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

On the Documentation of Self-Admitted Technical Debt in Issues

Laerte Xavier · João Eduardo Montandon · Fabio Ferreira · Rodrigo Brito · Marco Tulio Valente

Received: date / Accepted: date

Abstract Self-Admitted Technical Debt (SATD) is a particular case of Technical Debt (TD) in which developers rely on source code comments (SATD-C) or labeled issues (SATD-I) to report their sub-optimal technical solutions. In this paper, we first explore a sample of 286 SATD-I instances collected from five open source projects, including Microsoft Visual Studio Code and GitLab Community Edition. We show that in 45% of the studied issues TD was introduced to ship earlier (*i.e.,* to deliver faster), and in almost 60% it refers to DESIGN flaws. Besides, we report that most developers pay SATD-I to reduce its costs or interests (66%). To complement the previous exploratory results, we investigate the adoption of tools to support SATD-I documentation. For that, we build a large-scale dataset of 72K SATD-C and 20K SATD-I instances, extracted from 190 GitHub projects. We also implement a prototype tool, called ADMITD, to automatically report SATD-C as GitHub issues. We use this dataset and tool to reveal that developers are not interested in the automatic transformation of SATD-C in SATD-I. Moreover, we show that it might not be feasible to create a tool to recommend explicit links between SATD-C and SATD-I instances.

Keywords Self-Admitted Technical Debt · Issue Tracker · GitHub · GitLab

Laerte Xavier
Department of Computer Science, UFMG, Brazil, E-mail: laertexavier@dcc.ufmg.br

João Eduardo Montandon
Technical College (COLTEC), UFMG, Brazil, E-mail: joao.montandon@dcc.ufmg.br

Fabio Ferreira
Center of Informatics, IF SUDESTE MG, Brazil, E-mail: fabio.ferreira@ifsudestemg.edu.br

Rodrigo Brito
Department of Computer Science, UFMG, Brazil, E-mail: britorodrigo@dcc.ufmg.br

Marco Tulio Valente
Department of Computer Science, UFMG, Brazil, E-mail: mtov@dcc.ufmg.br

1 Introduction

In software development, the *done is better than perfect* maxim reflects the inevitable trade-off between keeping software quality and releasing on time. In this context, the Technical Debt (TD) metaphor—first framed by Cunningham in 1992 [7]—refers to the unavoidable maintenance and evolution costs of such *not-quite-right* solutions. Several circumstances drive developers to assume these debts, such as deadline pressure, existing low quality code, and poor software process [28]. In fact, the term has been widely adopted since its definition [24] and became subject of various studies, mostly regarding its identification [30,60,39,1], management [9,45,40,29,50], and assessment [59, 49,22,4,43].

Self-admitted technical debt (SATD) is a particular case of TD where developers explicitly document their sub-optimal implementation decisions [37, 3,33,57]. However, to our knowledge, the majority of SATD studies rely on source comments to identify SATD instances. Particularly, they search for specific TD-related terms in source code comments—such as *fixme*, *TODO*, and *hack*. By contrast, developers can also report technical debt out of the source code, by creating issues in tracking systems documenting their sub-optimal decisions [4,8,26]. To document the debt, they label these issues with terms such as *technical debt* or *debt*. An example is presented in Figure 1. The figure shows an issue from GitLab requesting the removal of duplicated code (in this case, a permission variable). As we can see, it received a *technical debt* label.

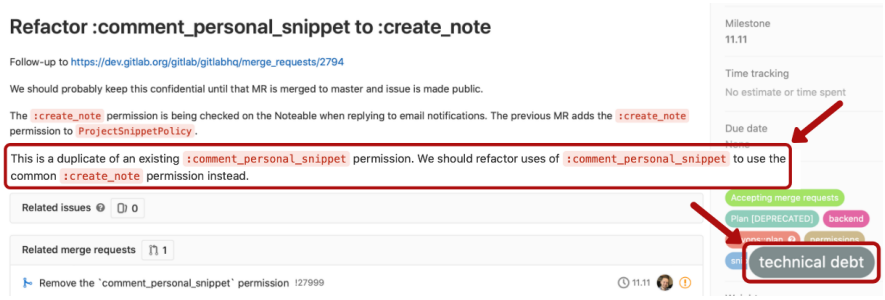

Fig. 1 Example of SATD in a GitLab's issue.

In this paper, we focus on two types of SATD[1]:

1. SATD documented using source code comments (SATD-C), which has been extensively studied in the past;
2. SATD documented using issues (SATD-I), which is less studied in the literature. One exception is a previous conference paper where we started to study this type of TD [52]. In Section 1.1, we describe our key findings in

[1] In fact, SATD can also be documented in other artifacts, such as commit messages, wikis, forum discussions, etc.

this initial study. Then, in Section 1.2, we explain how the current work extends and complements our first conference paper.

1.1 Initial Study

In a previous work [52], we explored SATD-I through the analysis of paid instances, *i.e.,* issues documenting TD that were successfully closed by developers. For that, we collected and characterized an initial dataset containing 286 SATD-I instances from five relevant open-source systems. We used this dataset to answer three research questions:

RQ1. What types of technical debt are paid in SATD-I*?* We found that almost 60% of the SATD-I in our initial dataset is related to DESIGN flaws. Other types of SATD-I include UI (10%), TESTS (9%), and PERFORMANCE (8%).

RQ2. Why do developers introduce SATD-I*?* About 45% of the surveyed developers (12/30 developers) indicate that SATD-I was introduced to ship earlier. In other words, to deliver faster, developers consciously added shortcuts in their code which were expected to be fixed in the future. In nine cases, the debt was later admitted.

RQ3. Why do developers pay SATD-I*?* We reveal that most developers pay SATD-I to reduce its interest, and to have a clean code. Moreover, we also found that SATD-I is commonly responsible for slowing down code evolution.

Therefore, in this initial study, we confirmed that **technical debt is also documented using issues** and we characterized this practice in a set of well-known open source projects.

1.2 Extended Study

In this extended journal manuscript, we turned our focus on **tools to support developers in documenting technical debt using issues**. Specifically, we evaluate the feasibility of tools to deal with SATD in two distinct aspects:

- To automatically create issues to document TD concerns expressed in source code comments.
- To automatically create links between SATD-C and SATD-I instances that are related.

We first create a large-scale dataset of 20,265 issues marked with a TD-related label (SATD-I) and 72,669 code comments documenting TD (SATD-C), extracted from 190 GitHub projects. We use this extended dataset to answer two additional research questions:

RQ4. Are developers interested in tools to create issues from SATD-C*?* First, we implement a prototype tool for automatically generating issues from SATD comments as GitHub issues, called ADMITD. We equipped this tool with a

set of heuristics to enhance the acceptance of its recommendations. Then, we validate our results with the principal developers of ten open source projects.

RQ5. Do developers refer to SATD-I *in* SATD-C*?* Next, we search for explicit links between SATD comments and SATD issues. Specifically, we intend to check whether developers mention SATD-I IDs or URLs in SATD-C instances, such as in following code comment from COCKROACHDB/COCKROACH:

```
// TODO(irfansharif): We should reconsider usage of
NodeLivenessStatus. (...) See #50707 for more details.
```

The existence of such links would mean that SATD-C and SATD-I are somehow related (at least in some cases). As a consequence, it would be feasible to implement tools that detect related SATD-C and SATD-I instances, although the SATD-C does not include a reference to the associated SATD-I. In such cases, the tool could recommend the creation of the link.

1.3 Paper Structure

In Section 2, we detail our initial study as follows: we dedicate Section 2.1 to present the dataset used in this study. Section 2.2 presents the classification study performed to answer RQ1. In Section 2.3, we detail the survey conducted to answer RQ2 and RQ3. Next, we dedicate Section 3 to present the extended study as follows: first, we detail our new dataset in Section 3.1. In Section 3.2, we present ADMITD tool and answer RQ4. Section 3.3 provides answers for RQ5. Finally, Sections 4, 5 and 6 present implications, related work and conclusions of this work, respectively.

2 Initial Study

In this section, we present our initial study on the documentation of technical debt in issues [52]. We first collect an initial dataset containing 286 SATD-I instances (Section 2.1). Next, we conduct two studies to investigate (i) the types of SATD commonly reported in issues (Section 2.2); and (ii) the motivations that drive developers to introduce and pay such debts (Section 2.3).

2.1 Initial Dataset

We collect an initial dataset containing SATD-I instances from five open-source systems: GitLab and four GitHub-based systems. We selected GitLab because it is a well-known platform that supports a git-based version control service and also a CI/CD pipeline. Moreover, we had previous knowledge—from our research in the area—on GitLab's practice to label TD-related issues. In this section, we explain how we selected the GitLab issues used in these initial studies (Section 2.1.1). We also explain how we mined and selected four GitHub

projects that follow a practice similar to the one used by GitLab, *i.e.,* they also use specific labels on issues that discuss technical debt (Section 2.1.2). Finally, we provide a quantitative overview of this first dataset (Section 2.1.3).

2.1.1 GitLab CE

Differently from GitHub, GitLab's source code is publicly available in the platform, *i.e.,* GitLab is an open source project that is developed and maintained using its own services. In fact, the project has two editions: Community (CE) and Enterprise (EE). The latter is a commercial version and the former is an open-source edition. GitLab's development happens on both repositories, which are continuously synchronized. Since they are public, we rely on issues from GitLab CE.

First, we used GitLab's REST API to select all issues with a *technical debt* label that were closed in the six months before our search. We only selected closed issues because our primary focus is to explore technical debt that was paid. Moreover, we restricted the selection to the last six months to increase the chances of receiving answers in the survey that we performed with GitLab's developers—and also to increase the confidence on the survey answers (see Section 2.3).

After applying the described selection criteria, we found 188 issues. The first author of this paper carefully inspected each one and removed 65 issues (34.6%) that represent duplicated issues, issues that only include discussions, and ignored issues. He also verified that no issue was automatically tagged by a static analysis tool. For example, he discarded an issue where the developer concluded that:

Heh, this is a duplicate of gitlab-ee#3861 (closed), which is being worked on right now by @cablett. I'll close it![2]

Besides, during the classification of the 123 remaining issues, we identified and removed six issues that only request new features, bug corrections, or build failure fixes (*i.e.,* despite having a technical debt label, they are not related with TD). For example, we discarded an issue that reports:

Commit count and other project statistics are incorrect.[3]

After this step, we selected 117 SATD-I instances from GitLab.

2.1.2 GitHub-based Projects

We also searched for SATD-I in open-source GitHub systems. We restricted the search to the top-5,000 most starred GitHub repositories, since stars is a commonly used proxy for the popularity of GitHub repositories [5, 42]. We used GitHub's GraphQL API to search for all issues of such repositories that

[2] `https://gitlab.com/gitlab-org/gitlab-ce/issues/34659`
[3] `https://gitlab.com/gitlab-org/gitlab-ce/issues/44726`

were closed in the previous six months—due to the same reasons explained for GitLab—and that include one of the following labels: *technical debt*, *Technical Debt*, and *debt*. In other words, we decided to select SATD issues by using this set of labels as a precise sign of the presence of such discussions in the issue. As a result, we found 252 issues in 23 repositories. However, we decided to discard 34 issues from 19 repositories with less than 10 issues. The rationale was to focus the study on repositories where labelling issues denoting TD is a common practice.

As previously conducted for GitLab issues, the first author of this paper inspected all 218 issues selected in GitHub (*i.e.,* $252-34$ issues) and discarded 49 issues (22%) that do not have a clear indication of representing an actual case of TD payment. In the end, 169 SATD-I instances coming from four GitHub repositories were selected for inclusion in our dataset.

2.1.3 Dataset Characterization

Table I shows the name of the systems in our dataset, the platform where they are hosted (GH refers to GitHub and GL refers to GitLab), the tags they use to denote SATD-I and the number of issues selected in each system. As we can observe, there is a concentration of issues in GitLab-CE (40.9%) and on MICROSOFT/VSCODE (46.2%), which is the popular IDE from Microsoft whose development history is publicly available on GitHub. The remaining SATD-I instances come from INFLUXDATA/INFLUXDB (7.3%), MIRUMEE/SALEOR (3.5%), and NEXTCLOUD/SERVER (2.1%). INFLUXDATA/INFLUXDB is a platform for time series storage and manipulation. MIRUMEE/SALEOR is an open source eCommerce platform, and NEXTCLOUD/SERVER is a framework for communicating with Nextcloud (a service for hosting files on the cloud).

Table 1 Selected repositories.

Repository	**Plat.**	**Tag**	**Satd-I**	**%**
MICROSOFT/VSCODE	GH	*debt*	132	46.2%
GITLAB/GITLAB-CE	GL	*technical debt*	117	40.9%
INFLUXDATA/INFLUXDB	GH	*Technical Debt*	21	7.3%
MIRUMEE/SALEOR	GH	*technical debt*	10	3.5%
NEXTCLOUD/SERVER	GH	*technical debt*	6	2.1%
Total			286	100%

Figure 2 shows violin plots comparing the issues selected in the study with all other issues.[4] We can see that SATD-I takes more time to be closed (16.7 vs 4.0 days, median values). They also have more comments (5 vs 3 comments) and labels (3 vs 2 labels). These observations are statistically confirmed by

[4] We acknowledge that some of the non-selected issues might also refer to TD (for example, developers may have forgotten to label them as such). Despite that, this fact does not invalidate our key goal in the paper, which is studying a valid and large sample of issues that explicitly document TD.

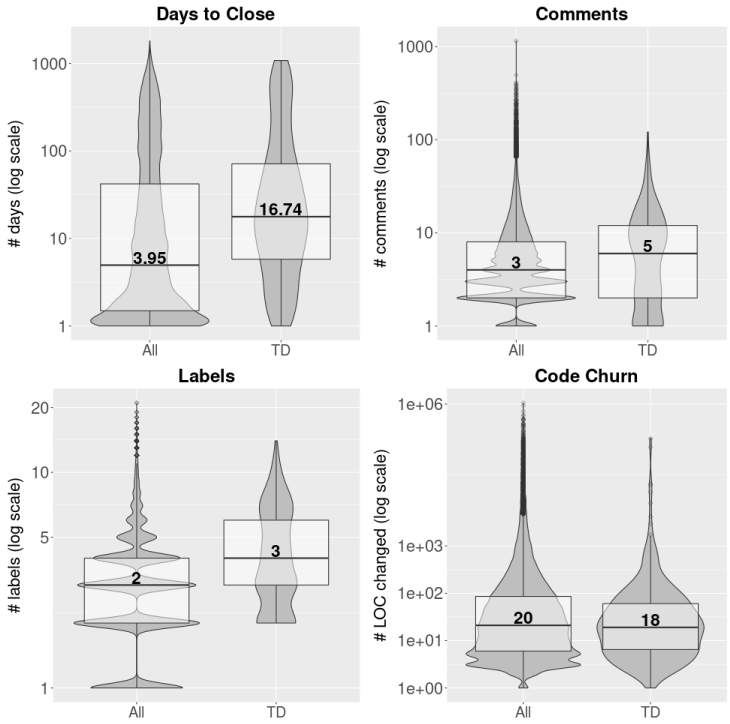

Fig. 2 Distribution of days, comments, labels, and code churn per issue.

applying the one-tailed variant of the Mann-Whitney U test (p-value ≤ 0.05). Finally, the last chart shows the code churn of SATD-I versus all issues in our dataset. This metric refers to the number of added and deleted lines of code in the commits responsible for closing the issues. The median code churn is 18 added/deleted lines (paid TD issues) versus 20 added/deleted lines (for all issues). However, in this case, the distributions are not statistically different (p-value = 0.13). *i.e.,* SATD issues are not different from other issues in terms of added and deleted lines of code.

2.2 Classification Study

In this section, we present our first exploratory study. Exploratory research is commonly adopted to investigate a problem that is not clearly defined or studied in the literature. It is conducted to provide a better understanding of the phenomenon and support further conclusions [51]. In this context, few previous research assessed the usage of issues to admit TD [7,4]. Particularly, none of them used labels as a proxy for TD identification. Therefore, in this study we aim to explore the types of TD commonly documented, discussed and fixed using labeled issues. Particularly, we seek to answer our first research question:

RQ1. What types of technical debt are paid in SATD-I*?* We aim to reveal the main types of TD documented in SATD-I. Similar classifications have been previously performed in the literature for SATD-C [34]. In this RQ, we seek to expand this knowledge by considering SATD types beyond the code.

We dedicate Section 2.2.1 to present the methodology applied to classify the SATD-I instances in our dataset. In Section 2.2.2, we present the results as follows: first, we discuss issues related to DESIGN (the most popular type of TD) and its corresponding sub-classification; next, we present the results for the other types of SATD-I. Section 2.2.3 discusses threats to validity.

2.2.1 Study Design

To identify the types of technical debt paid by developers, we carefully analyzed 286 SATD-I instances using *closed-card sort* [44], a technique to classify a set of documents into predetermined categories. This technique involves the following steps: (i) defining the set of categories, (ii) initial reading of the issues, (iii) classifying the issues by independent authors, (iv) resolving conflicts. We perform closed card sorting using categories previously elicited in the literature. Specifically, we reused the categories described in a study performed by Li *et al.* [27]. In this work, the authors describe a systematic mapping to identify and analyze scientific papers on TD from 1992 to 2013.[5] We classify the issues in our dataset according to the following categories proposed by Li *et al.*:

- DESIGN: refers to technical shortcuts used in internal method design and high-level architecture.
- UI: refers to debt on the elements of user interfaces.
- TESTS: refers to the absence of tests or to workarounds on existing code for testing.
- PERFORMANCE: refers to debt that affects system performance (*e.g.,* in time and memory usage).
- INFRASTRUCTURE: refers to debt on third-party tools, obsolete technologies or deprecated APIs.
- DOCUMENTATION: refers to insufficient, incomplete, or outdated documentation.
- CODE STYLE: refers to code style violations.
- BUILD: refers to debt in the build system, as when using scripts that make the build more complex or slow.
- SECURITY: refers to shortcuts that expose system data or compromises user permission access.
- REQUIREMENTS: refers to debt on requirements specification that leads to implementation problems.

[5] Although this taxonomy is based on research published before the SATD term was coined, we decided to use it because it covers problems that can occur beyond the source code (*e.g.,* UI and BUILD problems).

Since Design was the most popular case of Satd-I (as we found in our first round of classification), we decided to perform a sub-classification of this type of issue. Thus, we defined four categories:

- Complex Code: refers to intra-method poorly implemented code or to naming issues.
- Architecture: refers to high-level design problems, including inadequate organization of packages.
- Clean Up: refers to the elimination of obsolete or dead code.
- Code Duplication: refers to code clones that should be removed to improve maintainability.

After defining the mentioned categories, three authors of this paper manually analyzed the issues, by reading their descriptions and existing discussions, with the goal of assigning one (or more) categories. Each issue was analyzed by two independent authors. In 178 cases (62.2%) they agreed in the first proposed classification. For the Design subclassification (169 issues), the authors agreed in 92 cases (54.4%). The key challenge in this classification was to understand the purpose of the issue, based on the different pieces of information that it includes (*i.e.,* title, body, comments, labels and closing pull/merge-requests). Therefore, there were cases in which the authors prioritized different aspects to classify the issue, resulting in conflicting results. These cases were solved during the final step of the *closed-card sort* technique, where the authors discussed each conflict and reached a consensual classification considering all aspects of the issue. To facilitate the identification and discussion of the issues in this section, we label them using the initials of the repository name (*i.e.,* VS refers to microsoft/vscode; GL to gitlab/gitlab-ce; IF to influxdata/influxdb; SL to mirumee/saleor; and NX to nextcloud/server). The initials are then followed by an integer ID (*e.g.,* GL43 refers to issue 43 from GitLab).

2.2.2 RQ1. What types of technical debt are paid in Satd-I*?*

Satd-I *related to Design*

With 169 occurrences (59.1%), most of the selected issues refer to Design debt. In this case, we classified Design Satd-I into four subcategories, as presented in Table 2. As we can see, Complex Code is the type of Design TD more commonly paid by developers (43.8%), followed by Architecture (33.7%), Clean Up (18.9%), and Code Duplication (3.5%). Next, we describe and provide examples for each Design category.

Complex Code. In 74 cases (43.8%), Design issues are related to technical shortcuts that developers take when implementing methods. In this case, the payment involves changes only in the single method where the debt is located. As an example, the following issues are related to this type of Design Satd-I:

Table 2 Design SATD-I Classification.

Technical Debt	Occ.	%
COMPLEX CODE	74	43.8%
ARCHITECTURE	57	33.7%
CLEAN UP	32	18.9%
CODE DUPLICATION	6	3.5%

We should unify naming related to checkout functionality, as currently, we're mixing "checkout" with "cart", which leads to confusion when reading the code. I recommend that we settle on the name "checkout" and rename the Cart model and all other occurrences of cart. (SL1)

Currently, errors is an optional list of optional errors. While returning an empty list is probably not needed, current type forces the client to make sure the errors themselves are not null. (SL10)

Architecture. With 57 occurrences (33.7%), the second type of DESIGN TD most commonly paid by developers is related to high-level design flaws. To pay this type of debt, it is usually necessary to make changes in the organization of packages and modules, for example. The following issues are related to this type of SATD-I:

This class is way too big for its own good. For example, there's no need for it to update a project's main language in the same job/thread/process as the other work. (GL43)

The root of the TimeMachine tree contains a TimeSeries component. This component handles fetching time series data used in the TimeMachine (...) The aim of this refactor would be to move all state from the TimeSeries component into Redux and all logic into a thunk. (IF12)

Clean Up. Next, issues related to the presence of obsolete or dead code represent the third most common type of SATD-I, with 32 instances (18.9%). As an example, the following issue is related to this type of DESIGN TD:

In Milestone 11.4, we introduced personal_access_tokens. token_digest, so we can now remove personal_access _tokens.token. (GL47)

Code Duplication. Finally, with 6 occurrences (3.5%), the least common type of DESIGN SATD-I refers to duplicated code, as illustrated by the following issue:

There has been a lot of duplication of frontend code between Protected Branches and Protected Tag feature, this issue is intended to reduce duplication. (GL81)

Other Types of SATD-I

Table 3 presents the classification of the remaining SATD issues. As we can see, if we do not count issues related to DESIGN (59.1%), the most common type of paid TD refers to UI (10.1%), TESTS (8.7%), and PERFORMANCE (8%). Next, we describe these categories.

Table 3 Other Types of SATD-I.

Technical Debt	Occ.	%
UI	29	10.1%
TESTS	25	8.7%
PERFORMANCE	23	8%
INFRASTRUCTURE	18	6.3%
DOCUMENTATION	12	4.2%
CODE STYLE	8	2.8%
BUILD	4	1.4%
SECURITY	3	1.1%
REQUIREMENTS	3	1.1%

UI. With 29 occurrences (10.1%), the second most common type of SATD-related issues refers to debt on user interface code. In this case, developers implemented shortcuts that result in usability flaws, as mentioned in the following issue:

Today there is a "Building..." label appearing around the problems entry when building a project. I think this originates from a time where we did not have support to show progress in the status bar. (VS29)

Tests. In 25 cases (8.7%), SATD-related issues report the absence of tests or request improvements on existing tests. The following issue illustrates this type of TD:

I want to have some tests that will give me a better perspective for usage of DB queries under GraphQL API. I would like to have explicit logic to validate that it works as expected. (SL3)

Performance. With 23 occurrences (8%), the fourth most common type of SATD-I is related to performance concerns, in terms of time or memory usage. This is illustrated as follows:

underscore.js is bundled in vendor/core.js but it's the unminified version. Can we replace it with the minified version? The file size is a lot smaller. (NX1)

Every widget and actions in each extension has a global listener to check if there is a change and update itself. This causes 100s of listeners being added to a global event. (VS65)

> SATD-I is paid mostly to fix DESIGN flaws (∼60%). But we also found paid TD related to UI (10%), TESTS (9%), and PERFORMANCE (8%), for example.

Multiple-Category Types of SATD-I

After concluding this classification study, the third author of the paper reanalyzed all TD issues classified as UI, PERFORMANCE, BUILD, and INFRASTRUCTURE. The goal was to check whether these issues also include a discussion related to design or architecture concerns. The results are summarized in Table 4. As can be seen, multiple-category discussions are not common. In fact, they occurred in only two categories.

Table 4 Multiple-Category Discussion in SATD-I.

Technical Debt	# Instances	# Multiple-Category
UI	29	2
PERFORMANCE	23	2
INFRASTRUCTURE	18	0
BUILD	4	0

2.2.3 Threats to Validity

First, we acknowledge that this study is restricted to 286 closed issues classified as technical debt according to SATD-related labels. Although the issues were selected from relevant repositories, maintained by organizations like Microsoft and GitLab, we cannot generalize our findings to other systems, especially to the ones that apply different approaches to manage technical debt (*i.e.,* do not use TD-related labels). Second, we selected the issues by using TD-related labels as a proxy for technical debt identification. However, as discussed by Kruchten *et al.* [24], the concept of technical debt has been diluted since its original proposition. Thus, the misunderstanding of this concept by those who added the TD-labeled issues would affect the results of our study. To alleviate this threat, the first author of this paper carefully analyzed the initial dataset of 406 TD-labeled issues, and discarded 120 issues (29.6%) that did not have a clear indication of TD payment. Third, we should mention the subjective nature of the closed-card sort method. Despite the rigor followed by the authors to perform the classification, the replication of this activity may lead to different results. To mitigate this threat, special attention was paid during the discussions to resolve conflicts and to assign the final themes. Finally, we acknowledge a possible threat concerning the selection of the TD-related issues on GitHub-based projects.

2.3 Survey with Developers

In the initial part of our study, we also conducted a survey to reveal developers motivations for SATD-I insertion and payment. Particularly, we surveyed developers responsible for closing the 286 SATD-I instances collected in our first dataset. We relied on their responses to answer the following research questions:

RQ2. Why do developers introduce SATD-I*?* In this RQ, we reuse Martin Fowler's quadrant [14] to understand the origin of the studied SATD-I instances (*i.e.,* TD latter admitted vs TD introduced to ship earlier).

RQ3. Why do developers pay SATD-I*?* Next, we move a step further to investigate the motivations that drive developers to pay SATD-I. Besides, we also elicit the main problems caused by these debts.

We first present the methodology followed in this survey (Section 2.3.1). After that, we present the results for each research question (Sections 2.3.2 and 2.3.3). Section 2.3.4 highlights threats to validity.

2.3.1 Survey Design

To conduct this survey, we sent emails to developers that closed the SATD issues studied in this paper. Specifically, we selected from our initial dataset developers with public email address who were responsible for (i) closing a specific issue; or (ii) accepting a pull/merge-request that closes the issue (in GitLab, merge-requests are equivalent to pull-requests). From the total of 286 issues, we retrieved a list of 85 distinct emails. In the cases where the same developer was responsible for more than one issue, we selected the most recently closed one.

For each developer, we sent the questionnaire in an interval of at most six months after the date when the issues were closed. Figure 3 shows the template of the survey email. First, we presented the issue that represents the debt paid by the developer. Next, we proposed three questions with the goal of (1) investigating the reasons why developers pay technical debt; (2) unveiling maintenance problems caused by TD; and (3) understanding the intentions behind TD insertion. Questions (1) and (2) were open-ended, while question (3) provided two predefined options, reused from the technical debt quadrant proposed by Martin Fowler [14]. According to the author, technical debt can be classified into a quadrant divided into reckless/prudent and deliberate/inadvertent debt. We decided to give options only related to the deliberate/inadvertent axis. The rationale is that the reckless/prudent classification might result in biased answers because it requires the developer to make a self-judgment of his/her own work (*i.e.,* he/she needs to classify his/her own work as reckless or prudent). Although developers could simply select one of the answers, we allowed them to provide their own answers or to include comments to predefined answers.

I figured out that you closed the following issue from [repository name]:

[issue title] [issue link]

which is labeled as [TD-related tag].

I kindly ask you to answer the following questions:

1. Why did you decide to pay this TD?

2. Could you describe the maintenance problems caused by this TD?

3. Could you classify this TD under the following categories:

a. It was deliberately introduced to ship earlier
b. When it was introduced, we were not aware about the best design
c. Other answers (please clarify)

Fig. 3 Email sent to developers who paid SATD-I.

We received 30 answers coming from developers of four repositories (*i.e.,* response rate of 35.3%). Table 5 details the number of emails sent and the answers received per repository. MICROSOFT/ VSCODE and INFLUXDATA/INFLUXDB have the highest response rate (both with 40%). However, they do not represent the majority of the answers, once we received 23 answers from GitLab developers.[6]

Table 5 Survey answers.

Repository	Sent	Answers	%
MICROSOFT/VSCODE	10	4	40%
INFLUXDATA/INFLUXDB	5	2	40%
GITLAB/GITLAB-CE	66	23	34.9%
NEXTCLOUD/SERVER	3	1	33.3%
MIRUMEE/SALEOR	1	0	0%
Total	85	30	35.3%

To interpret the survey answers (1) and (2), the first author followed an *open card-sorting* [44]. This technique is used to identify themes (*i.e.,* patterns) in textual documents through the following steps: (i) identifying themes from the answers, (ii) reviewing the themes to find opportunities for merging, and (iii) defining and naming the final themes. During the analysis, one answer was discarded because the developer did not actually discuss the issue. In a final step, the last author reviewed and confirmed the proposed themes. In the

[6] It is worth mentioning that 33 developers (out of the 85 contacted) were also responsible for opening the corresponding issues. Among the 30 developers who answered our emails, 10 were also the authors of the SATD-I.

following discussion, we label the quotes with D1 to D29 to indicate developers answers.

2.3.2 RQ2. Why do developers introduce SATD-I*?*

To answer this question, we provided two predefined options: the first is related to developers decision of introducing TD as a choice for agility. The second corresponds to the scenario where developers only perceived the debt after it was introduced. We also left the opportunity for developers to clarify their answers and provide further information. Figure 4 presents the obtained results. As we can observe, most of the studied debts were intentionally introduced by developers *to ship earlier* (12 answers). In nine cases, developers were not aware of the TD when it was introduced (*i.e.,* the debt was initially unintentional and then *later admitted*). Finally, six developers provided other motivations. Next, we detail each of these reasons and provide quotes from extra comments discussed by developers.

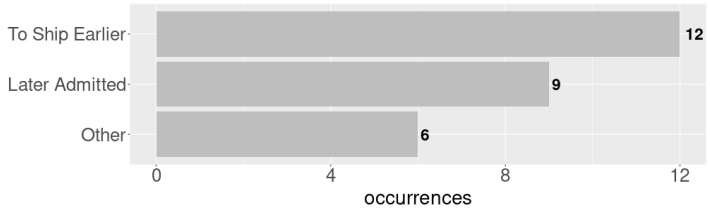

Fig. 4 Reasons for introducing SATD-I.

SATD introduced to ship earlier. In 12 answers (44.5%), developers confirmed that the technical debt was introduced to speed up development. In other words, to deliver faster, developers consciously added shortcuts in their code which were expected to be fixed in the future. To remind about this fact, they also decided to document the TD using an issue. D7, D16, and D9 provided further details for this reason:

It was thoroughly discussed and weighed up before we take the decision to accept the TD *to be dealt with on a next release. The* TD *wasn't introducing any critical performance issues or bugs to the system. Furthermore, we were confident that we could fix the* TD *in the next release, which happened.* (D7)

I think that usually when we introduce a technical debt it either helps us to ship something earlier/faster or makes first iteration of implementation much easier in general. (D16)

We were aware and were ok with the implementation for now as long as we fixed it afterwards (D9)

Later admitted TD. For nine developers (33.3%), the debt was originated by their lack of understanding about the best design solution at the time the code

was initially implemented. After discovering or facing the TD, they decided to admit it opening an issue. The following answers illustrate this scenario:

We figured we'd never hit "that" usecase. But we did. (D23)

The class just grew over time without planning. (D15)

Other reasons. Finally, six developers provided other reasons for introducing TD in their code (22.2%). Answers include the advent of new technologies that turned the old code a debt, and also the mischoice of design alternatives. This is illustrated in the following examples:

It slowly became TD *while at the time of the initial development it was most likely fine to code that way.* (D29)

I think the original author just overlooked that exposing these methods wasn't really needed. (D13)

> In most of the cases, SATD-I is introduced as a deliberate choice for agility (44.5%).

2.3.3 RQ3. Why do developers pay SATD-I*?*

In order to investigate the reasons why developers pay SATD-I, we combine answers from questions (1) and (2) of our e-mails (Section 2.3.1). First, we directly asked developers the reasons that drive such payment. Next, we complement our findings by eliciting a list of associated maintenance problems.

We first identified five distinct reasons why developers pay SATD-I, as reported in Table 6. As we can see, reducing TD interest is the most common motivation for SATD-I payment (65.5%), followed by the desire to have a clean code (27.6%). In some cases, a given answer produced more than one motivation. This explains why the number of occurrences is higher than the number of answers (29 answers). Next, we discuss these reasons.

Table 6 Reasons why developers pay SATD-I.

Why did you decide to pay this TD?	**Occ.**	**%**
To reduce TD *interest*	19	65.5%
To clean code	8	27.6%
To get familiarised with the codebase	2	6.9%
To collocate with other related work	2	6.9%
To increase test coverage	1	3.5%

To reduce TD interest. With 19 answers (65.5%), the most common reason for paying SATD-I is to reduce TD interest. Although developers did not directly mention the term *interest*, eliminating the maintenance burden caused

by the studied issues was mentioned in several answers. For example, D1 and D19 mention this motivation:

This was adding extra maintenance for me. (D1)

The component was growing too big, making it difficult to maintain. (D19)

To clean code. In eight cases (27.6%), technical debt payment is related to the desire of having a clean codebase (*e.g.,* to reduce code complexity and remove duplication). For example, the following answers are related to this motivation:

To keep the code clean and easy to read/maintain. (D27)

To get the benefits of a cleaner code (. . .). After fixing the TD*, understanding the code got easier. It also got smaller.* (D7)

> Technical debt is periodically paid to reduce its interests (66%), and to clean code (28%).

We also asked the survey participants to comment on the specific maintenance problems that motivated them to close the studied SATD-I instances. Table 7 presents the list of the most common answers. In this case, three answers (out of the 29 analyzed) were discarded because they were not clear. According to the remaining answers, TD is mostly responsible for slowing down code evolution, increasing maintenance effort due to duplicated code, and making it harder to read and understand code. The three problems occur with the same frequency (six answers for each).

Table 7 Maintenance problems caused by technical debt.

What problems are caused by this TD?	**Occ.**	**%**
Code was difficult to evolve	6	20.7%
Duplicated code was demanding extra effort	6	20.7%
Code was difficult to read and understand	6	20.7%
Code performance was poor	5	17.2%
Code was error-prone	4	13.8%
UI presented visual defects	1	3.5%

> Technical debt is commonly responsible for slowing down code evolution, duplicating maintenance effort, and making it harder to read and understand code.

2.3.4 Threats to Validity

In addition to general threats reported in Section 2.2.3 that also applies to this study—regarding generalization, dataset selection and subjectivity concerns—

we acknowledge that the presented results are based on the opinion of 29 developers, mostly from GitLab. Despite that, we claim that the obtained response rate (35.3%) represents a relevant mark in typical software engineering studies. Moreover, against our belief, the correctness of developers answers is also a threat to be reported. To alleviate it, we restricted our study to issues closed in the last six months, which was important to guarantee a higher response rate and to increase answers reliability.

3 Extended Study

In this paper, we also investigate tools to support developers in documenting and discussing SATD in issues. This tool support was suggested by many commenters after our initial study was indexed by Hacker News, in 2020.[7] To accomplish that, we first extend our initial dataset with additional SATD-I issues and a set of SATD-C comments (Section 3.1). Next, we conduct a study to investigate developers interest in creating SATD-I issues from SATD-C comments (Section 3.2). Finally, we evaluate the frequency of SATD-I references in SATD-C comments (Section 3.3).

3.1 Extended Dataset

To perform this extended study, we first build a new dataset, including both SATD-C and SATD-I instances. We decided to replace our initial dataset for three reasons:

- To include more systems (5 systems vs 190 systems in this new dataset);
- To include SATD-C instances (the initial dataset covered only SATD-I instances);
- To also consider opened issues, since they might also benefit from the kind of tool investigated in this section.

As result, this new dataset includes 20,265 SATD-I instances and 72,669 SATD-C instances from 190 GitHub projects. We explain the mining steps that we followed to select the new repositories and their SATD instances in Section 3.1.1. Next, we provide an overview of the new dataset in Section 3.1.2.

3.1.1 Mining Steps

To build this new dataset, we performed the following steps:

1. Project selection. We first focused on retrieving repositories in which SATD-I is a common practice (since it is the least explored form of SATD). As in Section 2.1, we searched for TD-related labels among the top-5,000 GitHub repositories, ordered by number of stars [42,5]. In contrast to our initial

[7] `https://news.ycombinator.com/item?id=22915584`

dataset—where we selected repositories containing three specific labels—we now adopted different steps to increase the number of retrieved repositories. We started this procedure by identifying the labels related to TD among all the 97,106 labels adopted in the 5K most-popular GitHub repositories. From this initial amount, we removed labels associated with less than 10 issues. The rationale is to discard labels that are not frequently used. By applying this filter, we discarded 60,032 labels, such as: *today*, *announcement*, and *Partner*. After that, we adopted multiple regular expressions to remove commonly used labels that *do not* denote TD [6]. For example, we removed labels like *bug* (2,635 labels), *enhancement* (1,828 labels), *feature* (1,606 labels), and *question* (1,455). By applying these heuristics, we discarded 19,209 labels. Finally, the first author of this paper manually read the remaining 17,865 labels $(97,106 - 60,032 - 19,209)$, in order to select the ones that explicitly indicate technical debt. *e.g., tech debt*, *debt*, *cleanup*, *workaround*. The second author of this paper also inspected a set of 500 randomly selected labels to confirm the classification. As a result, we obtained 219 labels, related to 190 repositories.

2. Mining SATD-I *instances.* Based on the initial selection of 190 repositories, we used GitHub's GraphQL API to search for issues tagged with the defined SATD-related labels. *i.e.,* we used the set of 219 labels collected in Step 1 as an indication of TD concerns documented in the studied issues. We also considered both open and closed issues. The rationale is to investigate both existing and paid instances of TD. As a result, we selected 20,265 SATD-I instances (4,866 open issues and 15,399 closed issues).[8]

3. Identifying SATD-C *instances.* Finally, we collected SATD-C instances by cloning and parsing the source code of the 190 repositories selected in Step 1. We analyzed the latest version of each repository at the moment of cloning. For each file, we first extracted code comments by using a regex-based service provided by the Python API *comment_parser*.[9] Then, we filtered every comment containing at least one of the following terms: *TODO*, *workaround*, *fixme*, and *hack*. We selected these terms as they are the most popular ones when reporting TD instances in source code comments [19,37,3]. As a result, we identified 74,306 comments, distributed through 182 repositories, *i.e.,* in eight repositories, no comment including such terms were identified. To validate this selection, the first author inspected 3K comments (randomly selected). He confirmed all of them indeed refer to SATD-C. However, in order to have an additional opinion, the second author analyzed a subset of 500 comments, also randomly selected from the initial sample of 3K comments. He also confirmed all of them are SATD-C instances. Next, the third author inspected the same subset of 500 comments. During the analysis, he raised a discussion about SATD-C instances that do not include any description about the debt

[8] We acknowledge that some of the selected issues might not document TD concerns (for example, developers may have incorrectly labeled some issues). Despite that, this fact does not invalidate our key goal in this extended study, which is studying tool support for SATD documentation.

[9] `https://pypi.org/project/comment-parser/`

(*e.g.,* comments that only include a TODO without any further textual description). As a result, we decided to remove 1,592 comments that not include text besides the searched words. During this validation, we found a bug (or a limitation) in our regex tool when handling very large comments. Therefore, we decided to remove comments with more than 64K characters (*i.e.,* 45 comments, in total). We also inspected some of these removed comments and found they are indeed false positives (all of them refer to minified JavaScript files, for example). In the end, our extended dataset includes 72,669 SATD-C instances (*i.e.,* $74,306 - 1,592 - 45$).

3.1.2 Dataset Characterization

Our extended dataset includes 20,265 SATD-I and 72,669 SATD-C instances, collected from 190 well-known GitHub repositories (10,075 stars per repository, on the median). For example, the list of studied repositories includes: VUEJS/VUE (186K stars), MICROSOFT/VSCODE (114K stars), KUBERNETES/KUBERNETES (79K stars) and ANGULAR/ANGULAR (75K stars). Figure 5 describes the distribution of both SATD forms in such repositories. We can observe a concentration on code comments as means to document SATD (*i.e.,* 81 SATD-C vs 42 SATD-I instances per repository, median values). This observation is statistically confirmed by applying the Mann-Whitney U test (p-value ≤ 0.05).

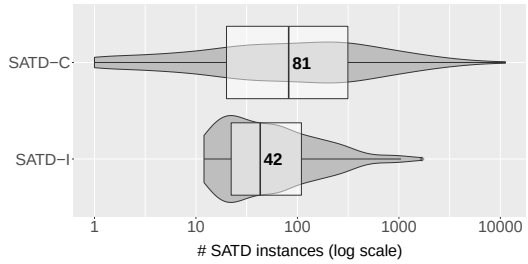

Fig. 5 Number of SATD-I and SATD-C instances per repository.

To better characterize our dataset and shed light on key quantitative measures about both datasets of SATD instances, we defined four dimensions[10]:

- *Size:* represents the volume of text used to describe TD.
- *Age:* characterizes how long the debt remains in code.
- *Activity:* represents the level of activity to document TD.
- *Engagement:* characterizes community involvement.

SATD-C *Characterization.* In Figure 6, we provide violin plots to describe the SATD-C instances in our dataset. To measure the first dimension (size), we

[10] However, we highlight that it is not possible to directly compare and contrast the two sets (SATD-C and SATD-I), since the metrics used in each one are distinct.

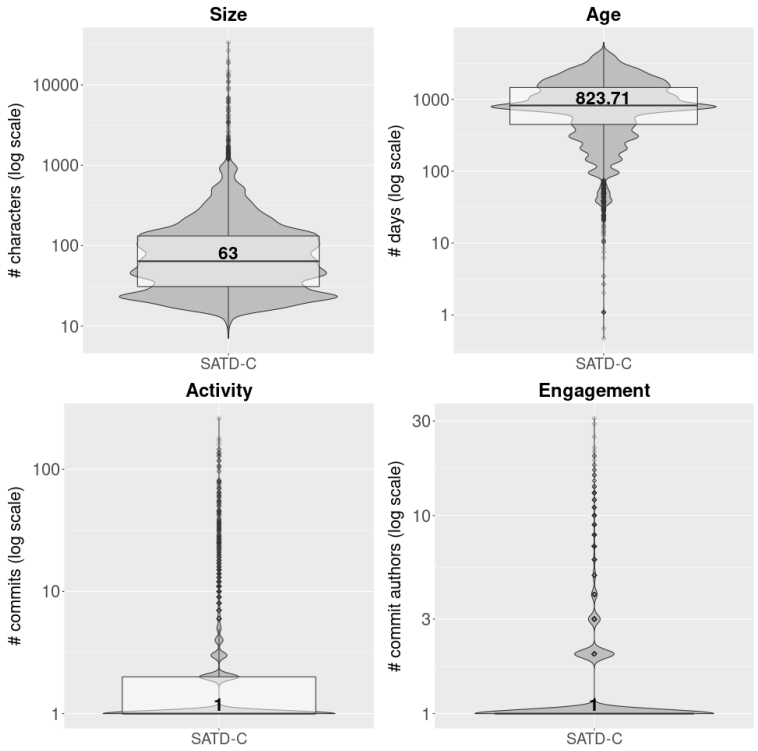

Fig. 6 Characterization of SATD-C instances in terms of size, age, activity and engagement.

counted the number of characters used in comment texts. In this case, the first quartile, median, and third quartile are 30, 63 and 131 characters. For age, we calculated the difference (in days, rounded with two decimal places) between the date of our data collection and the commit date when the SATD-C was introduced. The introduction date was obtained after traversing back the git tree until we find the commit responsible for adding each debt. As a result, the obtained quartiles are: 448.24, 823.71, 1,465.41 days, respectively. To characterize the level of activity performed by developers to document SATD-C, we counted the number of commits that modified the comment since its introduction. The first quartile, median, and third quartile are 1, 1 and 2 commits. Finally, to measure the community engagement, we counted the number of developers that modified the SATD-C comment. In this case, the three quartiles are equal to 1 author per SATD-C.

SATD-I *Characterization.* Figure 7 details the distribution of results for the metrics defined in each dimension, considering the SATD-I instances in our dataset. As for SATD-C, we used the number of characters as measure for the size dimension. The obtained quartiles are 203, 442 and 1,005 characters. To characterize the age of the studied SATD-I, we calculated the delta between the data collection and creation dates for the opened issues. In this case, the

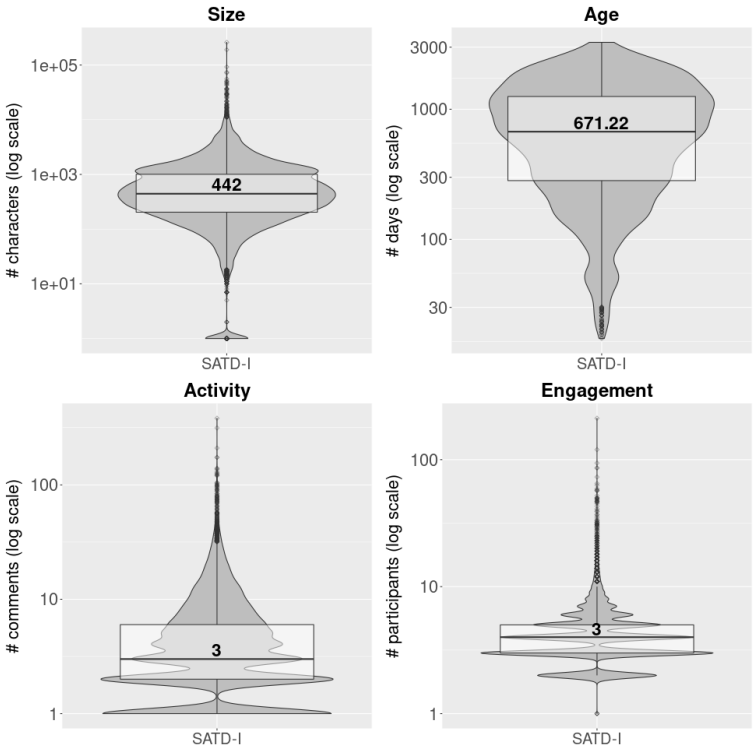

Fig. 7 Characterization of SATD-I instances in terms of size, age, activity and engagement.

first quartile, median, and third quartile are 281.77, 671.22 and 1,251.18 days. Next, we selected the number of comments as a metric to characterize the activity in SATD-I issues. We observed 2 comments in the first quartile, 3 in the median, and 6 in the third quartile. Finally, we used the number of participants in the issue to describe the community engagement. The results for this dimension are 2, 3 and 4 participants per issue.

3.2 Transforming SATD-C in SATD-I

In this section, we assess developers' interest in tools to automatically create GitHub issues from SATD-C. Such tools might be effective to support the migration of SATD-C into SATD-I. In this case, developers can benefit from issue trackers features to manage SATD, such as: discussions, assignments, and increased visibility. Particularly, we aim to answer the following research question:

RQ4. Are developers interested in tools to create issues from SATD-C*?* To answer the RQ, we first implement and evaluate a prototype tool, called ADMITD. This tool automatically identifies and reports SATD-C as GitHub issues. To evaluate its feasibility, we survey developers from 10 GitHub projects.

We dedicate Section 3.2.1 to present AdmiTD. Section 3.2.2 details our study design. In Section 3.2.3, we answer RQ4. Finally, Section 3.2.4 discusses threats to validity.

3.2.1 AdmiTD *Tool*

We first implemented AdmiTD, a prototype tool that automatically identifies and reports Satd-C as GitHub issues. To identify SATD comments, AdmiTD relies on the heuristics reported in Section 3.1.1 (*i.e.,* we extract source code comments and search by common TD-related terms reported in the literature). For each identified Satd-C, our prototype tool automatically creates an issue in the analyzed repository, containing three major parts: (1) title; (2) body; and (3) label. To create the title of the issue (flagged with #1 in Figure 8), AdmiTD relies on the first sentence of the comment. Next, to create the body of the issue (flag #2), the tool analyzes the git log of the Satd-C lines and retrieves its introducing commit, author, and date. The issue body describes this information, as well as the code snippet of the debt. We fixed a length of at most the next 10 lines of code. Finally, the issue is automatically labelled with the TD-related label commonly used in the repository (flag #3).

Fig. 8 Example of issue automatically created by AdmiTD, highlighting the generated (1) title, (2) body, and (3) label.

Particularly, the issue shown in Figure 8 was created by AdmiTD for getredash/redash. It was generated from the following Satd-C introduced in April 1st, 2015 (see flags #1 and #2 in the figure):

```
# TODO: this test can be refactored to use mock version of
should_schedule_next to simplify it.
```

In this case, the Satd-C author reports that a referred test should be refactored to use mocks. Besides, the *Tech Debt* label was included as adopted in the repository (flag #3).

Finally, to improve the quality of the generated issues and reduce duplicates, AdmiTD merges issues related to the same code comment (*i.e.,* Satd-C instances that would generate the same title). For that, issue bodies are appended with a separating line between each occurrence.

3.2.2 Survey Design

To explore developers interest in automatically transforming Satd-C in Satd-I, we conducted a survey with a sample of developers from 10 repositories selected in our dataset. For this, we first forked each repository, and applied AdmiTD in the fork (to avoid polluting the original repository with possibly unwanted issues). We also added two restrictions in AdmiTD to highlight relevant debts and facilitate developers' evaluation: (i) we only created issues for Satd-C instances retrieved from relevant files (*i.e.,* files that concentrate 80% of the source code changes in the repository); and (ii) we limited the number of Satd-I to 25 issues, randomly selected (*i.e.,* we restricted the report to the first page of GitHub issue tracker).

Table 8 shows the repositories selected to answer this second RQ, the number of existing Satd-C and Satd-I, as well as the number of issues generated by AdmiTD. As we can see, the tool created 154 issues for 1,314 Satd-C instances. The difference between the number of generated issues and Satd-C instances is due to the restrictions for relevant files, and due to the maximum size of GitHub issue pages, as well as to the merge approach implemented by AdmiTD. For example, in mkdocs/mkdocs two issues were merged and one Satd-C does not occur in a relevant file. All issues can be found at the forked repositories listed at AdmiTD GitHub page.[11]

Table 8 Satd-I instances automatically generated by AdmiTD.

Repository	# Satd-C	# Satd-I	# Gen.
osquery/osquery	45	22	25
balderdashy/sails	53	216	25
getredash/redash	834	40	25
falconry/falcon	51	39	25
naver/pinpoint	280	48	23
gabime/spdlog	13	19	11
encode/django-rest-framework	14	63	6
gionkunz/chartist-js	12	15	6
appium/appium	5	37	4
mkdocs/mkdocs	7	16	4
Total	**1,314**	**515**	**154**

After generating the aforementioned issues, we emailed the core developers of each repository, sharing the link of the issues in our forked repository, explaining our tool and goals, and asking a single question: *Is it worthwhile to create such issues in your repository?*

[11] `https://github.com/admitd`

3.2.3 RQ4. Are developers interested in tools to create issues from SATD-C*?*

From a total of 10 mails sent, we received four answers (response rate of 40%). However, none of the received answers included strong evidence of a positive perception of the automatically generated issues. In other words, although developers use GitHub issues to document and discuss their debts, they may not be interested in the automatic SATD-C→SATD-I transformation. For example, the core developer of MKDOCS/MKDOCS commented on the noise produced by SATD-I issues:

This is an interesting idea, but I don't think it is appropriate for mkdocs. Some projects may use a workflow where they would like to track each TODO as a bug. However, generally speaking I think that it will just create noise. For me a TODO is something that can be improved or fixed, but it isn't urgent. So if a developer spots it and has time they can take a look. Adding the noise of Github doesn't seem useful.

He also commented on the usage of IDE-based tools to search and index SATD-C instances:

Many IDEs or other developer tools can let you easily view these. I think that is enough.

A relevant example of such tools is the TODO TREE: an extension for VS Code with more than 1M installs.[12]

In addition, the core developer of OSQUERY/OSQUERY discussed the easiness of annotating TD in code comments to speed up development:

I think having TODOs/debt annotated in code is OK and we want to encourage folks to ship (tested) code quickly without having to be perfect or with all features implemented.[13]

> Developers did not provide strong evidence of being interested in tools to automatically create SATD-I from SATD-C.

3.2.4 Threats to Validity

First, our new dataset is based on a list of 219 TD-related labels, manually elicited from labels present in the top-5K most popular GitHub repositories. Although this list is based on the state-of-the-art [52], we acknowledge that it is not exhaustive. Moreover, we relied on a conservative set of terms to identify technical debt in code comments, such as: *TODO*, *workaround*, *fixme*, and *hack*. Therefore, it is possible that other SATD-C instances were not selected in our study, *i.e.,* comments with different keywords. Even though, we claim

[12] `https://marketplace.visualstudio.com/items?itemName=Gruntfuggly.todo-tree`

[13] Interestingly, this comment suggests that previous studies on SATD-C might have maximized the occurrence of TD, by considering any *TODO* to be problematic. As stated by the respondent, a *TODO* can just be a minor observation for the developer in the future.

that our final dataset is one of the largest in the literature, including more than 20K SATD-I and 72K SATD-C instances.

Another threat that would affect this study refers to the decision to rely on developers' self-admission to identify both SATD-I and SATD-C. We mitigate it by adopting a conservative list of TD-related keywords to mine SATD-C, and by manually analyzing the GitHub labels used to document SATD-I. Additionally, the first author manually inspected a relevant subset of the final dataset. We also mitigate the bias of developers' answers by restricting our survey to core developers.

We also acknowledge that it is possible that some SATD-I instances in our extended dataset do not refer to TD concerns. For example, developers' misunderstanding of the definition of the TD concept may result in incorrectly labeled issues. Our strategy to identify SATD-C comments may also represent a threat, since we adopted a simple and straightforward pattern-based approach. For both threats, we claim that we relied on conservative decisions to mitigate the occurrence of false-positives (*i.e.,* we selected a precise set of TD-related labels, as well as a meaningful set of TD keywords).

Additionally, we highlight that this study is limited to the 20,265 issues and 72,669 comments classified as SATD. Despite the size of the studied dataset and the relevance of their respective repositories (10K stars, on the median), our findings may not be generalized. Moreover, the results discussed in this study are based on the impressions of few developers. However, we claim that the received answers provided valuable insights.

3.3 Linking SATD-C to SATD-I

Given the negative results in Section 3.2, we decided to study another approach to support developers handling both SATD-C and SATD-I. Instead of creating new SATD-I from SATD-C, we investigate the viability of implementing tool support for connecting SATD-C to existing instances of SATD-I. We claim that such tools can make the navigation from SATD-C to SATD-I easier and straightforward. Therefore, these tools might help developers to include more details about a given debt, and to keep track of duplicated and outdated reports.

Our first step is to assess how often SATD-I issues are referenced from SATD-C comments (SATD-C→SATD-I). Thus, we seek to answer the following research question:

RQ5. Do developers refer to SATD-I *in* SATD-C*?* To answer this last RQ, we search for explicit links between SATD-C and SATD-I. For this, we mine the occurrences of SATD-I's URLs and IDs in SATD-C's comments. The existence of such links would mean that it is feasible to create such tool.

We dedicate Section 3.3.1 to describe the methods applied to find these links, as well as the obtained results. Section 3.3.2 presents threats to validity.

3.3.1 RQ5. Do developers refer to SATD-I *in* SATD-C*?*

Searching Cross-References

We implemented a custom procedure to triangulate the debts from both groups. Basically, we implemented a script to analyze the code comments of each SATD-C and to extract the numbers that matched one of the following conditions:

(a) Considered as a single token, such as *"TODO: Issue 949 - the following code . . . "*.
(b) Preceded by the `#` symbol, *e.g., "TODO #7967 help refactor"*.
(c) Preceded by the `/issue/` token, for example: *"TODO: change to 200* `https://github.com/loadimpact/k6/issues/1250`*"*.

Then, we cross-check these numbers with the issue codes of the SATD-I collected in Section 3.1 to link both SATD-C and SATD-I. SATD-C instances referring to issues outside of the extended dataset were discarded as they do not fulfill the SATD-I criteria adopted initially, *i.e.,* contain at least one TD-related label. Lastly, the first author manually inspected each link to ensure they indeed represent actual SATD instances (which indeed was confirmed in all cases).

From the 190 repositories initially analyzed, 23 of them contained at least one SATD-C→SATD-I occurrence (12.1%). Figure 9 presents the number of SATD-C and SATD-I collected in these repositories, as well as the number of SATD-C→SATD-I occurrences we found. Our triangulation process matched 80 references in total. From the perspective of SATD-C, this means that 0.36% of them refer to a SATD-I instance; conversely, 1.28% of SATD-I is referenced by at least one SATD-C instance in our dataset.

Fig. 9 Number of SATD-I and SATD-C instances considering only repositories with at least one SATD-C→SATD-I reference.

Table 9 lists the number and distribution of these references for each of the 23 repositories. COCKROACHDB/COCKROACH has the highest number of references (20), followed by RADAREORG/RADARE2 (11), and both ELASTIC/KIBANA and KUBERNETES/KUBERNETES, with 8 instances, each; together these four hold 58.8% of all SATD-C→SATD-I links. By contrast, 13 repositories have only one reference. Despite this high concentration, the occurrence

of Satd-C→Satd-I represents less than 5% in the top-4 from both Satd-I and Satd-C perspectives.

Table 9 Number of Satd-C→Satd-I references for each repository.

Repository	# Ref	% Satd-C	% Satd-I
cockroachdb/cockroach	20	0.61%	2.79%
radareorg/radare2	11	0.62%	4.06%
elastic/kibana	8	0.70%	2.69%
kubernetes/kubernetes	8	0.21%	0.92%
adobe/brackets	5	0.78%	3.29%
loadimpact/k6	5	0.71%	7.25%
dotnet/roslyn	3	0.23%	1.20%
kubernetes/minikube	3	6.38%	1.97%
angular/angular-cli	2	0.41%	1.27%
withspectrum/spectrum	2	0.93%	1.57%
microsoft/vscode	1	0.23%	0.06%
grpc/grpc	1	0.32%	0.36%
metabase/metabase	1	0.28%	0.45%
hashicorp/consul	1	0.08%	1.30%
influxdata/influxdb	1	0.21%	0.71%
firecracker-microvm/firecracker	1	5.26%	0.83%
microsoft/react-native-windows	1	0.26%	1.25%
elastic/beats	1	0.34%	5.56%
openshift/origin	1	0.03%	0.34%
jetstack/cert-manager	1	0.35%	2.38%
WordPress/gutenberg	1	0.74%	0.71%
tektoncd/pipeline	1	0.10%	2.08%
perkeep/perkeep	1	0.13%	8.33%
Total	**80**	**0.36%**	**1.28%**

After manually inspecting the content of such Satd-C→Satd-I occurrences, we noted that Satd-C were generally created to highlight the points in the code impacted by the associated Satd-I instance. For instance, the following Satd-C was identified at kubernetes/kubernetes:

```
#TODO refactor all tests to use real watch mechanism, see #72327
```

The referenced Satd-I, in turn, describes the need to *"Use fake client real watch mechanism in PV controller tests"*.

In any case, mentioning Satd-I in Satd-C is not a widespread practice in GitHub projects. Even after reducing our analysis to the few projects containing such a link, we observe that such referral is, in fact, barely used. Moreover, an in-depth analysis showed that 26 references (out of the 80 references initially detected) refer to duplicate links. In other words, 26 Satd-C comments refer to duplicate issues. This result suggests that Satd-I may be used to document TD that spans in multiple places in code.

> Overall, only 80 out of 22,327 (0.36%) Satd-C explicitly refer to Satd-I. Thus, cross-referring Satd-C and Satd-I is not a widespread practice.

Other Matching Approaches

In order to try to expand our dataset of SATD-C→SATD-I references, we analyzed more flexible matching techniques. The idea was to search for other traces that could relate both SATD. In this sense, we decided to try out two distinct approaches: *textual similarity*, where a pair of SATD-C→SATD-I is created in case they are textually similar to each other; and *timestamp proximity*, where the link is generated if both SATD are created next to each other given a time interval.

Textual Similarity. For each pair of SATD-C and SATD-I, we assume that the former is semantically related to the latter whenever there is a high textual similarity between them. For this, we relied on the *SequenceMatcher* class, available at the *difflib*[14] Python module; this class provides functions to compute the textual similarity level between two string sequences, returning a score in $[0, 1]$ range. We calculated the ratio for all possible SATD-C→SATD-I combinations, considering the texts retrieved from SATD-C comment blocks and SATD-I issue bodies. Then, we filtered the pairs with a ratio of 0.75 or higher. However, *we did not find any references based on these criteria.*

Timestamp Proximity. We considered that one SATD-C instance refers to a SATD-I if the latter was created at most 24 hours after the creation of the former; as a result, we leveraged a total of 879 pairs through this pattern. To verify the effectiveness of this approach, we selected a random sample of 87 occurrences and analyzed each one in order to validate this temporal connection.[15] However, *we were not able to identify any real association.* To illustrate this finding, consider the following SATD-C:

```
#Workaround for https://github.com/microsoft/vscode/issues/12865
check new scrollY and reset if necessary
```

As noted, this comment clearly mentioned issue #12865, but it was wrongly linked with issue #15515, as they were created in an interval of three hours.

Mixed Approach. We performed one last experiment by combining both approaches. This time, we considered a SATD-C→SATD-I as valid if they were created in a 24-hour window, but have a similarity of 0.30 or higher. As a result, this procedure returned 12 references. The first author manually checked each one, but *observed that they were all invalid.*

> More flexible approaches did not improve our search for SATD-C→SATD-I references. We were not able to find any new reference using such approaches.

[14] `https://docs.python.org/3/library/difflib.html#module-difflib`

[15] Determined after specifying a limit of 95% confidence level, with a margin of error of 10%.

3.3.2 Threats to Validity

In addition to the threats reported in Section 3.2.4—related to our new dataset and to generalization concerns—we first acknowledge that our strategies to match SATD-C and SATD-I are not exhaustive. In fact, developers can adopt different approaches to link them (*e.g.,* by using a tertiary software artifact). Moreover, to assess textual similarity we deal with the construct validity by adopting well-known functions in the Python community. Finally, we highlight that we set relaxed thresholds for the textual similarity ratio (0.75 and 0.30) in order to increase the chances of finding references. Even with such decisions, we could not improve our search.

4 Implications

Based on our results, we shed light on the following practical implications:

1. SATD-C *and* SATD-I *have different natures.* This difference explains the negative results we reported when investigating tool support for SATD-I (RQ4 and RQ5). We claim this result happened due to the distinct natures of SATD-C and SATD-I. Indeed, this claim is reinforced by the results we achieved in another recent paper, where we conducted a survey with 59 developers who documented TD in open source projects using both source code comments and issues [53]. As our key findings, we concluded that SATD-C is commonly adopted to report low-level TD (*i.e.,* debts associated with code snippets located next to the TD comment). For example, in the following SATD-C comment, a developer from ELASTIC/KIBANA indicates that a particular excerpt of the code is duplicated and should be cleaned up:

```
// TODO: everything below performs verification of manifest.yml
// files, and hence duplicates functionality already implemented
// in the package registry. At some point this should probably
// be replaced (or enhanced) with verification based on
// https://github.com/elastic/package-spec/
```

On the other hand, SATD-I is more suitable to document high-level concerns (*i.e.,* debts that tend to spread out over the code, such as the ones referring to design concerns). Figure 10 illustrates an example of a high-level TD concern documented through an issue. In this case, the developer from COCKROACHDB/COCKROACH created this SATD-I to report a debt that spans in multiple places of the code.

Moreover, we also found that the decision for adopting code comments or issues may also depend on other factors, such as the priority, impact or importance of TD problems. For example, developers usually document high priority debts in issues and tend to adopt code comments to provide context for future readers of the code.

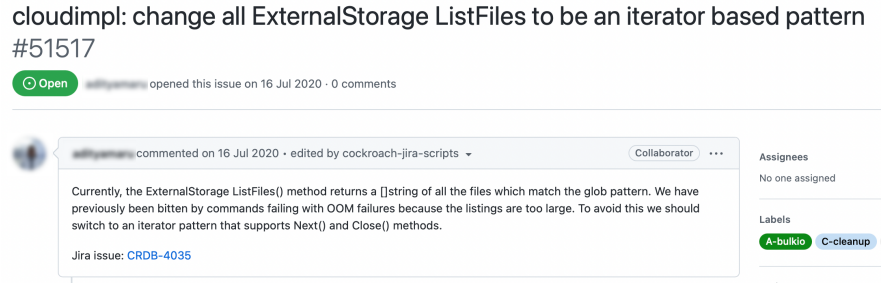

Fig. 10 Example of high-level SATD in COCKROACHDB/COCKROACH issues.

2. Issues might be more useful to report TD *related to crosscutting concerns.* In the line of the previous implication, we conjecture that issues might be more useful to document crosscutting TD. Particularly, in RQ1 we show that 40% of the studied SATD-I instances were related to TD more challenging to be documented in code comments (*e.g.,* UI, PERFORMANCE, and BUILD debts). In fact, this result complements previous SATD-C studies—that mainly identified SATD types related to source code—and prospects new horizons to SATD research, mainly related to less studied TD types.

3. Tools for linking SATD *types might not be worthy.* Still based on our negative results in RQ4 and RQ5, the implications to tool builders seem to be clear: they should not invest in tools to connect both types of SATD, using approaches similar to ADMITD. In terms of SATD-C tools, the most promising ones are related to indexing SATD-C instances, as mentioned by one of the surveyed developers. Similarly, we also envision similar tools for indexing SATD-I instances.

4. Researchers should include SATD-I *in* SATD *studies.* Although there is a relevant difference between the amount of TD documented in GitHub issues and in code comments (42 SATD-I and 81 SATD-C instances per repository, on the median), we consider that issues must be considered by SATD researches, as they tend to represent technical debts in a different shape. Therefore, it is recommended that studies on SATD rely not only on debts reported in comments (as usual), but also include instances documented in issue trackers.

5 Related Work

5.1 Studies on SATD-C

The concept of Self-Admitted Technical Debt (SATD) was first introduced by Potdar and Shihab [37]. In this work, the authors observe that developers commonly document TD through source code comments. Through the analysis of more than 100K code comments, they find that (i) 2.4%–31% of source code files contain self-admitted technical debt, (ii) experienced developers tend to

introduce more debts, and (iii) 26%–63% of SATD gets removed. Bavota and Russo [3] replicated this study on a larger dataset that includes 600K commits and 2 billion comments. The authors first confirm the previous findings, observing that the amount of SATD increases over time and tends to survive a long time in the system.

Maldonado *et al.* [33] investigate five Java open source projects with the purpose of examining the amount of TD removed, who performs the removal, how long it lives in a project, and what activities lead to the removal. As a result, the authors show that the majority of SATD is removed from projects by the same developer who introduced the debt (*i.e.,* self-removed), as part of bug fixing activities and the addition of new features. Zampetti *et al.* [57] perform a follow-up study, based on the dataset elicited by Maldonado *et al.* [33], to quanti- and qualitatively investigate how self-admitted technical debt is removed. Specifically, the authors assess the amount of SATD removals that are actually accidental transformations, as well as the extent to which SATD removals are documented in commit messages.

Sierra *et al.* [41] investigate the possibility of using source code comments that indicate technical debt to resolve architectural divergences. The authors used a dataset of previously classified SATD comments to trace architectural divergences in an open-source system. They found that 14% of divergences could be directly traced. Therefore, they stand that it is viable to use SATD comments as an indicator of architectural divergences. Zampetti *et al.* [56] investigated the adoption of SATD-C as a proxy to recommend developers to write new code. The authors also indicated when SATD should be documented (or "self-admitted"). As a result, the presented approach achieved good results, improving readability, size, and complexity metrics.

Regarding SATD-C identification, Farias *et al.* [11], Huang *et al.* [19], Liu *et al.* [30], and Guo *et al.* [18] also identify SATD by mining source code comments. Moreover, other studies propose the use of natural language processing (NLP) techniques to support SATD identification [34]. For example, Flisar and Podgorelec [12], Huang *et al.* [19], Ren *et al.* [38], Fahid *et al.* [10], and Wang *et al.* [48] use machine learning techniques for automating SATD detection. Dai and Kruchten *et al.* [8] improves these approaches by identifying non-code-level technical debt. Maldonado *et al.* [35] proposed a technique to precisely identify SATD, outperforming the current state-of-the-art, based on fixed keywords and phrases. Farias *et al.* [15] evaluate a set of contextualized patterns to detect SATD-C using code comment analysis. As a result, the authors show that the adoption of pattern-based analysis can contribute to improve existing methods for automatically identifying and classifying SATD items.

Recent research moved in the direction of improving SATD-C identification, removal, and management. For example, Zampetti *et al.* [58] showed that SATD removal follows recurrent patterns. They indicated that it is feasible to automatically recommend strategies to pay SATD concerns related to changing API calls, conditionals, method signatures, exception handling, and return statements. Iammarino *et al.* [21,20] investigated the relationship between refactoring and SATD-C removal. Although, the authors show that refactorings tend to

co-occur with SATD-C removals, they indicate that such improvements belong to different activities performed at the same time. Fucci *et al.* [17] investigated the extent which the term "self-admitted" can be used in the context of TD documented in source code. The results suggest that SATD-C may possibly be used as a sub-optimal strategy to perform code review. Kashiwa *et al.* [23] investigates the nature of SATD comments introduced during modern code reviews. The authors show that 28%–48% of SATD-C are introduced during code reviews, as such comments are used as means of communication between reviewers and authors.

To explore SATD-C in-depth, several studies focused on particular aspects of this practice. For example, Tan *et al.* [46] conducted an online survey to investigate whether practitioners repay their own debt intentionally. As results, the authors highlight the relevance of the sense of self-responsibility in driving developers to repay SATD-C. Maipradit *et al.* [31,32] investigate a particular class of SATD-C, named as "On-Hold" SATD. In this case, the authors provide an automated classifier that can identify debts which contains a condition to indicate that a developer is waiting for a certain event or an updated functionality. Fucci *et al.* [16] explore developers' habits in SATD annotation. They mainly show that SATD-C related to functional problems or on-hold conditions tend to be more negative. Besides, few SATD comments include external references. Other research include the investigation of SATD-C in particular environments. For example, Azuma *et al.* [2] investigates SATD in Dockerfiles. Xiao *et al.* [54] characterizes SATD-C in build systems. Lage *et al.* [13] studies usability TD. Melina [47] explores SATD in R packages.

5.2 Studies on SATD-I

Early researches indirectly tackled the adoption of issues as means to document technical debt. For example, Martini *et al.* [36] conducted several qualitative studies to investigate the cost of managing TD, the tools used to track it, and how a tracking process is introduced in practice. As a result, they show that only 7.2% of the participants methodically track technical debt. In this context, the majority of them adopt issues and backlog tools for this purpose. Yli-Huumo *et al.* [55] investigated the practices adopted by eight development teams to manage technical debt. In 6 cases, they observed that the teams informally adopted the strategy of creating JIRA issues to document TD concerns. As a result, the authors propose a framework to manage TD, including the recommendation to create issues to document their occurrence. Silva *et al.* [43] also investigate the occurrence of TD discussions beyond the code. Particularly, the authors investigate the different types of Technical Debt that can lead to the rejection of pull requests. As a result, the authors highlight that design and test debts cause 63% of the rejections.

The first study to prospect the idea of deeply analysing SATD in issues (*i.e.,* SATD-I) was performed by Bellomo *et al.* [4]. In this work, the authors manually examined a sample of 1,264 issues, mined from four industry and

governmental issue trackers. They found that developers discussed TD in 109 examples, demonstrating that they were aware of its concept and risks. Dai and Kruchten [8] also studied the possibility of detecting TD in issue comments by applying natural language processing and machine learning techniques to identify TD in 8,149 issues. As a result, the authors provide 114 keywords that can be used to detect different types of TD from issue descriptions.

Recently, Li *et al.* [26] conducted a case study to investigate the existence of SATD in issue discussions. For this, they manually investigated a sample of 500 issues from two open source projects (Hadoop and Camel). As a result, the authors classified a set of 117 discussions about TD in categories previously proposed in the literature. They also used these discussions to explore identification strategies and payment activities. In a follow-up study, Li *et al.* [25] focused on identifying TD evidences in issue trackers. The authors collected a training dataset that includes 4.2K issues, and broke down in 23K issue sections. They used this dataset to propose an approach to automatically identify TD evidences in issue trackers. The resulting approach outperforms baseline techniques and indicates that SATD keywords are intuitive.

5.3 Comparison with Previous Studies

To the best of our knowledge, this is the first study that empirically investigates the interplay between two forms of documenting TD, *i.e.,* by using source code comments (SATD-C) and by using issues (SATD-I). As discussed in Section 5.1, previous studies focused on evaluating SATD documented in code comments, providing insights about its adoption, removal, management, and best practices. Thus, most of such studies do not expand their conclusions to TD documented in other means.

By contrast, other papers that study SATD-I (Section5.2) have focused only on this particular strategy to document SATD. Thus, our main contribution is the study of both forms of documenting TD, *i.e.,* we did not restrict the study to a single form but we relied on datasets with real instances of SATD-C and SATD-I to study their relationships. Moreover, we also report negative results on the feasibility of implementing two distinct tools to deal with SATD: (1) tools that automatically create issues to document TD reported in source code comments; (2) tools that create links between SATD-C and SATD-I instances. These results may help tool builders and researchers to assess alternative tool strategies.

6 Conclusion

In this paper, we performed three complementary studies with the purpose of answering five research questions on Self-Admitted Technical Debt documented through labelled issues (SATD-I). Initially, we conducted two exploratory studies that analyzed 286 SATD-I instances to (1) identify the types

of SATD-I more frequently paid, (2) understand the intentions behind SATD-I insertion, and (3) investigate the reasons why developers pay SATD-I.

Furthermore, we extend these studies by exploring the adoption of tools to report TD. For that, we first built a dataset of 20,265 SATD-I instances and 72,669 SATD-C instances, mined from 190 well-known GitHub projects (*e.g.,* VUEJS/VUE, MICROSOFT/VSCODE and KUBERNETES/KUBERNETES). We also implemented ADMITD, a prototype tool that automatically identifies and reports SATD-C in GitHub issues. We used this dataset and tool to (4) explore developers interest in automatically creating GitHub issues based on SATD-C; (5) investigate whether developers cross-reference SATD-I in SATD-C. As a result, we show that:

- In almost 60% of the cases, we found that SATD-I is related to DESIGN flaws (with a concentration on method-level debt in 44% of this total);
- About 45% of the studied debt was introduced to ship earlier;
- Most developers pay SATD-I to reduce its interest, and to have a clean code;
- Developers are not interested in tools to automatically create SATD-I from SATD-C. We attempted to use ADMITD in 10 GitHub repositories without success;
- It might not be feasible to create a tool to link SATD-C and SATD-I instances. When looking for such references in our dataset we find out that this is not a widespread practice, *i.e.,* it happens for less than 1% of the SATD-Cs.

Future work. As future work, we first envision an in-depth analysis of the code transformations performed to pay SATD-I debts. Based on this dataset of transformations, we may develop tools and techniques to guide developers on TD payment (*e.g.,* by recommending how to perform changes that contribute to the actual removal of the debt). Finally, we also prospect as future work conducting an in-depth investigation on whether code comments labeled with *TODOs* are indeed TD, and which type of debt they document. Particularly, this study may fill a gap in the literature, by shedding light on situations where *TODO* comments are just a remark for future work.

Replication Package. We provide the complete dataset used in this paper and a replication package at: `https://doi.org/10.5281/zenodo.6532378`.

Acknowledgements We thank the developers who participated in our surveys and shared their ideas and practices about technical debt documentation and payment. This research is supported by grants from FAPEMIG, CNPq, and CAPES.

References

1. Alves, N.S., Mendes, T.S., de Mendonça, M.G., Spínola, R.O., Shull, F., Seaman, C.: Identification and management of technical debt. Information and Software Technology **70**(C), 100–121 (2016)

2. Azuma, H., Matsumoto, S., Kamei, Y., Kusumoto, S.: An empirical study on self-admitted technical debt in dockerfiles. Empirical Software Engineering **27**, 1–26 (2022)
3. Bavota, G., Russo, B.: A large-scale empirical study on self-admitted technical debt. In: 13th Working Conference on Mining Software Repositories (MSR), pp. 315–326 (2016)
4. Bellomo, S., Nord, R.L., Ozkaya, I., Popeck, M.: Got technical debt? Surfacing elusive technical debt in issue trackers. In: 13th International Conference on Mining Software Repositories (MSR), pp. 327–338 (2016)
5. Borges, H., Hora, A., Valente, M.T.: Understanding the factors that impact the popularity of GitHub repositories. In: 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 334–344 (2016)
6. Cabot, J., Cánovas Izquierdo, J.L., Cosentino, V., Rolandi, B.: Exploring the use of labels to categorize issues in open-source software projects. In: 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 550–554 (2015)
7. Cunningham, W.: The WyCash portfolio management system. In: 7th Object-oriented Programming Systems, Languages, and Applications (OOPSLA), pp. 29–30 (1992)
8. Dai, K., Kruchten, P.: Detecting technical debt through issue trackers. In: 5th International Workshop on Quantitative Approaches to Software Quality (QuASoQ), pp. 59–65 (2017)
9. Ernst, N.A., Bellomo, S., Ozkaya, I., Nord, R.L., Gorton, I.: Measure it? Manage it? Ignore it? Software practitioners and technical debt. In: 10th Joint Meeting on Foundations of Software Engineering (FSE), pp. 50–60 (2015)
10. Fahid, F.M., Yu, Z., Menzies, T.: Better technical debt detection via SURVEYing. Computing Research Repository **abs/1905.08297** (2019)
11. Farias, M.A., Santos, J.A., Kalinowski, M., Mendonça, M., Spinola, R.O.: Investigating the identification of technical debt through code comment analysis. In: 18th International Conference on Enterprise Information Systems (ICEIS), pp. 284–309 (2016)
12. Flisar, J., Podgorelec, V.: Identification of self-admitted technical debt using enhanced feature selection based on word embedding. IEEE Access **7**(1), 106475–106494 (2019)
13. da Fonseca Lage, L.C., Kalinowski, M., Trevisan, D., Spinola, R.: Usability technical debt in software projects: A multi-case study. In: 13th International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 1–6 (2019)
14. Fowler, M.: Technicaldebtquadrant. `https://martinfowler.com/bliki/TechnicalDebtQuadrant.html`. [accessed 10-October-2019]
15. de Freitas Farias, M.A., de Mendonça Neto, M.G., Kalinowski, M., Spínola, R.O.: Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary. Information and Software Technology **121**, 106270–106270 (2020)
16. Fucci, G., Cassee, N.W., Zampetti, F., Novielli, N., Serebrenik, A., Penta, M.D.: Waiting around or job half-done? Sentiment in self-admitted technical debt. In: 18th International Conference on Mining Software Repositories (MSR), pp. 1–10 (2021)
17. Fucci, G., Zampetti, F., Serebrenik, A., Penta, M.D.: Who (self) admits technical debt? In: 36th International Conference on Software Maintenance and Evolution (ICSME), pp. 672–676 (2020)
18. Guo, Z., Liu, S., Liu, J., Li, Y., Chen, L., Lu, H., Zhou, Y.: How far have we progressed in identifying self-admitted technical debts? a comprehensive empirical study. ACM Transactions on Software Engineering Methodology **30**, 1–56 (2021)
19. Huang, Q., Shihab, E., Xia, X., Lo, D., Li, S.: Identifying self-admitted technical debt in open source projects using text mining. Empirical Software Engineering **23**(1), 418–451 (2018)
20. Iammarino, M., Zampetti, F., Aversano, L., Di Penta, M.: An empirical study on the co-occurrence between refactoring actions and self-admitted technical debt removal. Journal of Systems and Software **178**, 110976–110976 (2021)
21. Iammarino, M., Zampetti, F., Aversano, L., Penta, M.D.: Self-admitted technical debt removal and refactoring actions: Co-occurrence or more? In: 35th International Conference on Software Maintenance and Evolution (ICSME), pp. 186–190 (2019)
22. Kamei, Y., Maldonado, E.D.S., Shihab, E., Ubayashi, N.: Using analytics to quantify the interest of self-admitted technical debt. In: 1st International Workshop on Technical Debt Analytics (TDA), pp. 68–71 (2016)

23. Kashiwa, Y., Nishikawa, R., Kamei, Y., Kondo, M., Shihab, E., Sato, R., Ubayashi, N.: An empirical study on self-admitted technical debt in modern code review. Information and Software Technology **146**, 106855–106855 (2022)
24. Kruchten, P., Nord, R.L., Ozkaya, I.: Technical debt: from metaphor to theory and practice. IEEE Software **29**(6), 18–21 (2012)
25. Li, Y., Soliman, M., Avgeriou, P.: Identifying self-admitted technical debt in issue tracking systems using machine learning. Empirical Software Engineering **1**, 1–1 (2022)
26. Li, Y., Soliman, O., Avgeriou, P.: Identification and remediation of self-admitted technical debt in issue trackers. In: 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 495–503 (2020)
27. Li, Z., Avgeriou, P., Liang, P.: A systematic mapping study on technical debt and its management. Journal of Systems and Software **101**(C), 193–220 (2015)
28. Lim, E., Taksande, N., Seaman, C.: A balancing act: what software practitioners have to say about technical debt. IEEE Software **29**(6), 22–27 (2012)
29. de Lima, B.S., Garcia, R.E., Eler, D.M.: Toward prioritization of self-admitted technical debt: an approach to support decision to payment. Software Quality Journal **1**, 1–1 (2022)
30. Liu, Z., Huang, Q., Xia, X., Shihab, E., Lo, D., Li, S.: SATD Detector: a text-mining-based self-admitted technical debt detection tool. In: 40th International Conference on Software Engineering: Companion (ICSE-Companion), pp. 9–12 (2018)
31. Maipradit, R., Lin, B., Nagy, C., Bavota, G., Lanza, M., Hata, H., Matsumoto, K.: Automated identification of on-hold self-admitted technical debt. In: 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 54–64 (2020)
32. Maipradit, R., Treude, C., Hata, H., Matsumoto, K.: Wait for it: Identifying "on-hold" self-admitted technical debt. Empirical Software Engineering **25**, 3770–3798 (2020)
33. Maldonado, E.D.S., Abdalkareem, R., Shihab, E., Serebrenik, A.: An empirical study on the removal of self-admitted technical debt. In: 33rd International Conference on Software Maintenance and Evolution (ICSME), pp. 238–248 (2017)
34. Maldonado, E.D.S., Shihab, E.: Detecting and quantifying different types of self-admitted technical debt. In: 7th International Workshop on Managing Technical Debt (MTD), pp. 9–15 (2015)
35. Maldonado, E.D.S., Shihab, E., Tsantalis, N.: Using natural language processing to automatically detect self-admitted technical debt. IEEE Transactions on Software Engineering **43**(11), 1044–1062 (2017)
36. Martini, A., Besker, T., Bosch, J.: Technical debt tracking: Current state of practice: A survey and multiple case study in 15 large organizations. Science of Computer Programming **163**, 42–61 (2018)
37. Potdar, A., Shihab, E.: An exploratory study on self-admitted technical debt. In: 30th International Conference on Software Maintenance and Evolution (ICSME), pp. 91–100 (2014)
38. Ren, X., Xing, Z., Xia, X., Lo, D., Wang, X., Grundy, J.: Neural network-based detection of self-admitted technical debt: from performance to explainability. ACM Transactions on Software Engineering and Methodology **28**(3), 1–45 (2019)
39. Rios, N., de Mendonça Neto, M.G., Spínola, R.O.: A tertiary study on technical debt: types, management strategies, research trends, and base information for practitioners. Information and Software Technology **102**(1), 117–145 (2018)
40. Sierra, G., Shihab, E., Kamei, Y.: A survey of self-admitted technical debt. Journal of Systems and Software **152**(1), 70–82 (2019)
41. Sierra, G., Tahmid, A., Shihab, E., Tsantalis, N.: Is self-admitted technical debt a good indicator of architectural divergences? In: 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 534–543 (2019)
42. Silva, H., Valente, M.T.: What's in a GitHub star? Understanding repository starring practices in a social coding platform. Journal of Systems and Software **146**, 112–129 (2018)
43. Silva, M., Terra, R., Valente, M.T.: Does technical debt lead to the rejection of pull requests? In: 12nd Brazilian Symposium on Information Systems (SBSI), pp. 1–7 (2016)
44. Spencer, D.: Card Sorting: Designing Usable Categories. Rosenfeld Media (2009)

45. Storey, M.A., Ryall, J., Bull, R.I., Myers, D., Singer, J.: TODO or to Bug: exploring how task annotations play a role in the work practices of software developers. In: 30th International Conference on Software Engineering (ICSE), pp. 251–260 (2008)
46. Tan, J., Feitosa, D., Avgeriou, P.: Do practitioners intentionally self-fix technical debt and why? In: 37th International Conference on Software Maintenance and Evolution (ICSME), pp. 251–262 (2021)
47. Vidoni, M.: Self-admitted technical debt in r packages: An exploratory study. In: 18th International Conference on Mining Software Repositories (MSR), pp. 179–189 (2021)
48. Wang, X., Liu, J., Li, L., Chen, X., Liu, X., Wu, H.: Detecting and explaining self-admitted technical debts with attention-based neural networks. In: 35th International Conference on Automated Software Engineering (ASE), p. 871–882 (2020)
49. Wehaibi, S., Shihab, E., Guerrouj, L.: Examining the impact of self-admitted technical debt on software quality. In: 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 179–188 (2016)
50. Wiese, M., Rachow, P., Riebisch, M., Schwarze, J.: Preventing technical debt with the tap framework for technical debt aware management. Information and Software Technology **148**, 106926 (2022)
51. Wohlin, C., Runeson, P., Hst, M., Ohlsson, M.C., Regnell, B., Wessln, A.: Experimentation in Software Engineering. Springer (2012)
52. Xavier, L., Ferreira, F., Brito, R., Valente, M.T.: Beyond the code: Mining self-admitted technical debt in issue tracker systems. In: 17th International Conference on Mining Software Repositories (MSR), pp. 137–146 (2020)
53. Xavier, L., Montandon, J.E., Valente, M.T.: Comments or issues: Where to document technical debt? IEEE Software **1**, 1–14 (2022)
54. Xiao, T., Wang, D., Mcintosh, S., Hata, H., Kula, R.G., Ishio, T., Matsumoto, K.: Characterizing and mitigating self-admitted technical debt in build systems. IEEE Transactions on Software Engineering **1**, 1–1 (2021)
55. Yli-Huumo, J., Maglyas, A., Smolander, K.: How do software development teams manage technical debt? - an empirical study. Journal of Systems and Software **120**(C), 195–218 (2016)
56. Zampetti, F., Noiseux, C., Antoniol, G., Khomh, F., Penta, M.D.: Recommending when design technical debt should be self-admitted. In: 33rd International Conference on Software Maintenance and Evolution (ICSME), pp. 216–226 (2017)
57. Zampetti, F., Serebrenik, A., Penta, M.D.: Was self-admitted technical debt removal a real removal? An in-depth perspective. In: 15th International Conference on Mining Software Repositories (MSR), pp. 526–536 (2018)
58. Zampetti, F., Serebrenik, A., Penta, M.D.: Automatically learning patterns for self-admitted technical debt removal. In: 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 355–366 (2020)
59. Zazworka, N., Shaw, M.A., Shull, F., Seaman, C.: Investigating the impact of design debt on software quality. In: 2nd Workshop on Managing Technical Debt (MTD), pp. 17–23 (2011)
60. Zazworka, N., Spínola, R.O., Vetro', A., Shull, F., Seaman, C.: A case study on effectively identifying technical debt. In: 17th International Conference on Evaluation and Assessment in Software Engineering (EASE), pp. 42–47 (2013)