selected three or four SATD items for code/design debt and one or two SATD items for other types of SATD for each source (e.g., code comments) based on the SATD type proportion (see Table 11). Thus, for the 15 SATD sample items, we have ten code/design debt items, one requirement debt item, two documentation debt items, and two test debt items, which include different types of SATD items and follow the distribution of different types of SATD. Therefore, we consider this threat as, at least partially, mitigated.

The last threat to reliability comes from analyzing the interview data. To minimize this threat, the first and second authors carried out the *Constant Comparative* analysis process [25], [26] independently; in case of discrepancy, we compared and discussed the results until we were able to reach an agreement.

6.3 External Validity

Threats to external validity concern the generalizability of the results. In this study, we analyzed work artifacts and conducted interviews in a large software company in the embedded systems industry. Our findings may, to some extent, generalize to other industrial projects of this application domain and of similar size and complexity. In several instances, such as time to close SATD and non-SATD issues and the percentage of SATD in code comments, we have demonstrated how our findings support previous studies. However, we can not claim any further generalization.

7 Conclusion and Future Work

In this paper, we analyzed SATD in industrial projects using machine learning techniques and conducted 12 interviews to understand: 1) characteristics of SATD in industrial projects; 2) developers' attitudes towards identified SATD and statistics; 3) triggers to introduce and repay SATD; 4) relations between SATD in source code comments, issues, and commits; 5) practices used to manage SATD; 6) challenges and tooling ideas for SATD management.

The results present characteristics of SATD in industrial projects and shed light on developers' opinions on SATD management and tooling support. This promotes future studies in this area, targeting to support developers in terms of SATD introduction, traceability, prioritization, repayment, and tool support.

In the future, based on the results of this work, we plan to characterize SATD in more industrial projects to further validate the observation that industrial developers tend to admit less SATD in code comments and more SATD in issues and commits in comparison with open-source developers, and explore the reasons behind it. Moreover, we plan to conduct a large-scale study to analyze the closing time and closing rate between different types of SATD in open-source projects, in order to investigate the priority differences between different types of SATD and non-SATD issues. Furthermore, we plan to study the relations between SATD in different sources and create an automatic approach to identify related SATD items. Additionally, we plan to create SATD documentation and repayment guidelines and evaluate them with software practitioners. Next, we plan to investigate the number of SATD items that are documented or repaid in terms of the different reasons to document SATD or pay back SATD. Lastly, we plan to build a tool that supports SATD management for software practitioners based on the desired features described in Section 4.9.

REFERENCES

[1] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing technical debt in software engineering (Dagstuhl seminar 16162)," *Dagstuhl Rep.*, vol. 6, no. 4, pp. 110–138, 2016.

[2] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2014, pp. 91–100.

[3] K. Dai and P. Kruchten, "Detecting technical debt through issue trackers," in *Proc. 5th Int. Workshop Quantitative Approaches Softw. Qual.*, 2017, pp. 59–65.

[4] Y. Li, M. Soliman, and P. Avgeriou, "Identification and remediation of self-admitted technical debt in issue trackers," in *Proc. IEEE 46th Euromicro Conf. Softw. Eng. Adv. Appl.*, 2020, pp. 495–503.

[5] F. Zampetti, G. Fucci, A. Serebrenik, and M. Di Penta, "Self-admitted technical debt practices: A comparison between industry and open-source," *Empir. Softw. Eng.*, vol. 26, no. 6, pp. 1–32, 2021.

[6] E. da Silva Maldonado, E. Shihab, and N. Tsantalis, "Using natural language processing to automatically detect self-admitted technical debt," *IEEE Trans. Softw. Eng.*, vol. 43, no. 11, pp. 1044–1062, Nov. 2017.

[7] X. Ren, Z. Xing, X. Xia, D. Lo, X. Wang, and J. Grundy, "Neural network-based detection of self-admitted technical debt: From performance to explainability," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 3, pp. 1–45, 2019.

[8] Y. Kamei, E. d. S. Maldonado, E. Shihab, and N. Ubayashi, "Using analytics to quantify interest of self-admitted technical debt," in *Proc. 4th Int. Workshop Quantitative Approaches Softw. Qual. 1st Int. Workshop Tech. Debt Analytics*, 2016, pp. 68–71.

[9] S. Mensah, J. Keung, J. Svajlenko, K. E. Bennin, and Q. Mi, "On the value of a prioritization scheme for resolving self-admitted technical debt," *J. Syst. Softw.*, vol. 135, pp. 37–54, 2018.

[10] E. d. S. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik, "An empirical study on the removal of self-admitted technical debt," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2017, pp. 238–248.

[11] F. Zampetti, A. Serebrenik, and M. Di Penta, "Was self-admitted technical debt removal a real removal? An in-depth perspective," in *Proc. IEEE/ACM 15th Int. Conf. Mining Softw. Repositories*, 2018, pp. 526–536.

[12] Y. Li, M. Soliman, and P. Avgeriou, "Automatic identification of self-admitted technical debt from four different sources," 2022, *arXiv:2202.02387*.

[13] E. d. S. Maldonado and E. Shihab, "Detecting and quantifying different types of self-admitted technical debt," in *Proc. IEEE 7th Int. Workshop Manag. Tech. Debt*, 2015, pp. 9–15.

[14] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li, "Identifying self-admitted technical debt in open source projects using text mining," *Empir. Softw. Eng.*, vol. 23, no. 1, pp. 418–451, 2018.

[15] X. Wang, J. Liu, L. Li, X. Chen, X. Liu, and H. Wu, "Detecting and explaining self-admitted technical debts with attention-based neural networks," in *Proc. IEEE/ACM 35th Int. Conf. Autom. Softw. Eng.*, 2020, pp. 871–882.

[16] Y. Li, M. Soliman, and P. Avgeriou, "Identifying self-admitted technical debt in issue tracking systems using machine learning," *Empir. Softw. Eng.*, vol. 27, Jul. 2022, Art. no. 131.

[17] R. van Solingen, V. Basili, G. Caldiera, and H. D. Rombach, "Goal Question Metric (GQM) approach," in *Encyclopedia of Software Engineering*. Hoboken, NJ, USA: Wiley, Jan. 2002, pp. 528–532.

[18] A. Yamashita, M. Zanoni, F. A. Fontana, and B. Walter, "Inter-smell relations in industrial and open source systems: A replication and comparative analysis," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2015, pp. 121–130.

[19] A. Bosu, J. C. Carver, C. Bird, J. Orbeck, and C. Chockley, "Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at Microsoft," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 56–75, Jan. 2017.

[20] G. Sierra, E. Shihab, and Y. Kamei, "A survey of self-admitted technical debt," *J. Syst. Softw.*, vol. 152, pp. 70–82, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121219300457

[21] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering: Guidelines and Examples*. Hoboken, NJ, USA: Wiley, 2012.

[22] T. C. Lethbridge, S. E. Sim, and J. Singer, "Studying software engineers: Data collection techniques for software field studies," *Empir. Softw. Eng.*, vol. 10, no. 3, pp. 311–341, 2005.

[23] M. DeJonckheere and L. M. Vaughn, "Semistructured interviewing in primary care research: A balance of relationship and rigour," *Fam. Med. Community Health*, vol. 7, no. 2, 2019, Art. no. e000057.

[24] I. Seidman, *Interviewing as Qualitative Research: A Guide for Researchers in Education and the Social Sciences*. New York, NY, USA: Teachers College Press, 2006.

[25] A. Strauss and J. Corbin, *Basics of Qualitative Research*. Thousand Oaks, CA, USA: Sage Publications, 1990.

[26] K.-J. Stol, P. Ralph, and B. Fitzgerald, "Grounded theory in software engineering research: A critical review and guidelines," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 120–131.

[27] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *Ann. Math. Statist.*, vol. 18, pp. 50–60, 1947.

[28] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *J. Educ. Behav. Statist.*, vol. 25, no. 2, pp. 101–132, 2000.

[29] S. Bellomo, R. L. Nord, I. Ozkaya, and M. Popeck, "Got technical debt? Surfacing elusive technical debt in issue trackers," in *Proc. IEEE/ACM 13th Work. Conf. Mining Softw. Repositories*, 2016, pp. 327–338.

[30] Atlassian Corporation PLC, "What are issue types?" 2022. [Online]. Available: https://support.atlassian.com/jira-cloud-administration/docs/what-are-issue-types/

[31] Microsoft Corporation, "Define features and epics in azure boards to organize your product and portfolio backlogs," 2022. [Online]. Available: https://docs.microsoft.com/en-us/azure/devops/boards/backlogs/define-features-epics?view=azure-devops&tabs=scrum-process/

[32] Atlassian Corporation PLC, "Defining priority field values," 2022. [Online]. Available: https://confluence.atlassian.com/adminjiraserver/defining-priority-field-values-938847101/

[33] B. Graaf, M. Lormans, and H. Toetenel, "Embedded software engineering: The state of the practice," *IEEE Softw.*, vol. 20, no. 6, pp. 61–69, Nov./Dec. 2003.

[34] L. Xavier, F. Ferreira, R. Brito, and M. T. Valente, "Beyond the code: Mining self-admitted technical debt in issue tracker systems," in *Proc. 17th Int. Conf. Mining Softw. Repositories*, 2020, pp. 137–146.

[35] C. Ebert and C. Jones, "Embedded software: Facts, figures, and future," *Computer*, vol. 42, no. 4, pp. 42–52, Apr. 2009.

[36] M. Iammarino, F. Zampetti, L. Aversano, and M. Di Penta, "Self-admitted technical debt removal and refactoring actions: Co-occurrence or more?," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2019, pp. 186–190.

[37] M. Iammarino, F. Zampetti, L. Aversano, and M. Di Penta, "An empirical study on the co-occurrence between refactoring actions and self-admitted technical debt removal," *J. Syst. Softw.*, vol. 178, 2021, Art. no. 110976.

[38] M. A. d. Farias, J. A. Santos, M. Kalinowski, M. Mendonça, and R. O. Spínola, "Investigating the identification of technical debt through code comment analysis," in *Proc. Int. Conf. Enterprise Inf. Syst.*, 2016, pp. 284–309.

[39] S. Wattanakriengkrai, R. Maipradit, H. Hata, M. Choetkiertikul, T. Sunetnanta, and K. Matsumoto, "Identifying design and requirement self-admitted technical debt using n-gram IDF," in *Proc. IEEE 9th Int. Workshop Empir. Softw. Eng. Pract.*, 2018, pp. 7–12.

[40] Z. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, "Satd detector: A text-mining-based self-admitted technical debt detection tool," in *Proc. 40th Int. Conf. Softw. Eng.: Companion Proc.*, 2018, pp. 9–12.

[41] A. F. de O. Passos, M. A. de Freitas Farias, M. G. de Mendonça Neto, and R. O. Spínola, "A study on identification of documentation and requirement technical debt through code comment analysis," in *Proc. 17th Braz. Symp. Softw. Qual.*, 2018, pp. 21–30.

[42] M. Yan, X. Xia, E. Shihab, D. Lo, J. Yin, and X. Yang, "Automating change-level self-admitted technical debt determination," *IEEE Trans. Softw. Eng.*, vol. 45, no. 12, pp. 1211–1229, Dec. 2019.

[43] S. Wattanakriengkrai et al., "Automatic classifying self-admitted technical debt using n-gram IDF," in *Proc. IEEE 26th Asia-Pacific Softw. Eng. Conf.*, 2019, pp. 316–322.

[44] J. Flisar and V. Podgorelec, "Identification of self-admitted technical debt using enhanced feature selection based on word embedding," *IEEE Access*, vol. 7, pp. 106475–106494, 2019.

[45] Z. Guo et al., "MAT: A simple yet strong baseline for identifying self-admitted technical debt," 2019, *arXiv:1910.13238*.

[46] L. Rantala and M. Mäntylä, “Predicting technical debt from commit contents: Reproduction and extension with automated feature selection,” *Softw. Qual. J.*, vol. 28, no. 4, pp. 1551–1579, 2020.

[47] A. Alhefdhi, H. K. Dam, Y. S. Nugroho, H. Hata, T. Ishio, and A. Ghose, “A framework for self-admitted technical debt identification and description,” 2020, *arXiv:2012.12466*.

[48] M. A. de Freitas Farias, M. G. de MendonçaM. Neto Kalinowski, and R. O. Spínola, “Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary,” *Inf. Softw. Technol.*, vol. 121, 2020, Art. no. 106270.

[49] R. Maipradit, C. Treude, H. Hata, and K. Matsumoto, “Wait for it: Identifying “on-hold” self-admitted technical debt,” *Empir. Softw. Eng.*, vol. 25, no. 5, pp. 3770–3798, 2020.

[50] R. Maipradit et al., “Automated identification of on-hold self-admitted technical debt,” in *Proc. IEEE 20th Int. Work. Conf. Source Code Anal. Manipulation*, 2020, pp. 54–64.

[51] R. M. Santos, I. M. Santos, M. C. R. Júnior, and M. G. de Mendonça Neto, “Long term-short memory neural networks and word2vec for self-admitted technical debt detection,” in *Proc. 22nd Int. Conf. Enterprise Inf. Syst.*, 2020, pp. 157–165.

[52] Z. Yu, F. M. Fahid, H. Tu, and T. Menzies, “Identifying self-admitted technical debts with jitterbug: A two-step approach,” *IEEE Trans. Softw. Eng.*, vol. 48, no. 5, pp. 1676–1691, May 2022.

[53] L. Rantala, M. Mäntylä, and D. Lo, “Prevalence, contents and automatic detection of KL-SATD,” in *Proc. IEEE 46th Euromicro Conf. Softw. Eng. Adv. Appl.*, 2020, pp. 385–388.

[54] X. Chen, D. Yu, X. Fan, L. Wang, and J. Chen, “Multiclass classification for self-admitted technical debt based on XGBoost,” *IEEE Trans. Rel.*, vol. 71, no. 3, pp. 1309–1324, Sep. 2022.

[55] Z. Guo et al., “How far have we progressed in identifying self-admitted technical debts? A comprehensive empirical study,” *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 4, pp. 1–56, 2021.

[56] R. M. Santos, I. M. Santos, M. C. Júnior, and M. Mendonça, “Evaluating a LSTM neural network and a word2vec model in the classification of self-admitted technical debts and their types in code comments,” in *Proc. Int. Conf. Enterprise Inf. Syst.*, 2021, pp. 542–559.

[57] D. Yu, L. Wang, X. Chen, and J. Chen, “Using BiLSTM with attention mechanism to automatically detect self-admitted technical debt,” *Front. Comput. Sci.*, vol. 15, no. 4, pp. 1–12, 2021.

[58] I. Sala, A. Tommasel, and F. Arcelli Fontana, “DebtHunter: A machine learning-based approach for detecting self-admitted technical debt,” in *Proc. Eval. Assessment Softw. Eng.*, 2021, pp. 278–283.

[59] K. Zhu, M. Yin, and Y. Li, “Detecting and classifying self-admitted of technical debt with CNN-BiLSTM,” *J. Phys.*, vol. 1955, no. 1, 2021, Art. no. 012102.

[60] S. Phaithoon et al., “FixMe: A GitHub bot for detecting and monitoring on-hold self-admitted technical debt,” in *Proc. IEEE/ACM 36th Int. Conf. Autom. Softw. Eng.*, 2021, pp. 1257–1261.

[61] T. Xiao et al., “Characterizing and mitigating self-admitted build debt,” 2021, *arXiv:2102.09775*.

[62] J. Yu, K. Zhao, J. Liu, X. Liu, Z. Xu, and X. Wang, “Exploiting gated graph neural network for detecting and explaining self-admitted technical debts,” *J. Syst. Softw.*, vol. 187, 2022, Art. no. 111219.

[63] H. Tu and T. Menzies, “DebtFree: Minimizing labeling cost in self-admitted technical debt identification using semi-supervised learning,” *Empir. Softw. Eng.*, vol. 27, no. 4, pp. 1–37, 2022.

[64] B. Russo, M. Camilli, and M. Mock, “WeakSATD: Detecting weak self-admitted technical debt,” 2022, *arXiv:2205.02208*.

[65] E. A. Alomar et al., “SATDBailiff-mining and tracking self-admitted technical debt,” *Sci. Comput. Program.*, vol. 213, 2022, Art. no. 102693.

[66] G. Zhuang, Y. Qu, L. Li, X. Dou, and M. Li, “An empirical study of gradient-based explainability techniques for self-admitted technical debt detection,” *J. Internet Technol.*, vol. 23, no. 3, pp. 631–641, 2022.

[67] F. Zampetti, C. Noiseux, G. Antoniol, F. Khomh, and M. Di Penta, “Recommending when design technical debt should be self-admitted,” in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2017, pp. 216–226.

[68] B. S. de Lima, R. E. Garcia, and D. M. Eler, “Toward prioritization of self-admitted technical debt: An approach to support decision to payment,” *Softw. Qual. J.*, vol. 30, pp. 729–755, 2022.

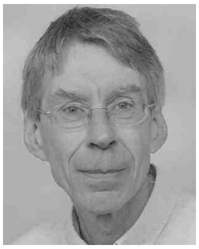
[69] S. Mensah, J. Keung, M. F. Bosu, and K. E. Bennin, “Rework effort estimation of self-admitted technical debt,” *Proc. CEUR Workshop Proc.*, vol. 1771, pp. 72–75, 2016.

[70] F. Zampetti, A. Serebrenik, and M. Di Penta, “Automatically learning patterns for self-admitted technical debt removal,” in *Proc. IEEE 27th Int. Conf. Softw. Anal., Evol. Reengineering*, 2020, pp. 355–366.

Yikun Li received the BE degree in software engineering from Southeast University, Nanjing, China, in 2015, and the MSc degree in artificial intelligence from the University of Groningen, Groningen, The Netherlands, in 2019. He is currently working toward the PhD degree with Bernoulli Institute for Mathematics, Computer Science, and Artificial Intelligence. His research interests include artificial intelligence for software engineering, technical debt management, and mining software repositories.

Mohamed Soliman received the bachelor’s and master’s degree in computer and systems engineering from Ain-Shams University, Cairo, Egypt, in 2005 and 2010, respectively, and the PhD degree from the University of Hamburg, Hamburg, Germany, in 2018, with a thesis entitled Acquiring Architecture Knowledge for Technology Design Decisions. Since September 2019, he has been an FSE Postdoc Fellow with the University of Groningen, Groningen, The Netherlands. His research focuses on software architectural knowledge capturing and reuse. He was a Software Developer and Technical Leader for eight years in different software companies, including multinational companies, e.g., Hewlett Packard, where he designed and implemented applications using different technologies and methods.

Paris Avgeriou is currently a professor of software engineering with the University of Groningen, Groningen, The Netherlands. His research focuses on software architecture, with strong emphasis on architecture modeling, knowledge, evolution, analytics, and technical debt. He is the Editor-in-Chief of the *Journal of Systems and Software* as well as an Associate Editor for IEEE SOFTWARE. He is the Vice Chair of the Dutch National Association for Software Engineering (VERSEN) and is on the board of the Dutch research school IPA. He has coorganized several international conferences, such as ICSME, ECSA, ICSA, and Tech Debt, and was on their steering committees. He champions the evidence-based paradigm in Software Engineering research and works toward closing the gap between industry and academia.



Lou Somers received the doctorate degree in theoretical physics from the University of Nijmegen, Nijmegen, The Netherlands. He is currently an associate professor with the Department of Mathematics and Computer Science, Eindhoven University of Technology (TU/e), Eindhoven, The Netherlands, and a senior staff Member of Canon Production Printing (retired). He has been involved in many national and European R&D projects. His courses revolve around information systems and databases, software engineering and project management. His research interests include performance modeling, design optimization, scheduling and management of complex, and software-intensive systems.