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

Self-Admitted Technical Debt and Comments' Polarity: An Empirical Study

Nathan Cassee · Fiorella Zampetti · Nicole Novielli · Alexander Serebrenik · Massimiliano Di Penta

March 14, 2022

Abstract Self-Admitted Technical Debt (SATD) consists of annotations —typically, but not only, source code comments— pointing out incomplete features, maintainability problems, or, in general, portions of a program not-ready yet. The way a SATD comment is written, and specifically its polarity, may be a proxy indicator of the severity of the problem and, to some extent, of the priority with which it should be addressed. In this paper, we study the relationship between different types of SATD comments in source code and their polarity, to understand in which circumstances (and why) developers use negative or rather neutral comments to highlight an SATD. To address this goal, we combine a manual analysis of 1038 SATD comments from a curated dataset with a survey involving 46 professional developers. First of all, we categorize SATD content into its types. Then, we study the extent to which developers express negative sentiment in different types of SATD as a proxy for priority, and whether they believe this can be considered as an acceptable practice. Finally, we look at whether such annotations contain additional details such as bug references and developers' names/initials. Results of the study indicate that SATD comments are mainly used for annotating poor implementation choices ($\simeq 41\%$) and partially

Nathan Cassee
Eindhoven University of Technology, The Netherlands
E-mail: n.w.cassee@tue.nl

Fiorella Zampetti
University of Sannio, Italy
E-mail: fiorella.zampetti@unisannio.it

Nicole Novielli
University of Bari, Italy
E-mail: nicole.novielli@uniba.it

Alexander Serebrenik
Eindhoven University of Technology, The Netherlands
E-mail: a.serebrenik@tue.nl

Massimiliano Di Penta
University of Sannio, Italy
E-mail: dipenta@unisannio.it

implemented functionality ($\simeq$ 22%). The latter may depend from "waiting" for other features being implemented, and this makes SATD comments more negatives than in other cases. Around 30% of the survey respondents agree on using/interpreting negative sentiment as a proxy for priority, while 50% of them indicate that it would be better to discuss SATD on issue trackers and not in the source code. However, while our study indicates that open-source developers use links to external systems, such as bug identifiers, to annotate high-priority SATD, better tool support is required for SATD management.

Keywords Self-Admitted Technical Debt; Sentiment Analysis; Empirical Study

1 Introduction

Self-Admitted Technical Debt (SATD) (Potdar and Shihab, 2014) refers to source code comments (as well as other annotations elsewhere) indicating that the corresponding source code is (temporarily) inadequate, e.g., because the implementation is incomplete, buggy, or smelly. The identification of SATD (da S. Maldonado et al., 2017; Ren et al., 2019), as well as its introduction or removal, have attracted significant attention of the research community (Bavota and Russo, 2016; da S. Maldonado et al., 2017; Zampetti et al., 2018; Rantala et al., 2020; Zampetti et al., 2020).

In this paper, we study the annotation practices of open-source developers from two perspectives. First, we use an existing curated dataset of SATD comments (da S. Maldonado et al., 2017) to study their content and their sentiment polarity. Second, we survey open-source developers to (i) ask them about specific annotation practices they adopt, and (ii) elicit the SATD comments that they would draft in five different scenarios representative of code not being right yet.

To understand which kinds of technical debt (TD) are annotated by developers, previous literature has also categorized SATD comments. The categorizations of SATD proposed so far are based on the various phases of the software development process (da S. Maldonado and Shihab, 2015; Bavota and Russo, 2016). As such, they (i) miss the opportunity to identify concerns transcending the boundaries of individual development phases such as waiting for other components to be ready, and (ii) are somewhat broad because the SATD content still lacks an in-depth classification. Specifically, while SATD might manifest at one phase of the software development process, resolving it might require activities typically associated with another phase. For instance, the following SATD comment, taken from the Apache Ant project, can manifest during testing but its resolution requires a bug to be fixed, i.e., a typical implementation activity:

> "doesn't work: Depending on the compression engine used, compressed bytes may differ. False errors would be reported. `assertTrue(‘‘File content mismatch’’, FILE_UTILS.contentEquals(...)));`."

Hence, while the existing categorizations contribute to the understanding of the SATD phenomenon, we think that a different categorization is required as a basis for the design of tools that can help support SATD resolution. By providing a more fine-grained classification of the problems experienced by contributors we expect that

more actionable insights can be obtained from SATD. Thus, we ask the following research question:

RQ$_1$: *What kind of problems do SATD annotations describe?*

To address **RQ$_1$**, we use 1038 SATD comments sampled from the dataset of da S. Maldonado et al. (2017) to perform a fine-grained classification. We classify SATD comments from the point of view of *their textual content*, as opposed to the *software development life-cycle*, as it was done in previous work (da S. Maldonado and Shihab, 2015; Bavota and Russo, 2016). Our taxonomy has been created by adopting a bottom-up strategy (i.e., what do SATD comments mention?) rather than a top-down (i.e., how do SATD comments map onto a software development life-cycle?). This leads us towards a taxonomy featuring nine top-level categories specialized into 32 sub-categories. The taxonomy spotlights categories that are, on the one hand, crosscutting to the life-cycle and, on the other hand, more related to the reasons why SATD was admitted and to the goal developers want to achieve.

Different authors have studied the sentiment and emotions expressed by developers (Mäntylä et al., 2016; Murgia et al., 2014; Novielli and Serebrenik, 2019; Lin et al., 2021). In particular, Mäntylä et al. (2016) and Murgia et al. (2014) studied emotions expressed in the context of issue reports, finding that there appears to be a link between issue priority and complexity and negative emotions present in issue reports. When describing TD, developers could express the same concept in neutral or in a rather negative fashion. For instance, in the following comment from JRuby the author expresses a negative attitude:

"// Yow...this is still ugly"

Several authors hypothesize that the expression of negative sentiment may be a proxy for the priority of a problem to be solved (Gachechiladze et al., 2017; Uddin and Khomh, 2017; Lin et al., 2019). In other fields, such as marketing, negative sentiment has a clear meaning. For instance, customers give greater weight to negative information (Wright, 1974), and negative reviews are more useful to customers' decisions than positive ones (Casaló et al., 2015; Sparks and Browning, 2011). However, to the best of our knowledge, nobody has studied how priority is expressed in different kinds of software development issues—and in particular TD-related issues—and whether developers use negative sentiment to indicate priority. This leads us to address the following research question:

RQ$_2$: *How do developers annotate SATD that they believe requires extra priority?*

To address **RQ$_2$**, we ask developers how they would annotate TD they believe requires more priority, and specifically whether they would (i) use negative sentiment to indicate higher priority and (ii) interpret a comment with negative sentiment as an indication of higher priority. Our results show that while the perception of negativity as a proxy for priority is not necessarily shared by all developers, it is still sufficiently common to confirm this relation as hypothesized in the previous work (Gachechiladze et al., 2017; Uddin and Khomh, 2017; Lin et al., 2019).

Other than that, we also seek to understand whether developers believe that the expression of negative sentiment in annotating TD is an acceptable practice. In particular, if developers believe that expressing negativity is not acceptable, then they might feel obliged to suppress it. Suppressing negative emotions is an example of emotional labor—i.e., the "process by which workers are expected to manage their feelings in accordance with organizationally defined rules and guidelines" (Hochschild, 1983)—in software developers (Serebrenik, 2017). While traditionally, software development has been stereotyped as a job less likely to induce emotional labor (Diefendorff and Richard, 2003), communication between developers and their collaborators makes their job an intrinsically social activity (Storey, 2012). Therefore, we ask the following:

RQ$_3$: *Do developers believe that the expression of negative sentiment in SATD is an acceptable practice?*

To address **RQ$_3$**, we directly ask open-source developers whether they believe that expressing negativity when annotating TD is an acceptable practice.

Furthermore, certain kinds of TD may be expressed with a different sentiment. For example, an issue affecting the system's functionality may be perceived as more critical than a documentation or maintainability issue, and therefore be expressed more negatively. This leads us to address the following research question:

RQ$_4$: *How does the occurrence of negative sentiment vary across different kinds of SATD annotations?*

To address **RQ$_4$**, we follow two different approaches, i.e., (i) we study the sentiment polarity of the 1038 SATD comments used for addressing **RQ$_1$**, and (ii) we use a survey asking respondents to draft SATD comments for different scenarios. The latter is used since that, within a specific open-source project, developers might not feel free to express the emotions they experience (Hochschild, 1983). We consider as non-negative all comments merely stating the problem or suggesting an improvement, e.g., "TO DO : delete the file if it is not a valid file", while we consider as negative all comments expressing a negative attitude, e.g., "TODO : YUCK!!! fix after HHH-1907 is complete".

From the answers given by our survey respondents to **RQ$_2$** we learn that open-source developers use links to external systems, such as bug identifiers, to annotate high priority SATD. Moreover, in a survey-based study on task annotations, Storey et al. (2008) found that developers tend to include additional references or information in task annotations. To better understand this phenomenon we investigate our last research question:

RQ$_5$: *To what extent do SATD annotations belonging to different categories contain additional details?*

To address **RQ$_5$**, we combine manual and automatic labeling of the comments from the dataset of da S. Maldonado et al. (2017) with manual labeling of the comments drafted by the survey respondents.

This paper is a follow-up to our previous work (Fucci et al., 2021). In this journal article, we extend our previous study in the following way:

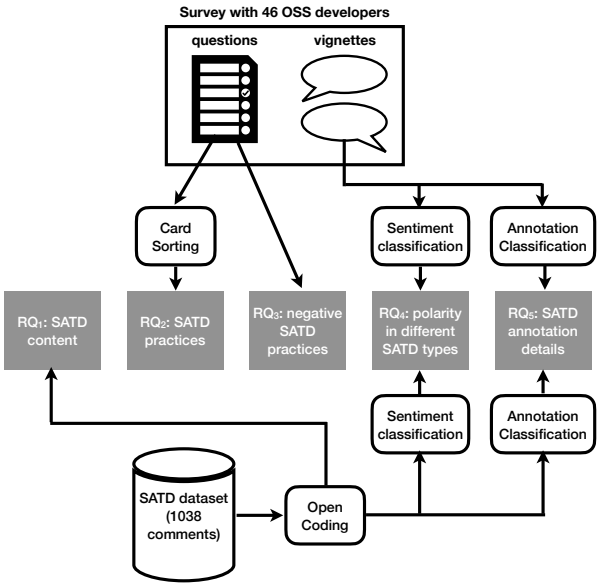

Fig. 1 Methodology

- We seek a better understanding of the TD annotation practices of open-source developers, and to that end we design and discuss a survey in which we ask open-source developers to (i) provide us with insights about their TD annotation practices, and (ii) draft SATD comments for five different scenarios;
- We add two new research questions: $\mathbf{RQ}_2$ and $\mathbf{RQ}_3$, which we address considering the results of our survey;
- We extend two existing research questions ($\mathbf{RQ}_4$ and $\mathbf{RQ}_5$) with the results of the survey.

By studying the annotation practices of developers we hope to better understand how developers use and perceive different kinds of SATD. In turn, this should help developers better triage and prioritize TD, and allow researchers to better understand how SATD containing negative sentiment influences, and is perceived by, developers. The full dataset, files used during the annotation, and qualitative data gathered during the survey, are publicly available.[1]

2 Study Design

To address the research questions stated in the introduction, we combine two different analyses, as depicted in Fig. 1. On the one hand, we take a sample of 1038 from an existing curated dataset of SATD comments (da S. Maldonado et al., 2017), and

[1] `https://figshare.com/articles/online_resource/Self-Admitted_Technical_Debt_and_Comments_Polarity_An_Empirical_Study/17024294`

Table 1 Number of SATD comments in the original dataset and in the sampled ones.

SATD Type	Initial Dataset	Without Duplication	Sampled
Defect	472	350	116 (11%)
Design	2703	2260	657 (63%)
Documentation	54	49	39 (4%)
Implementation	757	550	183 (18%)
Test	85	80	43 (4%)
TOTAL	4071	3289	1038

categorize their content (to address $\mathbf{RQ}_1$, and, by further classifying the presence of additional references in the comments, $\mathbf{RQ}_5$), and sentiment (to address $\mathbf{RQ}_4$). On the other hand, we survey 46 open-source developers to understand their perception to negative sentiment in SATD, and way they express priority in SATD. The survey is composed of two parts: (i) questions about SATD practices (addressing $\mathbf{RQ}_2$ and $\mathbf{RQ}_3$), and (ii) vignettes (Rossi and Nock, 1983; McNamara et al., 2018; Palomba et al., 2021) depicting realistic scenarios where developers can admit TD, and for which we ask survey participants to write possible SATD comments. The latter further contribute to answering $\mathbf{RQ}_4$ and $\mathbf{RQ}_5$.

2.1 Addressing $\mathbf{RQ}_1$: SATD content coding

To study the content of SATD comments we take an existing dataset of SATD comments and perform open coding of this dataset.

2.1.1 Dataset

We start from a curated dataset of SATD comments by da S. Maldonado et al. (2017), consisting of 4071 SATD comments belonging to 10 different open-source Java projects. These comments were classified by da S. Maldonado et al. (2017) into five categories (DEFECT, DESIGN, DOCUMENTATION, IMPLEMENTATION, and TEST). Note that IMPLEMENTATION debt also includes REQUIREMENT debt from the original taxonomy of da S. Maldonado and Shihab (2015).

First, we remove 782 duplicated comments (i.e., comments having the same content but attached to different source code elements) since our focus is on the comments' content. After the removal, we manually analyze a statistically significant random-stratified sample (strata are the SATD comments types in the initial dataset) accounting for 1038 SATD comments (confidence interval of 3.33% for a confidence level of 99%). Specifically, as reported in Table 1, our sample has the same percentage of SATD comments types as the initial dataset, guaranteeing that each SATD type is well represented in our study. For instance, our dataset without duplication counts 350 SATD belonging to DEFECT, i.e., $\simeq 11\%$ over the total number of SATD comments (3289), and in our sample, we have manually analyzed 116 SATD comments in the same category that accounts for 11% of the total number of SATD comments being analyzed.

2.1.2 Data analysis

To derive a taxonomy for SATD contents, we follow a card-sorting procedure, and specifically a cooperative (multiple annotators) open (no predefined categories) card-sorting (Spencer, 2009). This step has been conducted by the authors of the companion paper (Fucci et al., 2021).

In the first round, two of the authors independently created labels for 108 SATD comments randomly chosen from the dataset without duplication in proportion to each SATD type. Once completed, the two annotators discussed their labels, i.e., also resolving inconsistencies and redundancies, and grouped the tags into a hierarchy. After that, two different authors reviewed the initial set of created labels, in turn suggesting improvements, obtaining a taxonomy featuring 11 high-level categories specialized into 26 sub-categories.

In the second round, two authors used the first version of the taxonomy to label a different set of 115 SATD comments randomly picked from the dataset without duplicated instances and, again in proportion to each SATD type. Specifically, while reading a SATD comment content, the annotator could choose to reuse an existing label or to add a new one. Upon completion, the two annotators solved inconsistencies and evaluated the introduction of newly added labels. The updated version of the taxonomy has been sent to two different authors, that after some improvements ended up with a taxonomy featuring ten high-level categories specialized into 28 sub-categories. More specifically, two high-level categories have been used as specializations of other categories and one has been added (see details in the online dataset).

In the third round, using the same process, the authors manually analyzed 114 SATD comments. As a result, they obtained a new modified version of the taxonomy made up of 11 high-level categories, of which two are newly introduced ones and one became a sub-category. The high-level categories were properly specialized into 36 sub-categories, five of which were not reported in the previous version.

This final version of the taxonomy has been used to label the remaining 701 comments that were randomly assigned to four authors, such that each SATD comment was independently analyzed by two of them. Also in this case, the annotators could either use the existing labels or create a new one if no one fitted a specific comment. As it happens in teamwork card-sorting (Spencer, 2009), newly introduced labels (groups) became immediately available also for other annotators. Upon completion, the annotators discussed their classifications resolving inconsistencies, and revised the taxonomy. During the last round, the authors did not introduce any new high-level category while using two of them to specialize existing ones, even if there is the introduction of two new sub-categories. In summary, since in our last round no new high-level categories are introduced, the identified taxonomy is general enough. However, this does not exclude that, in the future, further contents could emerge and be therefore included in the taxonomy.

To address $\mathbf{RQ}_1$, we present our final version of the taxonomy, reporting for each category the number of SATD comments belonging to it together with some examples aimed at explaining the meaning of the category.

2.2 Addressing $\mathbf{RQ}_2$ and $\mathbf{RQ}_3$

For both $\mathbf{RQ}_2$ and $\mathbf{RQ}_3$, we seek to understand how developers annotate SATD that is more important, and whether they believe the annotation of TD with negative sentiment is an acceptable practice. Therefore, we use a survey to ask open-source developers whether they use a negative sentiment as a proxy for SATD priority and whether they consider the expression of negative sentiment in annotating TD as an acceptable practice. Specifically, we ask the questions shown in Table 2.

Table 2 Survey Questions

Question	Response Type
When writing source code, how often do you write source code comments indicating delayed or intended work activities such as TODO, FIXME, hack, workaround, etc.?	*Never, Rarely (Less than once a month), Sometimes (Monthly), Often (Weekly), Very often (Daily)*
When authoring comments that describe a problem, how often do you write negative source-code comments indicating delayed or intended work activities such as TODO, FIXME, hack, workaround, etc.?	*Never, Rarely (Less than once a month), Sometimes (Monthly), Often (Weekly), Very often (Daily)*
How often do you come across negative source-code comments indicating delayed or intended work activities such as TODO, FIXME, hack, workaround, etc.?	*Never, Rarely (Less than once a month), Sometimes (Monthly), Often (Weekly), Very often (Daily)*
Suppose you believe that an issue requires extra priority, how would you usually indicate this in a comment indicating delayed or intended work activities such as TODO, FIXME, hack, workaround, etc.?	*Open-text*
While writing a comment describing an issue in the source-code, I am more likely to write negative comments for issues that I believe are more important.	*Strongly disagree, Disagree, Neutral, Agree, Strongly agree*
Writing negative comments to assign extra priority to issues in the source-code is an acceptable practice.	*Strongly disagree, Disagree, Neutral, Agree, Strongly agree*
Whenever I come across a source-code comment describing a problem that is particularly negative, I interpret this as a more important issue than a source-code comment describing a problem that is more neutral.	*Strongly disagree, Disagree, Neutral, Agree, Strongly agree*

To learn the methods used by open-source developers to annotate high priority SATD ($\mathbf{RQ}_2$), two authors performed an open card-sort (Spencer, 2009) on the responses to the open-question on how developers annotate high priority TD, and a third author resolved the conflicts. Each response can be assigned to multiple cards, based on the content of the answer being provided. As it is widely hypothesized that negative sentiment in SATD is used to indicate priority (Gachechiladze et al., 2017; Uddin and Khomh, 2017; Lin et al., 2019), we augment the open question with a set of closed questions on whether developers interpret negative sentiment as a proxy for priority, as well as, whether they are more likely to write negative SATD for high priority issues.

Finally, the closed questions on whether developers consider the expression of negative sentiment in SATD as an acceptable practice, and how frequently develop-

ers come across or author negative SATD allows us to determine whether open-source developers believe that this is an acceptable practice ($\mathbf{RQ}_3$). We statistically compare the three distributions (SATD annotation, SATD negative annotation, and encountering negative SATD) using (a) a combination of the Kruskal-Wallis test (1952) with three post-hoc pairwise Wilcoxon rank-sum tests with the *p*-values adjusted to control for the false discovery rate, as recommended by Benjamini and Hochberg (1995), and (b) a more recently proposed multiple comparisons method of Konietschke et al. (2012).

2.3 Addressing $\mathbf{RQ}_4$

To address $\mathbf{RQ}_4$, we need to understand whether negative sentiment is more or less likely to occur for specific categories of SATD. To this aim, we analyze the sentiment of comments from the dataset of da S. Maldonado et al. (2017) and from a set of SATD comments drafted by respondents of the survey. This way, we combine results of two different kinds of studies, i.e., one conducted by mining SATD comments from real projects, and another in which survey participants are involved. Section 2.3.1 discusses the labeling protocol used to assign a sentiment polarity to SATD comments. The labeling procedure was originally consolidated on the set SATD comments from da S. Maldonado et al. (2017) and then applied to annotate also the SATD comments collected through the survey. Section 2.3.2, instead, discusses the survey in which we ask respondents to draft SATD comments. We would like to emphasize that, albeit obtained using the same protocol and guidelines, the negative sentiment distribution in the two datasets of SATD comments collected through software repository mining and survey, respectively, might not be directly comparable. Specifically, differences in the proportion of labels that we might observe could be related to the fact that each category in our taxonomy is made up of several different sub-categories, each one representing a specific development scenario. In our survey, we could address only a selection of such scenarios depicted by our vignettes (detailed in Table 3). As such, the SATD scenarios included in the Maldonado et al. dataset are higher in number (and more diverse in terms of specific SATD sub-categories) than the ones included in our survey.

2.3.1 Sentiment labeling of SATD

To address $\mathbf{RQ}_4$, all 1038 comments sampled from the dataset of da S. Maldonado et al. (2017) for $\mathbf{RQ}_1$ have been manually annotated with their sentiment polarity (Section 2.1.1), together with the comments drafted by the respondents of the survey (Section 2.3.2). In principle, we could have used automated tools to classify comments' polarity. However, previous work has shown that even SE-customized sentiment analysis tools may fail to produce a reliable annotation (Lin et al., 2018), especially if they are fine-tuned using a gold standard collected on a platform that is different from the one targeted for the study (Novielli et al., 2020). For this reason, we decided to perform a prelminary assessment of the performance of publicly available, SE-specific tools for sentiment analysis, as described in the following. To this

aim, we leverage a multiple annotator manual analysis and to create a gold standard against wich to compare the outcome of three publicly available sentiment analysis tools that have been specifically tuned for the software engineering domain, i.e., SentiStrength-SE (Islam and Zibran, 2018), Senti4SD (Calefato et al., 2018a), and SentiCR (Ahmed et al., 2017).

By definition, SATD describes an undesirable situation, so we do not expect to observe many positive comments and opt to classify sentiment as either *negative* or *non-negative*, where the latter category includes both positive and neutral comments. Comments conveying both positive and negative sentiment are labeled as *mixed*. We label as negative, comments containing expressions that clearly communicate negative sentiment, e.g., emotions or negative opinions about the underlying code, beyond the negativity inherent in problem reporting, e.g., SATD comments.

Determining sentiment for a text is a subjective task, i.e., the labels given by individuals depend on their cultural background, upbringing, and interpretation of the comment (Scherer et al., 2004). As such, following clear annotation guidelines is recommended for enabling reliable annotation (Novielli et al., 2018). For this reason, we defined a set of annotation guidelines by conducting a pilot labeling study. We randomly sampled 32 comments from the 1038 comments of the dataset of da S. Maldonado et al. (2017) and asked each author to label them individually, based on their subjective perception of each comment polarity. Then, we jointly discussed disagreements in a plenary session, resolving conflicts and addressing ambiguities in the definition of negative sentiment. Based on the results of our discussion, we drafted our coding guidelines to be used for the labeling study as follows:

- *negative*: the comment expresses negative sentiment about the underlying source-code (e.g., “this method is a nightmare”); specifically, we considered the following factors: terms highlighting urgency (like the presence of terms such as “asap” and “urgent”), the presence of multiple exclamation and question marks, as well as, the presence of some keywords being reported in upper case such as the term NOT in the comment: “// the plot field is NOT tested”;
- *non-negative*: the comment expresses either positive or no sentiment about the code referenced in the comment (e.g., “TODO: Why is this a special case?”);
- *mixed*: the comment expresses both positive and negative sentiment (e.g., “This is a fairly specific hack for empty string, but it does the job”).

We used the 32 SATD comments manually labeled during our pilot to evaluate the accuracy of the selected SE-specific sentiment analysis tools. We apply the tools “off-the-shelf”, i.e., without further tuning or training (Novielli et al., 2021). Looking at the agreement between manual labels and the tools’ predictions, we found that Senti4SD has the highest F-1 score (0.69), lower than the one reported by the authors of the tool (0.87) on the original training platform (Calefato et al., 2018a), i.e., Stack Overflow. By inspecting disagreements we found that some negative comments were missed by the tool due to the presence of a lexicon which is specific to SATD comments. For instance, “FIXME: Big fat hack here, because scope names are expected to be interned strings by the parser” is labeled as negative by the human judges but classified as neutral by Senti4SD. We conclude that the operationalization of sentiment by tools does not align with our operationalization of sentiment in SATD. For

this reason, we decide to manually label both the remaining SATD comments in the dataset and the SATD comments drafted by the developers in the survey.

To label the 1038 comments from da S. Maldonado et al. (2017), we divide the comments in our sample, excluding the ones already labeled in our pilot study (1006), over six annotators, including the authors of this paper, such that each annotator labeled an equal number of comments per SATD category, and each comment was labeled by at least two annotators, to mitigate the presence of any biases between annotators and over SATD categories. Moreover, to ensure reliability and consistency of our labeled dataset, we resolved all disagreements in plenary sessions involving all annotators. The agreement between the annotators for the sentiment labeling of the comments from the dataset of da S. Maldonado et al. (2017) is moderate, with a Krippendorff's α of 0.455 (Krippendorff, 2012), which is in line with agreement reported by previous studies on developers' sentiment annotation in short comments from software development platforms (Murgia et al., 2014). Lastly, to understand for what SATD categories negative sentiment is more likely to occur, and how this differs over the SATD categories of the taxonomy constructed in $\mathbf{RQ}_1$, we use a pairwise proportion test (Newcombe, 1998). Specifically, for each category, we compare the proportion of negative and non-negative comments. Because of the multiple comparisons, we control for the false discovery rate by adjusting the p-values using the Benjamini-Hochberg procedure (Benjamini and Hochberg, 1995). The Benjamini-Hochberg procedure adjusts the p-values as follows: Let $p_1, .., p_n$ be a collection of p-values ordered from the smallest to the largest one. For p_i the adjusted value p_i' is computed as $p_i * n/i$ (topped at 1). Hence the largest p-value of the collection is never modified, and the smallest p-value is increased most.

2.3.2 Survey

In addition to the survey questions described in Section 2.2, we ask the respondents to draft SATD comments for five different scenarios selected from the taxonomy identified in $\mathbf{RQ}_1$. Specifically, we took the five most populous categories and, for each of them, we designed a vignette representing the category (Rossi and Nock, 1983) (see Table 3).

During the survey we only showed the respondents the text of the vignette but not the category name, to ensure that respondents are not biased by the category name. For each vignette, we want to understand the TD comments that developers would write. Hence, after each vignette we also ask the following three questions:

(a) How likely will you add a comment recording this observation? *Very unlikely, Somewhat unlikely, Neither likely nor unlikely, Somewhat likely, Very likely.*
(b) What are your reasons for deciding to write a comment or not? *Open-text*.
(c) If you would add a comment, please draft the comment you would add in this situation? *Open-text*.

For each vignette, we label the comments written for question (c) with their sentiment polarity using the labeling procedure and operationalization of sentiment described in Section 2.3.1. Agreement between the annotators, over the SATD comments drafted for the survey was moderate with a Krippendorff's α of 0.503 (Krippendorff, 2012). Additionally, to learn whether negative sentiment is more likely to

Table 3 Vignettes used in the survey to describe different SATD categories.

SATD category	Vignette
Functional issue	You are working on an open-source mail client and you are working on a new feature. You observe that the auto-completion of e-mail addresses is broken: It should complete addresses using e-mail addresses from the address book and e-mail addresses used recently. However, it only uses addresses from the address book for the auto-complete. You do not have time to fix this immediately.
Partially/not impl. func.	You are working on an open-source mail client. You observe that one method is not yet finished: If the method detects invalid input it should raise a dialog window, and this is not currently implemented. You do not have time to fix this immediately.
Poor impl. choice	You are working on an open-source diagramming application. You observe that a code fragment is copied over and over again. You do not have time to refactor this immediately.
Documentation	You are working on an open-source diagramming application. You observe that there is a method without any documentation, in violation of the agreed upon coding guidelines. You do not have the time to read the method and write the documentation yourself.
Wait	While working on an open-source Java GUI application and you are implementing a new feature, however, to implement this feature you are dependent on an external API that is not yet available.

occur in specific categories we use a set of pairwise proportion tests (Newcombe, 1998), similarly to Section 2.3.1.

To ensure that the order of the vignettes does not impact the results obtained from the survey we create several survey variants in which we shuffle the order of the vignettes. In the analysis, we merge the results of the surveys with a different vignette order if there are no differences between the responses given for different survey variants corresponding to different orders. To determine whether the order in which we present the vignettes to the users influences the results we apply PERMANOVA (Anderson, 2017), which is a non-parametric equivalent to the Analysis of Variance (ANOVA). For each vignette we apply PERMANOVA with the dependent variable being the response to the closed question, i.e., "How likely will you add a comment recording this observation?", and the independent variable being the order in which the vignette was present in the survey. To account for the multiple comparisons we adjust p-values using the Benjamini-Hochberg procedure (Benjamini and Hochberg, 1995). For the vignette(s) where we find that the order influences the responses, we apply a post hoc pairwise PERMANOVA to determine which variants can be safely combined because the differences in the vignettes order did not influence the answers to the closed questions. When discussing the results we consider these subgroups separately.

2.4 Addressing $\mathbf{RQ}_5$: Identifying Additional Details in SATD

Developers use external references to annotate SATD that they believe is more important, including links to bug trackers, or bug ids. Additionally, from work by Storey et al. (2008), we know that developers tend to include: (i) references to another class,

method, plug-in, or module, (ii) developers' names or initials, (iii) references to bugs, (iv) URLs, (v) dates, and (vi) "memorable keywords" in SATD.

To understand how often these additional details occur in SATD we use a combination of manual labeling and automated detection to extract fields (i) through (vi) from the 1038 SATD comments. Due to the heterogeneity (as well as our unfamiliarity) with the practices of the projects that make up the dataset, and considering that in SATD comments keywords are mainly related to tags, e.g., TODO, FIXME, XXX etc. we have chosen to not identify "memorable keywords".

Firstly, we identify the following fields automatically:

- for class names, we search for all possible class names of a project, obtained from its git repository (all file versions from all branches), onto comments, using a case insensitive, word boundary match, and for methods references we use a simple regular expression ("$\backslash\backslash w+\backslash\backslash($", matching all words that contain one or more alphanumeric characters followed directly by an opening parenthesis);
- for bug references, we use the Fischer et al. approach (Fischer et al., 2003), e.g., matching JIRA-style references (e.g., "jruby-1234") or GitHub-style reference (e.g., "#1234");
- for URLs we match the following two regular expressions onto the SATD comments, i.e., `http://` and `https://`.

Also, while we initially detected bug-ids and dates (in this case matching various formats as "12 Jan 2002", or "20020112") automatically, we double-checked them manually because of the presence of several formats.

Based on a manual inspection of the dataset, we combine the results of the automatic detection with the manual labeling. Specifically:

(i) for class/method names and URLs we use the automated detection;
(ii) for developers names/initials and dates we rely on the manual labeling;
(iii) for bug-ids we combine the manual analysis with the automatic detection.

We report the occurrences of each field for each high-level category of the taxonomy, and evaluate how the perceptions of developers, as found by Storey et al. (2008), compare to the occurrences of these fields in the 1038 SATD comments.

Additionally, to understand how developers annotate SATD when they are asked to write comments in a more neutral setting, we manually label the comments drafted by respondents in Section 2.3.2 for the presence of names, dates, and references to bugs.

2.5 Survey Preparation and Sampling

To verify whether the survey discussed in Section 2.2 and 2.3.2 was understandable for developers, and to ensure that the survey takes ca. 10–15 minutes to complete we asked two non-academic developers to fill out a drafted version of the survey. Based on their feedback, we modified the wording of several questions to make the survey more clear.

The survey itself was prefaced with an informed consent form that is included as an appendix in the replication package and the survey has been approved by the

Ethical Review Board of the first author's institution. The population we target for this study are open-source developers, to make results comparable with the quantitative analysis conducted on the Maldonado et al. dataset. To reach developers within this population we used the following platforms:

- We sent out emails to the mailing lists of open-source software projects. The list of projects is identical to the list that was used for the study of Zampetti et al. (2021). This list also includes the mailing lists of five out of ten projects from the Maldonado et al. dataset (i.e., the ones for which we were able to access the mailing list) we used for the other part of the study. We did not limit survey participation to the projects from the Maldonado dataset to ensure larger participation in the survey. In total, we invited the developers of 93 open-source through the respective mailing lists, Discord, Slack, and Google Group channels.
- We posted the link to the survey to several Facebook and LinkedIn groups, which target open-source developers.
- We posted the survey on the Twitter accounts of the authors.
- We asked personal contacts for which we know that they contribute to open-source projects to fill out the survey.

Note that the question about how often respondents author SATD has been already posed before in a different study (da S. Maldonado et al., 2017). However, we include this question to understand whether our respondents are as familiar with SATD as in previous studies.

Finally, to ensure that we target only open-source developers we include a screening question asking whether the respondent contributed in an open-source project in the past three months. Moreover, to ensure that we collect no personal information we did not include any question asking about demographics, such as age, gender, or experience. In the authors' experience, the latter favors larger participation. Moreover, it was a constraint for the approval by our ethical committees.

3 Study Results

This section reports and discusses the study results, addressing the research questions formulated in the introduction.

3.1 Survey Responses

In total we obtained 46 responses to the survey, and in this section we discuss whether the order in which we presented the vignettes of the survey influenced the results obtained. None of the questions was mandatory, hence the number of responses for different questions might vary.

After the application of PERMANOVA (Anderson, 2017) to the five vignettes described in Section 2.3.2 we find that the responses for the vignettes of the macro-categories: Poor implementation choices, Partially implemented, Functional issues, and Wait, belonging to the different pools having a different ordering, can be safely

analyzed together, as the corresponding p-values are 0.71, 0.71, 0.52 and 0.95, respectively, i.e., all of them exceed the customary threshold of 0.05.

The responses to the macro-category Documentation are dependent on the order in which the vignette was included in the survey ($p \simeq 0.03$), and therefore we cannot merge all responses obtained for this vignette in different survey variants. The post hoc analysis revealed that there is a statistically significant difference between the answers obtained when the Documentation vignette is shown at the beginning of the survey, as the first or the second vignette (subgroup A—36 responses), as opposed to the answers obtained when the Documentation vignette is shown last (subgroup B—10 responses). We hypothesize that this difference can be attributed to how developers were biased by seeing documentation-related vignettes after having seen vignettes related to more critical issues (e.g., functional TD). In such cases, respondents might have been tempted to say that documentation is not important enough to write a SATD comment for. In conclusion, for this specific case, the ordering has influenced the results.

3.2 $\mathbf{RQ}_1$: What kind of problems do SATD annotations describe?

Fig. 2 depicts the taxonomy of SATD comments' content, obtained as described in Section 2.1.2: the small red boxes of Fig. 2 indicate the number of SATD comments (out of 1038) belonging to each category. Table 4, instead, shows the distribution and mapping between our high-level categories and the categories provided by da S. Maldonado et al. (2017). Note that, for some comments, we were not able to assign the leaf category while only the higher-level category. For instance, the SATD comment: "TODO: implement the entity for the annotation" in JFREECHART reports that the functionality is only partially implemented but does not contain any other information aimed at justifying why that happened.

Although our data came from a curated dataset (da S. Maldonado et al., 2017), we still found 40 instances that, according to our manual analysis, were not related to SATD (i.e., labeled as false positives). For instance, "Required otherwise it gets too wide" in SQL describes the design decision without indicating that it is suboptimal in any sense.

While (not surprisingly) most SATD comments highlight poor implementation choices (429 over 1038) mainly related to maintainability issues, as well as partially/not implemented functionality (229), we notice that functional issues (135) are not so frequent in our sample. Furthermore, we found 89 SATD comments classified as "Wait", meaning that a developer cannot improve the code or complete a functionality since they are waiting for a different event that has to occur in the same project or in a third-party component (e.g., "this is the temporary solution for issue 1011" in JFREECHART). As also reported in previous work (Bavota and Russo, 2016; da S. Maldonado and Shihab, 2015; Xavier et al., 2020), developers tend to admit TD also in artifacts that are different from the production code: indeed, we found 54 SATD comments dealing with documentation issues, and 36 SATD comments related to the test code. Finally, we found 21 SATD comments describing misalignment between requirements and design or implementation, as well as problems with

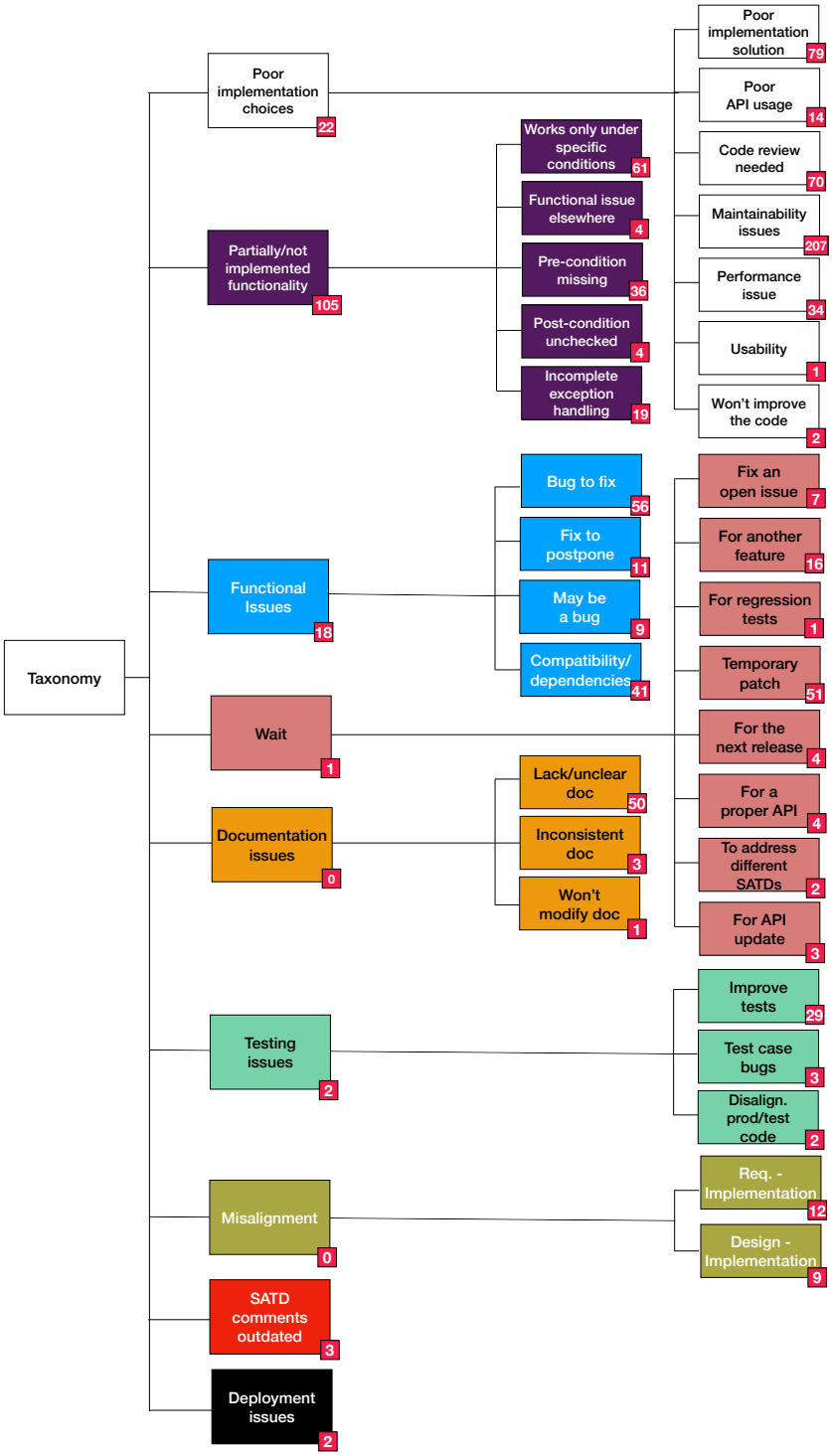

Fig. 2 Discussions contents in SATD comments

deployment (2) and SATD comments that are left in the code while not describing a TD anymore (3).

Next, we elaborate on each of the nine high-level categories of our taxonomy.

Poor implementation choice. This category includes (i) maintainability issues, (ii) poor implementation solutions, (iii) asking for code review, i.e., the developer is not sure of the actual design, (iv) performance issues, (v) poor API usages, i.e., reliance on a third-party component without actually understanding the proper way to use it, (vi) lack of intention to improve the code despite the awareness that it is not in the right shape, and (vii) usability issues.

Maintainability issues constitute the category with the highest number of samples, not merely within "Poor implementation choices" but overall, covering 20% of the comments. The latter is in line with the results reported by Zampetti et al. (2021) highlighting that more than 60% of the open-source developers in their study use annotations to indicate the need for maintainability improvement. Unsurprisingly, many maintainability issues require a refactoring activity such as a better distribution of responsibilities among software components (e.g., "TODO: We should have all the information that is required in the NotationSettings object" in ARGOUML), proper reuse of features (e.g., "TODO: Reuse the offender List" in ARGOUML), or else the replacement of magic numbers with proper constant variables (e.g., "// TODO: define constants for magic numbers" in ARGOUML).

Furthermore, we found 79 SATD comments reporting that the implemented solution has to be improved, e.g., "EATM This might be better written as a single loop for the EObject case" in EMF highlighting the need for simplifying the actual implementation removing a control structure. In other cases, the developers criticize the implementation choices and ask for a code review, e.g., "FIXME: Is "No Namespace is Empty Namespace" really OK?" in APACHE-ANT or "TODO: this assumes ranges are sorted. Is this true?" in ARGOUML. The latter confirms the findings by Ebert et al. (2018) who highlight that 8% of questions during code reviews express attitudes and emotions. Specifically, their manual coding shows that developers express doubts through criticisms ($\simeq 5\%$) inducing critical reflection in the interlocutor.

Finally, concerns related to the use of APIs and performance are reflected in the SATD comments: e.g., "FIXME: don't use RubyIO for this" in JRUBY alerts developers to replace the existing API for a specific task, while "TODO replace repeated `substr()` above and below with more efficient method" in JMETER indicates performance issues.

Partially/Not implemented functionality groups the SATD comments reporting that a feature is not ready yet. While, on the one hand, we found many cases (105) in which the SATD comment simply reports that the implementation is missing without adding any further details, on the other hand, we found comments indicating what is specifically missing from the implementation: e.g., a precondition ("TODO: delete the file if it is not a valid file" in ANT), or a postcondition check ("FIXME: Make bodyNode non-null in parser" in JRUBY).

We found comments clarifying that the feature works only under specific conditions (61) as "If c2 is empty, then we're done. If c2 has more than one element, then the model is crappy, but we'll just use one of them anyway" in ARGOUML. Our results are in line with findings from Zampetti et al. (2021) who report that about

Table 4 Distribution of our taxonomy top-level categories and how they map onto da S. Maldonado et al. (2017) categories.

Macro-category	Defect	Design	Doc.	Impl.	Test	Total
Poor implementation choices	22	361	2	43	1	429
Partially implemented	27	94	5	100	3	229
Functional issues	48	68	0	18	1	135
Wait	6	76	0	6	1	89
Documentation issues	0	19	30	4	1	54
Testing issues	0	1	0	0	35	36
Misalignment	1	13	2	5	0	21
SATD comments outdated	2	1	0	0	0	3
Deployment issues	1	0	0	1	0	2
False positive	9	24	0	6	1	40
TOTAL	116	657	39	183	43	1038

half of their survey respondents use SATD to report incomplete features, as well as, features exhibiting incorrect behavior under certain conditions.

Finally, some comments (4) indicate that the implementation is absent due to problems elsewhere: e.g., "Predecessors used to be not implemented, because it caused some problems that I've not found an easy way to handle yet. The specific problem is that the notation currently is ambiguous on second message after a thread split." in COLUMBA.

Functional issue includes all cases directly or indirectly related to the presence of a bug in the system and constitutes the third-largest category of SATD comments in our taxonomy. Unsurprisingly, most of them highlight the presence of a bug that should be fixed immediately, (i.e., 56 comments belonging to the Bug to Fix category): e.g., "FIXME: If `NativeException` is expected to be used from Ruby code, it should provide a real allocator to be used. Otherwise `Class.new` will fail, as will marshaling. JRUBY-415" in JRUBY. 11 SATD comments, instead, indicate the presence of misbehavior that is acceptable even though a better solution must be found, i.e., Fix to postpone: e.g., "this will generate false positives but we can live with that" in ANT.

The most interesting sub-category groups compatibility and dependency issues that are also not very easy to address (41 SATD comments). For instance, we found comments indicating that the code is not able to work properly in specific environments, e.g., "`waitFor()` hangs on some Java implementations" in JEDIT, or cases where the actual implementation inherits a bug from an external API being used, e.g., "Workaround for JDK bug 4071281 [...] in JDK 1.2" in JEDIT.

Wait includes all SATD comments in which the developer reports that the code has to be improved and/or completed once a different event occurs. In many cases (51) the comments report that the code is a temporary patch that needs to be removed later on, e.g., "TODO: temporary initial step towards HHH-1907" in HIBERNATE. Furthermore, 16 comments state that the code is not in the right shape since it requires a different feature to be ready first, e.g., "todo : remove this once `ComponentMetamodel` is complete and merged" in HIBERNATE. There are also seven SATD comments

where developers admit the presence of a TD in the code that cannot be addressed before an issue already opened is not fixed, e.g., "// TODO: This whole block can be deleted when issue 6266 is resolved" in ARGOUML. Differently from the comments belonging to the Fix to Postpone leaf in the Functional issue category where the TD corresponds to the functional issue for which developers do not have to rush to fix them, in this case the functional issue is simply the event developers are waiting for before removing a TD from the code. An interesting phenomenon related to waiting is an SATD comment requiring other SATD comments to be fixed (2), e.g., "TODO: simply remove this override if we fix the above todos" in HIBERNATE. We found four comments in which developers need to wait for a proper API to be found, e.g., "This really should be `Long.decode`, but there isn't one. As a result, hex and octal literals ending in 'l' or 'L' don't work." in JEDIT. Differently from the comments belonging to the Poor API usage leaf under the Poor Implementation Choices category where the TD corresponds to an inappropriate API usage, here we group comments where developers admits the presence of a workaround that must be removed once an appropriate API is found, i.e., the external event developers are waiting for.

Recently Maipradit et al. (2020b) have looked at "on-hold" SATD, i.e., debt containing a condition highlighting that a developer is waiting for a certain event or an updated functionality having been implemented elsewhere, that maps onto our "Wait" category. Our results confirm what found by Maipradit et al. (2020b), i.e., around 8% of the SATD comments contains a waiting condition, however, our taxonomy enlarges the set of possible events a developer is waiting for, indeed Maipradit et al. (2020b) only considered bugs to be fixed, or new releases/versions of libraries.

Documentation issues (54 over 1038). Many cases are related to the need for documenting a specific method/class such as "FIXME This function needs documentation" in COLUMBA. However, we also found three cases describing inconsistencies in the related documentation, e.g., "UML 1.4 spec is ambiguous - English says no Association or Generalization, but OCL only includes Association" in ARGOUML, and one case in which the author is reporting that the documentation cannot be modified even if it is required to modify it, i.e., "TODO: Currently a no-op, doc is read only" in ARGOUML.

Testing issues. 36 SATD comments refer to test code, including (i) untested features, e.g., "TODO add tests to check for: - name clash - long option abbreviations/" in JMETER, (ii) bugs in the current test suite, e.g., "this is the wrong test if the remote OS is OpenVMS, but there doesn't seem to be a way to detect it" in ANT, or (iii) misalignment of the test code with the production code, e.g., "TODO: [...] An added test of `isAModel(obj)` or `isAProfile(obj)` would clarify what is going on here" in ARGOUML.

Misalignment groups the SATD comments in which the developers report a mismatch between (i) requirements and implementation (12) such as "TODO: The Quickguide also mentions [...] Why are these gone?" in ARGOUML, where the developers ask whether the current implementation deviates from what was reported in the specification, or (ii) design and implementation (9) such as "TODO: This shouldn't be public. Components desiring to inform the Explorer of changes should send events" in ARGOUML, clearly highlighting a deviation from what was reported in the design document.

We also found three instances belonging to **outdated SATD comments** in which the SATD comment no longer reflects the source code evolution e.g., "todo: is this comment still relevant ??" in ANT. This category generally belongs to the problem of comments being outdated with respect to source code. For simple cases, especially related to comments explaining statements' behavior, detection approaches have been proposed (Fluri et al., 2007) and empirical studies have been carried out. As regards SATD, it is still possible that in many circumstances SATD comments remain in the system even after the mentioned problem has been addressed.

Two SATD comments reporting **Deployment issues**: the first one in ARGOUML (i.e., "As a future enhancement to this task, we may determine the name of the EJB JAR file using this display-name, but this has not be implemented yet.") highlights the need for improvements to the overall deployment phase while constructing the application jar. The second one in ANT (i.e., "the generated classes must not be added in the generic JAR! is that buggy on old JOnAS (2.4)"), reports about a problem while selecting the components to involve in the jar.

To understand the difference between our categories and those by da S. Maldonado et al. (2017), Table 4 shows how SATD comments belonging to different categories of their taxonomy are mapped to our high-level ones. Although 48 over 116 SATD comments in the "Defect" category are mapped onto our "Functional issues", the remaining SATD comments are mainly scattered onto the "Partially/not implemented functionality" and "Poor implementation choices". As an example of the former, the comment "TODO: we didn't check the height yet" in JFREECHART, originally considered as a defect SATD, has been categorized as a "Partially/not implemented functionality" since its content has nothing reporting the presence of a bug in the system due to the lack of a pre-condition check. As regards the latter, instead, "TODO: This method doesn't appear to be used." in JMETER mostly highlights possible maintainability issues, therefore it has been categorized as a "Poor implementation choice".

Similarly, while "Design" SATD comments mainly belong to our "Poor Implementation Choices" category (361 over 657), some refer to waiting (76), e.g., "Remember to change this when the class changes" in JMETER, partially implemented functionality (94), e.g., "TODO: complete this" in JFREECHART or functional issues (68), e.g., "TODO - is this the correct default?" in JMETER.

The "Implementation" SATD comments were originally labeled as "Requirement debt" by da S. Maldonado and Shihab (2015) and then renamed in their follow-up dataset. While, unsurprisingly, almost half of them belong to our "Partially/Not implemented Functionality" (which is indeed requirement debt, because the requirement has not been fully implemented), 43 cases are related to poor implementation choices, hence not related to requirements. For instance, there are comments in ARGOUML asking for code review, e.g., "TODO: Why is this disabled always?", or pointing out the presence of maintainability issues, e.g., "TODO: Reuse the offender List."

Finally, the categories of our taxonomy having a good fit with the ones of Maldonado et al. are "Documentation issues" and "Testing Issues". Still, in JMETER we found a documentation debt, e.g., "TODO Can't see anything in SPEC", we categorized as "Misalignment" since it relates to a discrepancy between specification and implementation, and either of the two can be wrong.

Table 5 To express that SATD should have higher priority developers recommend doing so outside of the source code or to use tags such as TODO, FIXME, or XXX.

Card Sorting Code	Occurrence
Should be discussed elsewhere (issue tracker, code review, mail, PM, backlog, tests)	19
Tag	14
Should not be indicated in the source code (alternative reporting mechanism is not indicated)	4
Rationale	2
Code should report an error	2
Not-ready work should not be merged	1
Tags make the code not ready to merge during code reviews	1
Tag followed by the name of the person who has to address it	1
Tag followed by the bug ID detailing the issue in the issue tracking system	1
Use specific keywords in the comment like issue, ASAP and high-priority	1

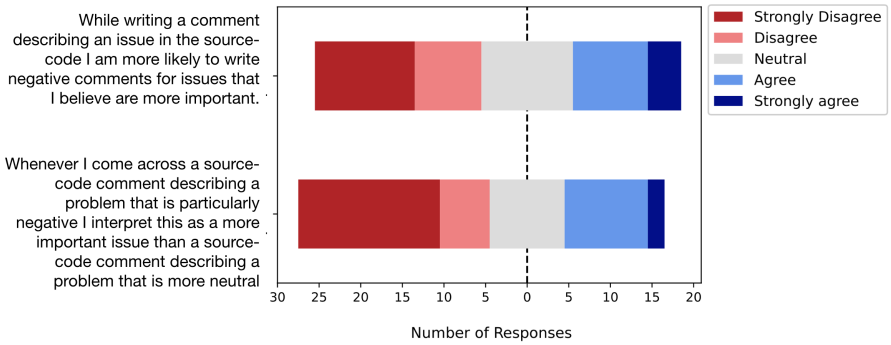

Fig. 3 Negativity in SATD comments and their priority

> **RQ_1 Summary:** We categorized the sample of 1038 SATD comments into nine top-level categories, separating functional issues from partially implemented functionality and poor implementation choices. We also considered on-hold TD ("Wait") as a specific category with 89 instances, while Documentation and Testing issues were almost mapped onto the categories of da S. Maldonado et al. (2017). We noticed how our content-based SATD categorization does not have a one-to-one mapping to lifecycle-based categories.

3.3 RQ_2: How do developers annotate SATD that they believe requires extra priority?

As explained in Section 2.2 to answer RQ_2, we have asked developers how they would indicate that a source code problem should be addressed with high priority. As this was an open question we performed card sorting among the provided answers, which results are summarized in Table 5. Survey respondents recommend doing so outside of the source code or to use tags such as TODO, FIXME, or XXX (with or without additional information such as bug ID or name of the person responsible for

fixing). Interestingly, a small group of respondents suggests that high priority issues should prevent the normal way of working through either run-time errors or blocking the code from being merged.

To verify what conjectured in literature about the relationship between negative comments and priority (Gachechiladze et al., 2017; Uddin and Khomh, 2017; Lin et al., 2019), we asked developers whether they are more likely to *write* negative comments for high-priority SATD comments, as well as, whether they are likely to *interpret* negative comments as conveying higher priority.

Each of these questions has been answered by 44 respondents out of 46. By inspecting Fig. 3, it is possible to observe that 29% of the respondents are more likely to express negativity when the issue has high priority and a similar share of respondents (27%) will interpret negative SATD comments as reflecting higher priority. Therefore, while the perception of negativity as a proxy for priority is not necessarily shared by all developers, we can still confirm the relation hypothesized in the previous work (Gachechiladze et al., 2017; Uddin and Khomh, 2017; Lin et al., 2019), as there appears to be a sizable group of developers that are more likely to write or interpret negative comments as reflecting high priority.

RQ_2 Summary: Respondents confirmed use of tags in the source code to indicate extra priority. However, nearly half of them indicate that it would be better to open issues instead and, in some cases, to even avoid merging a change if it is not ready. Furthermore, 29% of developers reported that they would write a comment containing negative sentiment for problems with high priority. Similarly, 27% of respondents reported they interpret negative sentiment as an indication of higher priority.

3.4 RQ_3: Do developers believe that the expression of negative sentiment in SATD is an acceptable practice?

Fig. 4 shows that using negative comments in the source code to indicate the priority of an issue is a matter of controversy. While 13% believe this to be an acceptable practice, 16% disagree, and 38% strongly disagree. However, it is interesting to notice that the percentage of the respondents who believe that the usage of negative comments in the source code to indicate priority is an acceptable practice is less than half of the percentage of the respondents exhibiting this behavior (as shown in Fig. 3).

Fig. 5 provides further insights into the developers' annotation practices as well as in the role of negativity. By comparing the left and the central bar charts visually, we can observe that a substantial share of developers write negative source-code comments recording SATD. In particular, 9 respondents indicate that they write negative comments often or very often. This might not appear much but at the same time, only 20 respondents report that they write *any* SATD comments often or very often, i.e., 45% of the respondents that (very) often write SATD comments also write negative SATD comments. This observation concurs with the fact that while, *in general*, 13% of the respondents believe that it would be appropriate to use negativity to express

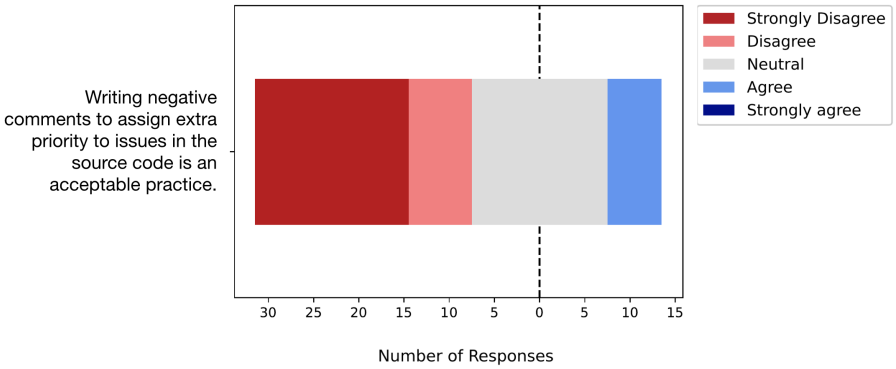

Fig. 4 13% of respondents believe that writing negative comments to indicate higher priority is an acceptable practice (light blue), while 16% disagree with this (pink) and 38% strongly disagree (red).

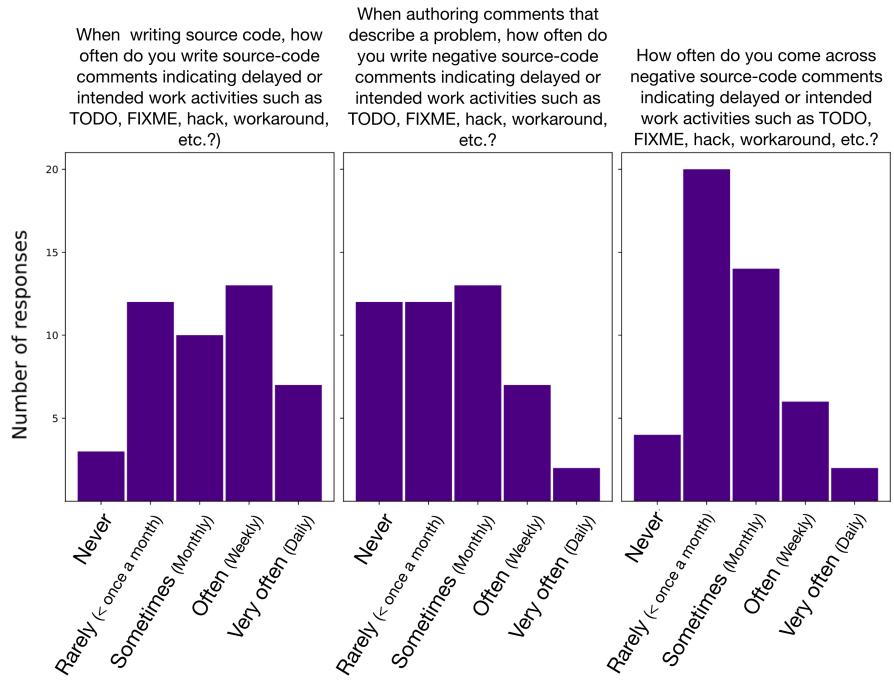

Fig. 5 Responses to closed questions of the survey

a higher priority of an issue (Fig. 4), this percentage increases to 45% if we only consider respondents that (very) often write SATD comments.

The comparison of the bar chart in the middle of Fig. 5 with the one on the right suggests that there are fewer developers that never write negative SATD than those that never encounter negative SATD. However, besides such a difference, these two distributions are very similar.

Table 6 Distribution of sentiment labels over the 994 comments from the da S. Maldonado et al. (2017) dataset.

Category	Negative	(%)	Non-negative	Mixed	Total
Poor implementation choices	125	(29%)	294	7	426
Partially implemented	29	(13%)	197	2	228
Functional issues	66	(49%)	67	2	135
Wait	41	(46%)	45	3	89
Documentation issues	18	(33%)	36	0	54
Testing issues	12	(33%)	24	0	36
Misalignment	6	(29%)	15	0	21
SATD comments outdated	1	(33%)	2	0	3
Deployment issues	1	(50%)	1	0	2
TOTAL	299	(30%)	681	14	994

Table 7 Distribution of sentiment labels over the comments drafted by the respondents for the five vignettes presented in the survey.

Category	Negative	(%)	Non-negative	Mixed	No-comment	Total
Poor implementation choices	2	(7%)	27	0	17	46
Partially implemented	4	(11%)	32	0	10	46
Functional issues	7	(23%)	24	0	15	46
Wait	1	(4%)	22	0	23	46
Documentation issues – A	2	(10%)	18	0	16	36
Documentation issues – B	0	(0%)	3	0	7	10
TOTAL	16	(11%)	126	–	Total comments: 142	

What is highlighted visually is indeed confirmed by the statistical comparison of the distributions. The only statistically significant differences are (i) between developers writing SATD comments and expressing negativity in such comments ($p \simeq 0.014$), and (ii) between developers writing SATD comments and encountering negativity in such comments ($p \simeq 0.021$).

RQ$_3$ Summary: While in general most developers believe that expressing negativity in SATD comments is not acceptable, the opinion shifts towards acceptance among respondents that frequently write SATD comments.

3.5 RQ$_4$: How does the occurrence of negative sentiment vary across different kinds of SATD annotations?

Following the methodology described in Section 2.3, the polarity of 998 SATD comments ($= 1038 - 40$, where 40 comments have been excluded as false positives, i.e., SATD comments that are not real SATD) has been manually classified. Four comments have been further excluded as the authors could not reach an agreement regarding their sentiment polarity. Hence, for this question, we looked at 994 SATD comments (hereinafter, *SATD dataset*) out of 1038 in the original dataset. We report the resulting distribution of sentiment labels in Table 6.

We apply the same protocol and guidelines for labeling the sentiment of the comments drafted by our survey respondents (hereinafter, *survey dataset*). We remind the reader that these comments were formulated by the survey participants in response to the five vignettes representing five different development scenarios where there is a need to admit the presence of a TD in the code. The results of this second labeling study are reported in Table 7.

In the following, we detail the results of the labeling studies performed on both SATD and survey datasets. Overall, we observe that 299 of the 994 comments (30%) in the SATD dataset convey negative sentiment polarity and only 14 items are labeled as mixed. We observe a lower percentage (11%) of negative sentiment in the survey dataset. As we will discuss below, while we report and discuss the results of both studies together, a direct comparison should not be done, given the wider diversity of SATDs in the first data set, and given the different settings of the two studies.

Based on sentiment distribution observed in the two datasets, we found that developers mostly complain about "Functional issues". Specifically, 49% of comments (66 out of 135) in the SATD dataset convey negative sentiment, e.g., "TODO: include the rowids!!!!" in HIBERNATE or "something is very wrong here" in COLUMBA. "Functional issues" is also the most negative category emerging from the comments in the survey dataset, with 23% of proposed comments conveying negative sentiment. Specifically, as reported in Table 7, 7 out of 31 SATD comments drafted for the functional issues' vignette convey a negative sentiment aimed at stressing the presence of an unexpected behavior within the code fragment by using tags such as FIXME or by emphasizing the urgency in addressing a problem (e.g., "Please investigate ASAP, autocompletion appears to be ignoring recently used email addresses.")

Similarly, developers appear annoyed by required changes being on hold: in the SATD dataset, 46% of comments (41 out of 89) belonging to the "Wait" category contains negative sentiment, such as "turn of focus stealing (workaround should be removed in the future!)" in COLUMBA. Similarly to self-directed anger studied by Gachechiladze et al. (2017), we also found cases in which developers blame themselves, e.g., "this is retarded. excuse me while I drool and make stupid noises" in JEDIT.

When looking at the sentiment for on-hold TD vignettes, we found that only 1 out 23 SATD comments contain a negative sentiment. This may depend on both the specific (sub) type of SATD in the vignette which is related to the lack of a proper API (for which often there is little to do), whereas the examples above refer to circumstances internal to the project, which may cause more negativity.

In the SATD dataset, negative sentiment is also found in 33% of "Documentation issues" (e.g., "TODO: are we intentionally eating all events? - tfm 20060203 document!" in ARGOUML) and "Testing issues" (e.g., "TODO enable some proper tests!!" in JMETER). This makes these two categories as the third most negative ones in the SATD dataset, similarly to what was observed in the survey dataset, albeit with different percentages (10% of the survey respondents conveyed negative sentiment in presence of documentation issues for subgroup A, while the three comments drafted for subgroup B were all non-negative).

As for "Poor implementation choices", which is the most frequently observed macro-category in our taxonomy with 426 comments, we observe 29% of negative

sentiment comments in the SATD dataset (e.g., "TODO: terrible implementation!" in HIBERNATE). As for the survey study, only 7% of our survey respondents appear annoyed by issues due to poor implementation choices. However, despite the different proportions, "Poor implementation choices" emerges as the fourth category in terms of percentage of negative sentiment in both datasets[2].

Concerning the "Partially implemented" category, in the SATD dataset, when reporting a partial or non-implemented functionality developers are unlikely to be negative (29 out of 228, corresponding to 13%), e.g., "calculate the adjusted data area taking into account the 3D effect... this assumes that there is a 3D renderer, all this 3D effect is a bit of an ugly hack…" in JFREECHART. For the survey dataset we observe that "Partially implemented" is the category with the second-highest proportion of negative comments (Table 7). In particular, for TDs due to partially/not yet implemented functionality, developers tend to not use a negative sentiment to report them (32 out of 36 comments) while simply stating what is missing in the current implementation (e.g., "Function not completed, Need to raise dialog after invalid input"). In both the survey dataset (Table 7) and the Maldonado et al. dataset (Table 6) developers tend to report what is missed. However, in the survey dataset developers tend to use more negative polarity. We conjecture that this may depend on several factors, ranging from the specific types of TD (again, more diverse in the dataset of da S. Maldonado et al. (2017) than in the vignettes), by the personal attitude of the SATD authors vs. survey respondents, and, last but not least, to the different context (realistic setting vs. artificial one).

As a follow-up study, we performed a pairwise comparison of negative polarity in the macro-categories in the SATD dataset of da S. Maldonado et al. (2017). The results, reported in Table 8, confirm that negative sentiment mostly occurs in presence of bugs or the need to wait to see an issue resolved. Specifically, comments in "Functional issues" and "Wait" appear significantly more negative than comments labeled as "Partially implemented" (Odds Ratio equal to 6.25 and 5.82, respectively) and more than twice as negative than "Poor implementation choice." Moreover, sta-

[2] As for the remaining categories observed in the SATD dataset, they contain a very small number of comments so the results might bring anecdotal evidence and need to be further verified with a larger study.

Table 8 Statistical comparison of negative polarity for the comments in the dataset of Maldonado et al. (OR$>$ 1 means that the proportion is significantly greater for the left-side category. Non-significant pairs are omitted in the table.)

Category 1	**Category 2**	**p-value**	**OR**
Functional issues	Partially implemented	$<$0.01	6.52
Functional issues	Documentation issues	0.04	2.43
Functional issues	Poor implementation choices	$<$0.01	2.30
Poor implementation choices	Partially implemented	$<$0.01	2.85
Wait	Partially implemented	$<$0.01	5.82
Wait	Poor implementation choices	0.02	2.05
Testing issues	Partially implemented	0.02	3.41
Documentation issues	Partially implemented	0.03	2.67

tistical analysis confirms that comments reporting partial implementation are the least negative, compared to the other categories.

RQ_4 summary: SATD about functional issues conveys more negative sentiment. Also, being "on-hold" for various reasons that do not depend on themselves, make developers communicating negative sentiment. Survey respondents were more neutral when reporting partial implementations due to the lack of a proper API, misalignment, or documentation/testing issues.

3.6 RQ_5: To what extent do SATD annotations belonging to different categories contain additional details?

Table 9 Distribution of dimensions used by developers to annotate technical debt over the 1038 comments from the Maldonado et al. dataset.

Category	Component	Name	Bug id	URL	Date
Functional issues	47 (35%)	12 (9%)	11 (8%)	1 (1%)	9 (7%)
Poor implementation choices	152 (35%)	48 (11%)	5 (1%)	0	16 (4%)
Wait	21 (24%)	4 (5%)	9 (10%)	2 (2%)	1 (1%)
Deployment issues	0	0	0	0	0
SATD comments outdated	1 (33%)	0	1 (33%)	0	0
Partially implemented	50 (22%)	22 (10%)	0	0	1 (¡ 1%)
Testing issues	7 (19%)	3 (8%)	0	0	0
Documentation issues	19 (35%)	11 (20%)	0	0	1 (2%)
Misalignment	7 (33%)	4 (19%)	0	0	0
TOT. (UNIQUE)	304 (30%)	104 (10%)	26 (3%)	3 (0.3%)	28 (3%)

Table 10 Distribution of dimensions used by developers to annotate technical debt over Macro-categories for comments drafted in the survey.

Category	Name	Bug id	Date	Total comments drafted
Functional issues	2 (6.25%)	4 (12.50%)	0 (0.00%)	31
Poor implementation choices	1 (3.22%)	5 (16.12%)	0 (0.00%)	29
Wait	0 (0.00%)	3 (12.00%)	1 (4.00%)	23
Partially implemented	0 (0.00%)	4 (11.11%)	0 (0.00%)	36
Documentation issues – A	0 (0.00%)	3 (15.00%)	0 (0.00%)	20
Documentation issues – B	0 (0.00%)	0 (0.00%)	0 (0.00%)	3

We perform a conceptual replication of the work on task annotations by Storey et al. (2008). Following the methodology described in Section 2.4, we leverage the SATD dataset together with the comments left by our respondents to the five vignettes included in the survey. We present the results of this analysis in Table 9 and Table 10. While frequently mentioned by the developers surveyed by Storey et al., additional details rarely appear in our study.

Specifically, 64% of developers from Storey et al. study declared to add references to classes/methods/plug-ins/modules. However, in our study we found the latter happening in 304 SATD comments (30%) which, although not as high as 60%, is a conspicuous fraction of the total. As for the authors' names, instead, only 10% of the SATD comments in our sample contain them, even if around 50% of developers explicitly added their names in the annotations. This may be confirmed considering that only 12 out of 135 SATD comments in the "Functional issues" category clearly report the name. However, about half of the SATD comments referring to a name fall into the "Poor implementation choices" category. One possibility is that during code reviewing processes, reviewers may identify the presence of wrong decisions and highlight them as source code comments. By looking at the comments left from our survey respondents, only 3 out of 148 comments (see Table 10) contain a reference to a developer name. The low percentage in our survey results might be justified because the respondents are invited to draft a comment related to a hypothetical situation.

Moving our attention to the inclusion of bug identifiers, 42% of the SATD comments in the dataset of da S. Maldonado et al. (2017) containing them belong to the "Functional issues" category, however, a non-negligible percentage (33%) concerns the "Wait" category. This is not surprising since developers may introduce a workaround due to a bug that needs to be fixed in the same project or in a third-party library being used. As regards the former, consider the SATD comment: "// TODO : YUCK!!! fix after HHH-1907 is complete" in HIBERNATE, while for the latter in ARGOUML we found a comment stating: "[...] NOTE: This is temporary and will go away [...] `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4714232`" in which the bug is in java.awt library. The same does not apply to the SATD comments from our survey respondents, where for each category we have less than 17% of comments clearly referring to bug identifiers. However, while there are no comments belonging to Documentation and partially implemented functionality issues in the original Maldonado et al. dataset, we found 4 out of 36 and 3 out of 20 for subgroup A and 0 out of 3 for subgroup B comments referring to a bug-id belonging to the same categories in our survey (e.g., "TODO - this is a bug, described in PRG-123, dialog window is not implemented (so what is raised??)").

Finally, looking both at dates and URLs, percentages from the dataset of da S. Maldonado et al. (2017) are very low compared to those reported in the survey by Storey et al. (2008) (3% and 0.3% vs 19%, and 30%). A possible interpretation is that unlikely as stated in the survey, developers assume redundant introducing signature and date (as such information is available in the versioning system anyway). Nevertheless, having them explicitly stated in the source code makes the accountability and tracing more evident. The same occurs also in the drafted comments from our survey where only 1 comment explicitly refer to a date (i.e., "// Blocked on external API by XYZ corp, expecting it to be online by 32 Juvember 2038.") However, as already said for the developer's name, also in this case, respondents are asked to write a comment for a hypothetical situation probably impacting the lack of the additions of further details.

RQ_5 summary: The addition of details such as bug identifiers and names is not so frequent when reporting TD in source code comments. However, developers tend to mention classes and methods more frequently, possibly to improve traceability and support themselves/others in addressing the SATD.

4 Discussion

Sentiment in SATD: a proxy for priority? Recently, software engineering researchers hypothesized that negatively loaded communication might be a proxy for identifying priority of a problem that need to be addressed (Gachechiladze et al., 2017; Uddin and Khomh, 2017; Lin et al., 2019). Similarly, in marketing research, more attention is devoted to negative rather than positive customers' reviews (Wright, 1974; Yin et al., 2010), in line with the assumption that negative feedback is usually more informative as it provides an indication of problems that need to be solved and that might influence consumers' decisions (Casaló et al., 2015; Sparks and Browning, 2011). Our study shows that, while only 13% of the respondents believe that it is acceptable to use negative comments to express priority, more than twice agree to do so, and a similar share of respondents will interpret negative SATD comments as reflecting higher priority. Hence while the perception of negativity as a proxy for priority is not necessarily shared by all developers, it is still sufficiently common to confirm this relation as hypothesized in previous work (Gachechiladze et al., 2017; Uddin and Khomh, 2017; Lin et al., 2019).

Both in the SATD source comments (Table 6) and in the survey (Table 7), negative sentiment is most frequently associated with reporting functional issues. In other words, developers perceive the presence of bugs as more annoying than other problems, such as waiting, partial implementations, testing, and documentation issues. While we acknowledge the need for further investigation of sentiment in SATD, e.g., on a larger dataset, we believe these findings already have actionable implications. Specifically, the amount of negativity observed in the "Functional issues" category suggests that developers should prioritize bug fixing over other issues, such as the implementation of missing functionality. This is also in line with previous findings by Mäntylä et al. (2016) reporting more negativity for bugs and more positive sentiment for feature implementation requests.

Waiting is the category commonly associated with the negative sentiment in the SATD comments but much rarely so in the survey. The high negative sentiment associated with being "on-hold" might be interpreted as an indication of a blocking issue urgently requiring attention. This is in line with previous findings by Ortu et al. (2015) reporting a positive correlation between negative sentiment and issue fixing time. Along the same line, Mäntylä et al. (2016) reported higher emotional activation as the issue resolution time increases, as well as higher arousal in high priority bug reports, thus indicating a presence of emotions with high activation and negative polarity, such as stress. As such, the presence of negative sentiment can be used as a proxy for automatic identification and prioritization of critical, blocking issues,

which might require the interventions of peers. Secondly, the information in the classification can be used to assist in the fine-grained problem of SATD prioritization.

When looking at the SATD polarity, one important element to consider is whether the comment belongs to source code written by the developer who introduced the comment, or whether, instead, the source code has been written to somebody else. In the first case, this means that one is "self-blaming" (e.g., "For some reason, I am not able to get the sheet to size correctly."), warning others that the artifact is not in an ideal state yet, and encouraging others to improve it (e.g., "I have no idea how to get it, someone must fix it" or "If someone knows a better way // please tell me"). This behavior might be related to self-directed anger (Gachechiladze et al., 2017) and may depend a lot on the context in which one works. Zampetti et al. (2021) have indicated that developers are more reluctant to self-admit technical debt in an industrial context than in an open-source one.

In the second case, one may be criticizing source code written by somebody else. Previous work (conducted on a different dataset than ours) has shown that this occurs in a relative minority of cases, with a percentage varying between 0 and 16% (Fucci et al., 2020). Therefore, it is very likely that the majority of SATD are related to their own code, and the negative sentiment mainly expresses un-satisfaction for what was done.

Support for SATD reporting. While, as mentioned above, functional TD is the macro-category triggering more negative comments, survey respondents clearly indicated that issue trackers should be used instead of source code comments to report high-priority SATD. This is because, differently from source code comments, issue trackers allow for better management of the problem (e.g., triaging, priority assignment, discussion, fixing or possibly reopening). Indeed, as a previous study by Xavier et al. (2020) has shown, SATD is also reported beyond source code, e.g., in issue trackers. However, (and this was confirmed by Xavier et al. (2020)), such SATD is infrequently traceable to the exact source-code location that exhibits the SATD. This problem of traceability raises the need for tooling that supports the reporting of SATD, i.e., not only the use of issue trackers but also the need for establishing traces between issues and the affected code fragments (this is not needed for SATD comments present in the source-code as they appear close to the affected code). Such traces between code and issues are for example present when developers use code reviewing tools or pull request discussions, as comments made during a review can point directly to the code.

Supporting developers in effective SATD comment writing: the role of sentiment. Based on the results of the sentiment analysis study, we believe that providing immediate feedback on the negative tone during comment-writing could support developers in more effective collaboration. Specifically, an early detection of harsh or hostile sentiment could not only enable discovering code of conduct violations (Tourani et al., 2017) but also support developers towards effective communication. A SATD sentiment analyzer could prompt developers to highlight the "toxicity" conveyed by their comments, and possibly suggest re-tuning their writing, to avoid irritating their peers. Also, it can recommend using alternative ways of expressing that SATD requires higher priority as suggested in Table 5.

Our vision is corroborated by the survey results: Fig. 4 indicates that nearly half of the survey respondents do not see writing negative comments as an acceptable practice. Furthermore, it is supported by previous findings on collaborative software development and technical knowledge-sharing. Motivated by developers reporting stress due to aggressive communicative behavior in open source communities, Raman et al. (2020) investigated the possibility to automatically detect and mitigate such unhealthy interactions. Steinmacher et al. (2015), instead, showed the impact of social barriers in attracting new contributors to open-source projects. Further studies investigated the impact of sentiment in collective knowledge-building: Calefato et al. (2018b) found a higher probability of fulfilling information-seeking goals on Stack Overflow when questions are formulated using a neutral style, while Choi et al. (2010) found that positive, welcoming tone and constructive criticism is beneficial for online collaboration in Wikipedia.

The evaluation of the SE-specific publicly available sentiment analysis tools we performed (see Section 2) indicated that a fine-tuning is needed before existing tools can be reliably used. Our gold standard for sentiment annotation in SATD represents the first step towards this goal. Furthermore, being able to reliably identify and distinguish hostile comments from non-toxic negative sentiment, as in reporting concerns due to a bug, is a crucial aspect to take into account in performing such fine-tuning to avoid marking non-toxic comments for moderation. By releasing our gold standard and guidelines for annotation, we hope to stimulate further research on negativity detection in SATD.

References perceived as important in comments (Storey et al., 2008), but not widely used in SATD comments. Survey respondents state that including bug IDs and name of the responsible person can be used to indicate that the SATD should have a higher priority (Table 5). Based on the results of our study we envision the emergence of tools supporting and guiding the authors towards adding proper references and information while adding SATD.

While previous work by Storey et al. (2008) stressed the perceived importance of various forms of references in task annotations, they occur much less frequently than one would expect. For example, while most developers (64%) participating in the study by Storey et al. declared they add references to classes, methods, plug-ins, and modules, such references appear only in 30% of our dataset of SATD comments; similarly, adding bug ids has been reported by 44% of the developers surveyed by Storey et al. only 3% of the SATD comments in our dataset contained bug ids. In our survey we have asked the respondents to provide examples of SATD comments that they *would* write given a situation (see Table 10): without further prompting 11.11–16.12% of the survey respondents have included bug ids in their comments across all categories of SATD, names of developers or date (as discussed by Storey et al. (2008)) are much less commonly mentioned. Hence, for all categories of our taxonomy to properly document SATD, developers should open bug reports in the issue tracker and reference them in the comment, as well as refer to other classes or methods to be updated.

Tool support could be developed to automatically detect introduction/change of SATD comments, and generate a date and signature for it, since half of developers in the study by Storey et al. include both their names and dates during task anno-

tations. Similarly, automated support could be provided to reference/open an issue every time a Functional SATD is detected. Also, when "on-hold" SATD comment is automatically detected (Maipradit et al., 2020b), developers may be guided to add a reference to a proper source. By helping to achieve properly structured SATD comments (depending on their type) with suitable references, not only those comments may become more traceable and understandable, but the available information will also help to better drive their manual (or semi-automated) resolution.

5 Related Work

In the following, we discuss relevant literature related to (i) studies about TD and SATD, and (ii) sentiment analysis in software development.

5.1 Technical Debt and Self-Admitted Technical Debt

In the past years, the research community empirically studied TD and SATD. Seaman and Guo (2011), Kruchten et al. (2013), Brown et al. (2010), and Alves et al. (2014) made different considerations about "technical debt" highlighting that TDs are a communication media among developers and managers to discuss and address development issues. Furthermore, Lim et al. (2012) highlighted that TD introduction is mostly intentional, and Ernst et al. (2015) pointed out how TD awareness is a cornerstone for TD management. Zazworka et al. (2011), instead, highlighted the need for proper handling and identification of TD to reduce their negative impact on software quality.

By looking at source code comments in open source projects Potdar and Shihab (2014) found that developers tend to "self-admit" TD. In a follow-up study, da S. Maldonado and Shihab (2015) developed an approach that by using 62 patterns identifies whether or not a comment is an SATD along with such categories as defect, design, documentation, requirement, and test debts. Bavota and Russo (2016), instead, have refined the above classification providing a taxonomy featuring 6 higher-level TD categories properly specialized into 11 sub-categories. Our work differs from that by Potdar and Shihab (2014) and Bavota and Russo (2016) in that we focus on the content reported in the SATD without considering the development life-cycle in which the SATD may be mapped.

Nevertheless, it is possible to identify a possible correspondence between the SATD categories identified by Bavota and Russo and those we have identified. Table 11 reports the mapping between the third-level SATD classification by Bavota and Russo (2016) and our taxonomy. By looking at the mapping, it is possible to state that, except for "Licensing", which was not encountered in our study, our taxonomy covers all the categories by Bavota and Russo (2016). Note that in some cases we could only create a mapping with their 2nd-level category, as in the case of "Documentation Issues/Inconsistent Documentation" mapped on their "Inconsistent comments", and "Functional Issues", mapped on their "Functional".

Being based on the technical content of commit messages rather than on the development process, our taxonomy provides a more detailed classification for some

Table 11 Mapping between Bavota and Russo (2016) SATD categories and our taxonomy. In some cases we could only create a mapping with the 2nd-level category of Bavota and Russo (2016), as in the case of "Documentation Issues/Inconsistent Documentation" mapped on their "Inconsistent comments", and "Functional Issues", mapped on their "Functional".

Bavota and Russo (2016)			**Our Taxonomy**	
1st Level	**2nd Level**	**3rd Level**	**Category**	**Sub-Category**
Code	Low Internal Quality		Poor Impl. Choices	Poor impl. solutions Poor API usage Code review needed Maintainability issues Performance issues Usability Won't improve the code
			Partially/Not Impl. Func.	
	Workaround		Wait	Temporary patch
Design	Code Smells		Poor Impl. Choices	Maint. Issues Poor Impl. Solutions
	Design Patterns		Poor Impl. Choices	Maintainability Issues Poor Impl. Sol.
Doc.	Incons. Comm.		Doc. Issues	Inconsistent Doc.
		Addressed TD	SATD outdated	
		Won't fix	Func. Issues	Fix to postpone
			Poor Impl. Choices	Won't improve the code
			Doc. Issues	Won't modify doc.
	Licensing			
Defect	Defects	Known defects to fix	Func. Issues	Bug to fix
		Partially fixed defects	Func. Issues	Temporary Patch
			Partially/Not Impl. Func.	Work under specific cond.
	Low Ext. Qual.		Poor Impl. Choices	Usability
Test			Testing Issues	Improve tests Test case bugs Disalign. prod/test code
Req.	Functional		Func. Issues	
		Improv. to feat. needed	Partially/Not Impl. Func.	Work under specific cond. Func. issue elsewhere Pre-cond. missing Post-cond. unchecked Incompl. except. handling
		New feat. to be impl.	Partially/Not Impl. Func.	Work under specific cond. Func. issue elsewhere Pre-cond. missing Post-cond. unchecked Incompl. except. handling
	Non Functional	Performances	Poor Impl. Choices	Performance issues

of the general categories in Bavota and Russo, e.g., "Low Internal Quality" is specialized in our taxonomy among different type of issues in the "Poor Implementation Choices" category. Finally, our taxonomy enriches the one already presented in previous literature since that 14 out of our 33 categories and/or sub-categories cannot be mapped on the taxonomy by Bavota and Russo (2016), unless doing a generic mapping on the first level of their taxonomy.

Concerning the SATD classification, Maipradit et al. (2020b), introduced the concept of "on-hold" SATD i.e., comments expressing a condition indicating that a de-

veloper is waiting for an event internal or external to the project under development. As a follow-up study, Maipradit et al. (2020a), built a classifier aimed at detecting on-hold SATD with an average AUC of 0.97. Moreover, they studied the on-hold SATD evolution by looking into the life-span of removed issue-referring comments finding that 13% of on-hold SATD are removed from the code more than one year after their resolution.

Fucci et al. (2020), conjectured that "self-admission" may not necessarily mean that the comment has been introduced by whoever has written or changed the source code. Their results highlight that SATD comments are mainly introduced by developers having a high level of ownership on the SATD-affected source code.

While most of the aforementioned work focused the attention on SATD in source code comments or commit messages, Xavier et al. (2020) studied SATD being reported in the issue trackers of five projects. Their findings indicate that SATD issues take longer to be fixed than other issues and that only 29% of those issues can be traced onto source code comments. As confirmed by the results of our survey, where respondents have indicated that SATD should be reported in issue trackers and not in the source code, we share with Xavier et al. (2020) the need to develop tools for better SATD management.

As regards the impact of SATD, Wehaibi et al. (2016) found that SATD leads to complex changes in the future, 1, additionally, Kamei et al. (2016) highlighted that $\simeq 42\%$ of TD incurs positive interest. From a different perspective, Zampetti et al. (2017) developed an approach for recommending when a design TD has to be admitted.

Differently from previous work, we focused our attention on the SATD content, i.e., what developers usually annotate about TD, as well as how they communicate the presence of this temporary solution, i.e., sentiment and external references.

Zampetti et al. (2021) conducted a survey with open-source and industry developers to investigate their TD admission practices. Their study found that TD admission is very similar between industry and open-source, although then behavior of industrial developers upon commenting source code is often constrained by organizational guidelines. Also, industrial developers are more afraid in admitting TD, because they see this as a way to reveal their weaknesses, and are afraid this may have consequence on their career.

The research community has also focused on SATD removal. da S. Maldonado et al. (2017) found that there is a high percentage of SATD being removed even if their survivability varies by project. Zampetti et al. (2018), instead, studied the relationship between comment removals and changes applied to the affected source code. They found how SATD can be either removed through focused changes (e.g., to conditional statements), but also by rewriting/replacing substantial portions of source code. Liu et al. (2021) also empirically analyzed the introduction and removal of different types of TD, in this case with a specific focus to machine learning projects. They found that the most frequently introduced TD during the development process is design debt, whereas in terms of removal developers tend to remove requirement debt the most, and design debt fastest. To aid developers in SATD removal, Zampetti et al. (2020) proposed SARDELE, a multi-level classifier able to recommend six SATD removal strategies using a deep learning approach. We believe that a more focused analysis of

the SATD content like the one done in our work could help to refine such approaches, allowing for more actionable suggestions.

The textual content of SATD comments is analyzed by Rantala et al. (2020), who developed a detector for Keyword-Labeled SATD, i.e., SATD highlighted by specific keywords such as TODO or FIXME. Their analysis shows, among others, the usages of keywords expressing not only the need for code changes, but also a situation of uncertainty. Our analysis complements the findings of Rantala et al. (2020) as it turns out that, in some circumstances, SATD also contains expressions of negativity.

TODO comments can be sometimes obsolete, but they may or may not be removed by developers. Therefore Gao et al. (2021) proposed an approach, named TD-Cleaner, to identify and remove obsolete TODO comments. Their approach is based on a neural encoder that learns from SATD comments, code changes, and commit messages. In principle, obsolete SATD could affect all categories we have considered (in RQ_1), although our manual analysis did not identify any explicit trace of such comments.

By mining the file history of these frameworks, we find that design debt is introduced the most along the development process. As for the removal of technical debt, we find that requirement debt is removed the most, and design debt is removed the fastest. Most of test debt, design debt, and requirement debt is removed by the developers who introduced them.

Zampetti et al. (2021) surveyed developers in the open-source and industry, investigating whether they admit SATD differently. They found that, in general, their behavior is similar. At the same time, industrial developers are more driven by their organizational guidelines, and are also (implicitly or explicitly) discouraged to admit SATD and/or to push code that is not ready. The finding of our survey further confirms what conjectured by Zampetti et al. (2021), because developers have pointed out that code with SATD should not be merged.

5.2 Sentiment Analysis in Software Development

Recently, a trend has emerged and consolidated to leverage sentiment analysis in empirical software engineering research (Novielli and Serebrenik, 2019; Lin et al., 2021). Murgia et al. (2014) presented an early exploratory study of emotions in software artifacts. By manually labeling issues from the Apache Software Foundation, they found that developers feel and report a variety of emotions, including gratitude, joy, and sadness. Ortu et al. (2015), instead, investigated the correlation between sentiment in issues and their fixing time showing how issues with negative polarity, e.g., sadness, have a longer fixing time. On the same line, Mäntylä et al. (2016) performed a correlation study between emotions and bug priority to derive symptoms of productivity loss and burnout. By looking at issue tracking comments they mined emotions and used them to compute Valence (i.e., sentiment polarity), Arousal (i.e., sentiment intensity), and Dominance (the sensation of being in control of a situation). Their findings highlight that bug reports are associated with a more negative Valence, and issue priority positively correlates with the emotional activation, with higher priority correlating with higher arousal. While not representing any causal relationship

between emotions and the investigated factors, both correlation studies suggest how sentiment can be used as a proxy for problems or priority in the development process, for monitoring the mood of software development teams, as well as identifying factors correlated to positive emotion, towards fostering effective collaboration and developers' productivity. Differently from the previous correlation studies, our study investigates how developers communicate the presence of technical debt by manually labeling the sentiment inside SATD comments and survey responses imitating SATD comments. Furthermore, while previous studies *conjectured* the link between sentiment and priority in the software development process, we have evaluated this conjecture by surveying software developers.

Researchers in requirements engineering use sentiment analysis as a source of information for requirements classification towards supporting software maintenance and evolution. Panichella et al. (2015) applied sentiment analysis for classifying user reviews in Google Play and Apple Store, Maalej et al. (2016) leveraged several text-based features, including sentiment, for automatically classifying app reviews into four categories, namely bug reports, feature requests, user experiences, and text ratings, while Portugal and do Prado Leite (2018) use sentiment analysis to acquire a deeper understanding of usability requirements.

While early studies of sentiment in software development made use of general-purpose sentiment analysis tools this approach is shown to be unreliable (Jongeling et al., 2017). To address this challenge multiple sentiment analysis tools have been specially designed for the software development domain (Islam and Zibran, 2018; Calefato et al., 2018a; Ahmed et al., 2017; Alkalbani et al., 2016; Chen et al., 2019; Ding et al., 2018). We have evaluated the applicability of such tools to SATD comments but as explained in Section 2.3.1 the tools missed some negative comments due to the presence of lexicon which is specific to SATD comments. Hence, the sentiment analysis in this paper has been performed manually.

As far as negative emotions are concerned, Gachechiladze et al. (2017) looked at the anger and its direction in collaborative software development, envisioning the tools detecting the anger target in developers' communication, by distinguishing between anger towards *self*, *others*, and *object*. In their vision, detecting anger towards self could be useful to support stuck developers, while anger towards others should be detected for community moderation purposes. Finally, detecting anger towards objects can enable the recommendation of alternative tools or task prioritization. As a preliminary step towards this goal, they created a manually annotated dataset of 723 sentences from the Apache issue reports and used it to train a supervised classifier for anger detection. Similarly to this study, we focus on negative emotion confirming that their detection and modeling can serve as a proxy for problems occurring in the software development process.

A complementary line of research considers biometric measurements to assess software developers' emotional states rather than texts authored by them (Müller and Fritz, 2015; Girardi et al., 2020, 2021).

6 Threats to Validity

Threats to *construct validity* concern the relationship between theory and observation. One threat is how the comments are classified in $\mathbf{RQ}_1$. Our knowledge of the analyzed systems may not be as deep as those of the original developers. To mitigate this threat, we analyzed not only the comments but also the corresponding source code when this was needed. A relevant threat for $\mathbf{RQ}_4$ is related to how "sentiment" is perceived by annotators but may not match the actual sentiment of developers. For what possible, the subjectiveness in $\mathbf{RQ}_1$ and $\mathbf{RQ}_4$ has been mitigated by establishing clear coding guidelines, and by doing initial joint sessions. Furthermore, we resolved all disagreements through a discussion during plenary meetings involving all the annotators. For sentiment labeling, we also measured the extent to which we could have reached an agreement by chance using inter-rater agreement metrics.

Concerning the first part of the survey which we used to answer $\mathbf{RQ}_2$ and $\mathbf{RQ}_3$, we ask developers to provide their perception about SATD practices and the extent to which negative polarity should be used when reporting SATD. We are aware that this kind of "self-assessment" conducted through a survey not only can be affected by the self-selection of the participants (e.g., less negative ones were those who decided to respond), but, also, that what answered to a questionnaire may be different from what one actually does in the practice.

In the survey study, we used vignettes (Rossi and Nock, 1983; McNamara et al., 2018; Palomba et al., 2021) to gather, from respondents, their reaction to certain development scenarios or to certain situations occurring in a project. Although the vignettes are inspired by the SATD source comments we have analyzed and realistic development scenarios, they might still be artificial with respect to the intrinsic complexity and the constraints of open-source software development. Moreover, although we have paid special attention to neutral wording in the vignettes, we cannot exclude that the vignettes' text influenced the sentiment of the SATD comment written by respondents. Finally, to avoid having an excessively long study (and therefore discouraging participation), we had to limit the number of vignettes to five. This makes their diversity, depth and breadth with respect to the other analysis we did on the Maldonado et al. dataset fairly limited. For this reason, we cannot directly compare the results of the two studies used to answer $\mathbf{RQ}_4$.

Threats to *internal validity* are related to factors internal to our study that can affect our results. Although we created a relatively large and statistically significant sample, we cannot exclude that our sampling strategy is weakly representative of the studied dataset. In particular, we sampled our dataset starting from the data and categories of Maldonado et al., so we might have inherited representativeness threats from the original study. Measurement imprecision in $\mathbf{RQ}_5$ has been mitigated, where it matters, through manual analysis.

A further threat affecting $\mathbf{RQ}_3$ and $\mathbf{RQ}_4$ may be represented by the sentiment of SATD comments submitted by the survey participants for our vignettes. While we asked them to behave as they were working on their own project, their actual sentiment may be different from a real development context (e.g., when a developer finds that somebody has introduced some poor source code) to an artificial setting, where one may tend to be more polite. At the same time, the artificial context of the

survey might release some of the pressure induced on the developers by the need to conform to the norms of the professional behavior at the workplace.

In this work we find that 29 of the comments from the dataset of Maldonado et al. contain references to external bug reports or urls. However, these references might not be up to date anymore as Li and Zhong (2021) have found that some bug reports become obsolete over time.

To ensure that we only study the practices of open-source software projects, we ask participants whether they have contributed to open-source software projects in the past three months. However, this does not exclude the possibility that we received responses from participants who mostly contribute to commercial software projects, and only sparingly contribute to open-source in the past three months. To minimize the risk of these participants answering based on their commercial experience, we explicitly included the text *'you are working on an open-source application'* in each question of the survey.

Finally, the order in which we presented vignettes may have impacted the comments written by respondents. We mitigated this threat by using versions of the survey with a different ordering, and by using PERMANOVA, (Anderson, 2017) to analyze the ordering effect and discuss how ordering could have influenced the results (see Section 3.1).

Threats to *external validity* concern the generalizability of our findings. The qualitative nature of the study (especially $\mathbf{RQ}_1$) and the need for manual inspection for all three research questions do not make a large-scale analysis feasible. Therefore, although the sample is statistically significant, it may not generalize to further projects and programming languages different from Java. For $\mathbf{RQ}_1$, although we reached saturation when identifying categories, we cannot exclude that new categories would emerge when looking at further datasets. Both components of the study—the SATD comment mining part and the survey—focus on a (relatively limited) set of open-source projects, therefore results might not generalize further. That being said, previous work of Zampetti et al. (2021) showed that the differences in SATD practices between industry and open source are fairly limited.

Finally, in our survey study, we recruited participants by advertising the questionnaires through messages to mailing lists, posts on social media, and personal contacts. On one hand, this is in line with our assumption that the different communication channels we used to recruit the participants do not influence the population. On the other hand, this allows us to reach a broader and more diverse audience. However, we are aware this might have potentially introduced threats due to mixed recruiting strategy.

7 Conclusion

In this paper, we have studied developers' practices related to Self-Admitted Technical Debt (SATD) in open-source software projects. More specifically, we investigated (i) the content of SATD comments, (ii) the methods used to indicate priority in SATD, (iii) the extent to which developers believe that the expression of negative sentiment in SATD is acceptable, (iv) how negative polarity occurs in different kinds of SATD,

and (v) whether developers add details such as URLs, contributors' names, timestamps, or bug IDs in SATD comments. The study has combined the manual classification of 1038 SATD comments from a curated dataset of da S. Maldonado et al. (2017), with a survey involving 46 open-source developers, which comprised open-ended and closed-ended questions about SATD annotation practices, as well as tasks requiring to write SATD comments for vignettes (Rossi and Nock, 1983) depicting scenarios where TD could be admitted.

We found that SATD is spread across different categories, and that different problems are described in SATD. SATD comments are often related to functional issues and partially-implemented functionality, but also to poor implementation choices, and waiting for other features to be ready/APIs to be available. Less frequent, though non-negligible, are SATD comments related to documentation and tests. A group of developers (13 out of 44) acknowledges the use of negativity in the source code to indicate extra-priority, tentatively confirming what conjectured in previous literature (Gachechiladze et al., 2017; Uddin and Khomh, 2017; Lin et al., 2019). At the same time, survey respondents indicated that, when discussing SATD, a negative polarity should be avoided, despite our analysis of the da S. Maldonado et al. (2017) dataset found a relatively large ($> 40\%$) proportion SATD with negative polarity, especially related to functional issues and "on-hold" debt.

Finally, although we found the presence of various pieces of additional information in SATD comments (including bug IDs), survey respondents argued that SATD comments in the source code should not be used to trigger development activities or to highlight problems; issue trackers should be used instead. However, the use of issue trackers does not solve the problem of ensuring traceability between issues and source code elements being affected by SATD.

All the above findings foster future research on SATD, primarily aimed at helping developers in better writing SATD, also considering that previous research already found ways to recommend when SATD should be admitted (Zampetti et al., 2017). Primarily, tools should help developers to properly write SATD comments, by using a suitable polarity, but at the same by including proper pieces of information such as links to external resources, or authorship information. More importantly, better support than just using issue trackers is highly desirable, especially to establish traceability between TD-affected code and issues.

Declarations

Conflict of interest

The authors have no conflicts of interest to declare that are relevant to the content of this article.

Acknowledgements We'd like to thank Gianmarco Fucci for his contribution to the companion paper (Fucci et al., 2021). Moreover, we are also grateful to the survey respondents who took the time to respond to our survey, and the authors of da S. Maldonado et al. (2017) for making available their dataset.

References

Ahmed T, Bosu A, Iqbal A, Rahimi S (2017) SentiCR: A customized sentiment analysis tool for code review interactions. ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering pp 106–111, DOI 10.1109/ASE.2017.8115623

Alkalbani A, Ghamry A, Hussain F, Hussain O (2016) Sentiment analysis and classification for software as a service reviews. In: 2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA), IEEE Computer Society, Los Alamitos, CA, USA, pp 53–58, DOI 10.1109/AINA.2016.148, URL https://doi.ieeecomputersociety.org/10.1109/AINA.2016.148

Alves NSR, Ribeiro LF, Caires V, Mendes TS, Spínola RO (2014) Sixth international workshop on managing technical debt, mtd@icsme 2014, victoria, bc, canada, september 30, 2014. In: International Workshop on Managing Technical Debt, IEEE Computer Society, pp 1–7

Anderson MJ (2017) Permutational Multivariate Analysis of Variance (PERMANOVA), American Cancer Society, pp 1–15. DOI https://doi.org/10.1002/9781118445112.stat07841, URL https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118445112.stat07841, https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118445112.stat07841

Bavota G, Russo B (2016) A large-scale empirical study on self-admitted technical debt. In: Kim M, Robbes R, Bird C (eds) International Conference on Mining Software Repositories, ACM, pp 315–326

Benjamini Y, Hochberg Y (1995) Controlling the false discovery rate: A practical and powerful approach to multiple testing. Journal of the Royal Statistical Society Series B (Methodological) 57(1):289–300

Brown N, Cai Y, Guo Y, Kazman R, Kim M, Kruchten P, Lim E, MacCormack A, Nord RL, Ozkaya I, Sangwan RS, Seaman CB, Sullivan KJ, Zazworka N (2010) Managing technical debt in software-reliant systems. In: Roman G, Sullivan KJ (eds) Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010, ACM, pp 47–52

Calefato F, Lanubile F, Maiorano F, Novielli N (2018a) Sentiment Polarity Detection for Software Development. Empirical Software Engineering 23(3):1352–1382, DOI 10.1007/s10664-017-9546-9

Calefato F, Lanubile F, Novielli N (2018b) How to ask for technical help? evidence-based guidelines for writing questions on stack overflow. Inf Softw Technol 94(C):186–207

Casaló LV, Flavián C, Guinaliu M, Ekinci Y (2015) Avoiding the dark side of positive online consumer reviews: Enhancing reviews' usefulness for high risk-averse travelers. Journal of Business Research 68:1829–1835

Chen Z, Cao Y, Lu X, Mei Q, Liu X (2019) Sentimoji: an emoji-powered learning approach for sentiment analysis in software engineering. In: Dumas M, Pfahl D, Apel S, Russo A (eds) Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software

Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019, ACM, pp 841–852, DOI 10.1145/3338906.3338977, URL https://doi.org/10.1145/3338906.3338977

Choi B, Alexander K, Kraut RE, Levine JM (2010) Socialization tactics in wikipedia and their effects. In: Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work, Association for Computing Machinery, New York, NY, USA, CSCW '10, p 107–116, DOI 10.1145/1718918.1718940, URL https://doi.org/10.1145/1718918.1718940

Diefendorff J, Richard E (2003) Antecedents and consequences of emotional display rule perceptions. The Journal of applied psychology 88:284–94, DOI 10.1037/0021-9010.88.2.284

Ding J, Sun H, Wang X, Liu X (2018) Entity-level sentiment analysis of issue comments. In: Begel A, Serebrenik A, Graziotin D (eds) Proceedings of the 3rd International Workshop on Emotion Awareness in Software Engineering, SEmotion@ICSE 2018, Gothenburg, Sweden, June 2, 2018, ACM, pp 7–13, DOI 10.1145/3194932.3194935, URL https://doi.org/10.1145/3194932.3194935

Ebert F, Castor F, Novielli N, Serebrenik A (2018) Communicative intention in code review questions. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 519–523

Ernst NA, Bellomo S, Ozkaya I, Nord RL, Gorton I (2015) Measure it? manage it? ignore it? software practitioners and technical debt. In: Foundations of Software Engineering, ACM, pp 50–60

Fischer M, Pinzger M, Gall H (2003) Populating a release history database from version control and bug tracking systems. In: Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on, IEEE

Fluri B, Wursch M, Gall HC (2007) Do code and comments co-evolve? on the relation between source code and comment changes. In: 14th Working Conference on Reverse Engineering (WCRE 2007), IEEE, pp 70–79

Fucci G, Zampetti F, Serebrenik A, Di Penta M (2020) Who (self) admits technical debt? In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 672–676

Fucci G, Cassee N, Zampetti F, Novielli N, Serebrenik A, Penta MD (2021) Waiting around or job half-done? sentiment in self-admitted technical debt. In: 2021 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR) (MSR), IEEE Computer Society, Los Alamitos, CA, USA, pp 403–414, DOI 10.1109/MSR52588.2021.00052, URL https://doi.ieeecomputersociety.org/10.1109/MSR52588.2021.00052

Gachechiladze D, Lanubile F, Novielli N, Serebrenik A (2017) Anger and its direction in collaborative software development. In: Proceedings of the 39th International Conference on Software Engineering: New Ideas and Emerging Results Track, IEEE Press, ICSE-NIER '17, p 11–14, DOI 10.1109/ICSE-NIER.2017.18

Gao Z, Xia X, Lo D, Grundy JC, Zimmermann T (2021) Automating the removal of obsolete TODO comments. In: ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021, pp 218–229, DOI 10.1145/3468264.3468553, URL https://doi.org/10.1145/3468264.3468553

Girardi D, Novielli N, Fucci D, Lanubile F (2020) Recognizing developers' emotions while programming. In: Rothermel G, Bae D (eds) International Conference on Software Engineering, ACM, pp 666–677

Girardi D, Lanubile F, Novielli N, Serebrenik A (2021) Emotions and perceived productivity of software developers at the workplace. IEEE Transactions on Software Engineering xxx(1):1–1, DOI 10.1109/TSE.2021.3087906

Hochschild R (1983) The managed heart: Commercialization of human feeling. The University of California Press

Islam MR, Zibran MF (2018) Sentistrength-se: Exploiting domain specificity for improved sentiment analysis in software engineering text. Journal of Systems and Software 145:125 – 146, DOI https://doi.org/10.1016/j.jss.2018.08.030, URL http://www.sciencedirect.com/science/article/pii/S0164121218301675

Jongeling R, Sarkar P, Datta S, Serebrenik A (2017) On negative results when using sentiment analysis tools for software engineering research. Empir Softw Eng 22(5):2543–2584, DOI 10.1007/s10664-016-9493-x, URL https://doi.org/10.1007/s10664-016-9493-x

Kamei Y, Maldonado EdS, Shihab E, Ubayashi N (2016) Using analytics to quantify interest of self-admitted technical debt. In: Lichter H, Fögen K, Sunetnanta T, Anwar T, Yamashita A, Moonen L, Mens T, Tahir A, Sureka A (eds) Joint Proceedings of the 4th International Workshop on Quantitative Approaches to Software Quality (QuASoQ 2016) and 1st International Workshop on Technical Debt Analytics (TDA 2016) co-located with the 23rd Asia-Pacific Software Engineering Conference (APSEC 2016), Hamilton, New Zealand, December 6, 2016, CEUR-WS.org, CEUR Workshop Proceedings, vol 1771, pp 68–71

Konietschke F, Hothorn LA, Brunner E (2012) Rank-based multiple test procedures and simultaneous confidence intervals. Electronic Journal of Statistics 6:738–759

Krippendorff K (2012) Content analysis: An introduction to its methodology. Sage

Kruchten P, Nord RL, Ozkaya I, Falessi D (2013) Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt. ACM SIGSOFT Software Engineering Notes

Kruskal WH, Wallis WA (1952) Use of ranks in one-criterion variance analysis. Journal of the American Statistical Association 47(260):583–621, DOI 10.1080/01621459.1952.10483441

Li Z, Zhong H (2021) An empirical study on obsolete issue reports. In: Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering, p page to appear

Lim E, Taksande N, Seaman C (2012) A balancing act: what software practitioners have to say about technical debt. IEEE Software 29(6):22–27

Lin B, Zampetti F, Bavota G, Di Penta M, Lanza M, Oliveto R (2018) Sentiment analysis for software engineering: how far can we go? In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, pp 94–104, DOI 10.1145/3180155.3180195, URL https://doi.org/10.1145/3180155.3180195

Lin B, Zampetti F, Bavota G, Di Penta M, Lanza M (2019) Pattern-based mining of opinions in Q & A websites. In: 2019 IEEE/ACM 41st International Conference

on Software Engineering (ICSE), pp 548–559, DOI 10.1109/ICSE.2019.00066

Lin B, Cassee N, Serebrenik A, Bavota G, Novielli N, Lanza M (2021) Opinion mining for software development: A systematic literature review. ACM Transactions on Software Engineering and Methodology xx:xx–xx

Liu J, Huang Q, Xia X, Shihab E, Lo D, Li S (2021) An exploratory study on the introduction and removal of different types of technical debt in deep learning frameworks. Empir Softw Eng 26(2):16, DOI 10.1007/s10664-020-09917-5, URL https://doi.org/10.1007/s10664-020-09917-5

Maalej W, Kurtanovic Z, Nabil H, Stanik C (2016) On the automatic classification of app reviews. Requirements Engineering 21:311–331

Maipradit R, Lin B, Nagy C, Bavota G, Lanza M, Hata H, Matsumoto K (2020a) Automated identification of on-hold self-admitted technical debt. In: 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, pp 54–64

Maipradit R, Treude C, Hata H, Matsumoto K (2020b) Wait for it: identifying "on-hold" self-admitted technical debt. Empirical Software Engineering 25(5):3770–3798

Mäntylä M, Adams B, Destefanis G, Graziotin D, Ortu M (2016) Mining valence, arousal, and dominance: Possibilities for detecting burnout and productivity? In: Proceedings of the 13th International Conference on Mining Software Repositories, Association for Computing Machinery, New York, NY, USA, MSR '16, p 247–258, DOI 10.1145/2901739.2901752

McNamara A, Smith J, Murphy-Hill E (2018) Does acm's code of ethics change ethical decision making in software development? In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2018, p 729–733, DOI 10.1145/3236024.3264833, URL https://doi.org/10.1145/3236024.3264833

Müller SC, Fritz T (2015) Stuck and frustrated or in flow and happy: Sensing developers' emotions and progress. In: Bertolino A, Canfora G, Elbaum SG (eds) 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1, IEEE Computer Society, pp 688–699, DOI 10.1109/ICSE.2015.334, URL https://doi.org/10.1109/ICSE.2015.334

Murgia A, Tourani P, Adams B, Ortu M (2014) Do developers feel emotions? an exploratory analysis of emotions in software artifacts. In: Proceedings of the 11th Working Conference on Mining Software Repositories, Association for Computing Machinery, New York, NY, USA, MSR 2014, p 262–271, DOI 10.1145/2597073.2597086, URL https://doi.org/10.1145/2597073.2597086

Newcombe RG (1998) Interval estimation for the difference between independent proportions: comparison of eleven methods. Statistics in Medicine 17(8):873–890, DOI https://doi.org/10.1002/(SICI)1097-0258(19980430)17:8⟨873::AID-SIM779⟩3.0.CO;2-I

Novielli N, Serebrenik A (2019) Sentiment and emotion in software engineering. IEEE Softw 36(5):6–9, DOI 10.1109/MS.2019.2924013, URL https://doi.org/10.1109/MS.2019.2924013

Novielli N, Girardi D, Lanubile F (2018) A benchmark study on sentiment analysis for software engineering research. In: Proceedings of the 15th International Conference on Mining Software Repositories, Association for Computing Machinery, New York, NY, USA, MSR '18, p 364–375, DOI 10.1145/3196398.3196403, URL https://doi.org/10.1145/3196398.3196403

Novielli N, Calefato F, Dongiovanni D, Girardi D, Lanubile F (2020) Can We Use SE-specific Sentiment Analysis Tools in a Cross-Platform Setting? Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories, MSR 2020 pp 158–168, DOI 10.1145/3379597.3387446, 2004.00300

Novielli N, Calefato F, Lanubile F, Serebrenik A (2021) Assessment of off-the-shelf SE-specific sentiment analysis tools: An extended replication study. Empir Softw Eng 26

Ortu M, Adams B, Destefanis G, Tourani P, Marchesi M, Tonelli R (2015) Are bullies more productive? empirical study of affectiveness vs. issue fixing time. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, pp 303–313, DOI 10.1109/MSR.2015.35

Palomba F, Andrew Tamburri D, Arcelli Fontana F, Oliveto R, Zaidman A, Serebrenik A (2021) Beyond technical aspects: How do community smells influence the intensity of code smells? IEEE Transactions on Software Engineering 47(1):108–129, DOI 10.1109/TSE.2018.2883603

Panichella S, Di Sorbo A, Guzman E, Visaggio CA, Canfora G, Gall HC (2015) How can i improve my app? classifying user reviews for software maintenance and evolution. In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 281–290, DOI 10.1109/ICSM.2015.7332474

Portugal RLQ, do Prado Leite JCS (2018) Usability related qualities through sentiment analysis. In: Fucci D, Novielli N, Guzman E (eds) 1st International Workshop on Affective Computing for Requirements Engineering, AffectRE@RE 2018, Banff, AB, Canada, August 21, 2018, IEEE, pp 20–26, DOI 10.1109/AffectRE.2018.00010, URL https://doi.org/10.1109/AffectRE.2018.00010

Potdar A, Shihab E (2014) An exploratory study on self-admitted technical debt. In: 30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014, pp 91–100

Raman N, Cao M, Tsvetkov Y, Kästner C, Vasilescu B (2020) Stress and burnout in open source: Toward finding, understanding, and mitigating unhealthy interactions. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results, Association for Computing Machinery, New York, NY, USA, ICSE-NIER '20, p 57–60, DOI 10.1145/3377816.3381732, URL https://doi.org/10.1145/3377816.3381732

Rantala L, Mäntylä M, Lo D (2020) Prevalence, contents and automatic detection of KL-SATD. In: 46th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2020, Portoroz, Slovenia, August 26-28, 2020, pp 385–388, DOI 10.1109/SEAA51224.2020.00069, URL https://doi.org/10.1109/SEAA51224.2020.00069

Ren X, Xing Z, Xia X, Lo D, Wang X, Grundy J (2019) Neural network-based detection of self-admitted technical debt: From performance to explainability. ACM Trans Softw Eng Methodol 28(3):15

Rossi PH, Nock SL (1983) Measuring social judgments : the factorial survey approach. Social Forces 12:598

da S Maldonado E, Shihab E (2015) Detecting and quantifying different types of self-admitted technical debt. In: 7th IEEE International Workshop on Managing Technical Debt, MTD@ICSME 2015, Bremen, Germany, October 2, 2015, pp 9–15

da S Maldonado E, Abdalkareem R, Shihab E, Serebrenik A (2017) An empirical study on the removal of self-admitted technical debt. In: ICSME, pp 238–248

da S Maldonado E, Shihab E, Tsantalis N (2017) Using natural language processing to automatically detect self-admitted technical debt. IEEE Trans Software Eng 43(11):1044–1062

Scherer KR, Wranik T, Sangsue J, Tran V, Scherer U (2004) Emotions in everyday life: probability of occurrence, risk factors, appraisal and reaction patterns. Social Science Information 43(4):499–570, DOI 10.1177/0539018404047701, URL https://doi.org/10.1177/0539018404047701, https://doi.org/10.1177/0539018404047701

Seaman C, Guo Y (2011) Measuring and monitoring technical debt. Advances in Computers

Serebrenik A (2017) Emotional labor of software engineers. In: Demeyer S, Parsai A, Laghari G, van Bladel B (eds) Proceedings of the 16th edition of the BElgian-NEtherlands software eVOLution symposium, Antwerp, Belgium, December 4-5, 2017., CEUR-WS.org, CEUR Workshop Proceedings, vol 2047, pp 1–6

Sparks BA, Browning V (2011) The impact of online reviews on hotel booking intentions and perception of trust. Tourism Management 32(6):1310–1323, DOI https://doi.org/10.1016/j.tourman.2010.12.011, URL https://www.sciencedirect.com/science/article/pii/S0261517711000033

Spencer D (2009) Card sorting: Designing usable categories. Rosenfeld Media

Steinmacher I, Conte T, Gerosa MA, Redmiles D (2015) Social barriers faced by newcomers placing their first contribution in open source software projects. In: CSCW 2015, Association for Computing Machinery, New York, NY, USA, CSCW '15, p 1379–1392, DOI 10.1145/2675133.2675215, URL https://doi.org/10.1145/2675133.2675215

Storey MA (2012) The evolution of the social programmer. In: Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, IEEE Press, MSR '12, p 140

Storey MA, Ryall J, Bull RI, Myers D, Singer J (2008) Todo or to bug: Exploring how task annotations play a role in the work practices of software developers. In: Proceedings of the 30th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '08, p 251–260, DOI 10.1145/1368088.1368123, URL https://doi.org/10.1145/1368088.1368123

Tourani P, Adams B, Serebrenik A (2017) Code of conduct in open source projects. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 24–33, DOI 10.1109/SANER.2017.7884606

Uddin G, Khomh F (2017) Opiner: An opinion search and summarization engine for apis. In: Proceedings of the 32nd IEEE/ACM International Conference on Auto-

mated Software Engineering, IEEE Press, ASE 2017, p 978–983

Wehaibi S, Shihab E, Guerrouj L (2016) Examining the impact of self-admitted technical debt on software quality. In: IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1, pp 179–188

Wright P (1974) The harassed decision maker: Time pressures, distractions, and the use of evidence. Journal of Applied Psychology 59(5):555–561

Xavier L, Ferreira F, Brito R, Valente MT (2020) Beyond the code: Mining self-admitted technical debt in issue tracker systems. In: Proceedings of the 17th International Conference on Mining Software Repositories, Association for Computing Machinery, New York, NY, USA, MSR '20, p 137–146, DOI 10.1145/3379597.3387459, URL https://doi.org/10.1145/3379597.3387459

Yin D, Bond SD, Zhang H (2010) Are bad reviews always stronger than good? asymmetric negativity bias in the formation of online consumer trust. In: Sabherwal R, Sumner M (eds) Proceedings of the International Conference on Information Systems, ICIS 2010, Saint Louis, Missouri, USA, December 12-15, 2010, Association for Information Systems, p 193, URL http://aisel.aisnet.org/icis2010_submissions/193

Zampetti F, Noiseux C, Antoniol G, Khomh F, Di Penta M (2017) Recommending when design technical debt should be self-admitted. In: International Conference on Software Maintenance and Evolution, IEEE Computer Society, pp 216–226

Zampetti F, Serebrenik A, Di Penta M (2018) Was self-admitted technical debt removal a real removal?: an in-depth perspective. In: Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018, pp 526–536

Zampetti F, Serebrenik A, Di Penta M (2020) Automatically learning patterns for self-admitted technical debt removal. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 355–366

Zampetti F, Fucci G, Serebrenik A, Di Penta M (2021) Self-admitted technical debt practices: a comparison between industry and open-source. Empir Softw Eng 26(6):131, DOI 10.1007/s10664-021-10031-3, URL https://doi.org/10.1007/s10664-021-10031-3

Zazworka N, Shaw MA, Shull F, Seaman CB (2011) Investigating the impact of design debt on software quality. In: Proceedings of the 2nd Workshop on Managing Technical Debt, MTD 2011, Waikiki, Honolulu, HI, USA, May 23, 2011, pp 17–23