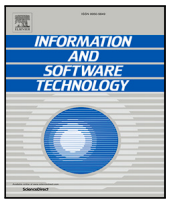
Information and Software Technology 187 (2025) 107862

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



# Binary and multi-class classification of Self-Admitted Technical Debt: How far can we go?

Francesca Arcelli Fontana [a], Juri Di Rocco [b], Davide Di Ruscio [b,*], Amleto Di Salle [c], Phuong T. Nguyen [b]

[a] Università degli studi di Milano–Bicocca, 20126 Milano, Italy
[b] Università degli studi dell'Aquila, 67100 L'Aquila, Italy
[c] Gran Sasso Science Institute, 67100 L'Aquila, Italy

ARTICLE INFO

*Keywords:*
Self-Admitted Technical Debt
NLP techniques
Neural networks
Large Language Models

ABSTRACT

**Context:** Aiming for a trade-off between short-term efficiency and long-term stability, software teams resort to sub-optimal solutions, neglecting the best software development practices. Such solutions may induce technical debt (TD), triggering maintenance issues. To facilitate future fixing, developers mark code with any issues using textual comments, resulting in Self-Admitted Technical Debt (SATD). Detecting SATD in source code is crucial since it helps programmers locate potentially erroneous snippets, allowing for suitable interventions, and improving code quality. There are two main types of SATD detection, i.e., *binary classification* and *multi-class classification*, grouping TD comments into SATD/Non-SATD categories, and multiple categories, respectively.
**Objective:** We attempt to understand to which extent state-of-the-art research has addressed the issue of detecting SATD, both binary and multi-class classification. Based on this investigation, we also propose a practical approach for the detection of SATD using Large Language Models (LLMs).
**Methods:** First, we conducted a literature review to understand to which extent the two types of classification have been tackled by existing research. Second, we developed `SALA`, a dual-purpose tool on top of Natural Language Processing (NLP) techniques and neural networks to deal with both types of classification. An empirical evaluation has been performed to compare `SALA` with state-of-the-art baselines.
**Results:** The literature review reveals that while binary classification has been well studied, multi-class classification has not received adequate attention. The empirical evaluation shows that `SALA` obtains a promising performance, and outperforms the baselines with respect to various quality metrics.
**Conclusion:** We conclude that more effort needs to be spent to tackle multi-class classification of SATD. To this end, LLMs hold the potential, albeit with more rigorous investigation on possible fine-tuning and prompt engineering strategies.

## 1. Introduction

While working on a software project, developers usually tolerate code containing various issues, aiming for a trade-off between short-term efficiency and long-term stability [1,2]. However, to guarantee software quality, later on, these issues need to receive proper treatment from the same developers or even different ones. To this end, developers have to mark such code with textual comments to signal the presence of any possible issues that were left unresolved. These comments constitute Self-Admitted Technical Debt [3,4] (SATD), i.e., technical debt intentionally manifested by developers. Therefore, detecting SATD embedded in source code as textual comments is an important phase of the software development cycle, allowing programmers to enhance program comprehension and reliability.

In practice, there are two types of SATD detection mechanisms, i.e., binary and multi-class classification [2]. The former deals with detecting SATD and non-SATD comments, whereas the latter goes one step further to detect various types of SATD. In fact, while segregating comments into SATD and non-SATD groups is important in identifying the presence of technical debt, classifying them into refined sub-categories helps developers better understand specific types of debt they have to deal with. Studying both types of SATD classification is essential in the context of detecting and solving technical debt. So far, there have

* Corresponding author.
*E-mail addresses:* francesca.arcelli@unimib.it (F. Arcelli Fontana), juri.dirocco@univaq.it (J. Di Rocco), davide.diruscio@univaq.it (D. Di Ruscio), amleto.disalle@gssi.it (A. Di Salle), phuong.nguyen@univaq.it (P.T. Nguyen).

https://doi.org/10.1016/j.infsof.2025.107862
Received 26 March 2025; Received in revised form 25 July 2025; Accepted 28 July 2025
Available online 7 August 2025

been several papers on SATD, including systematic literature reviews, investigating various aspects of the research issue. Sierra et al. [5] reviewed past and current studies in the detection, comprehension, and repayment of SATD. Though various research issues have been widely studied, according to our observation, no work has been done to investigate in detail the two types of SATD classification in a cohesive manner.

To address both types of classification, in our previous work [6], we implemented PILOT, a technical debt detector on top of a combination of various natural language processing (NLP) and machine learning (ML) algorithms. Different processing steps are conducted from the input data to extract the semantics among SATD comments. Afterward, we built a classification engine using neural networks to recognize SATD comments. In this paper, we leverage the capabilities of Large Language Models (LLMs) with the aim of enhancing the overall prediction accuracy. Our contribution in this paper is twofold as follows. *First*, we investigate how well the aforementioned types of classification have been addressed in the literature, following a process for searching relevant studies. Compared to existing studies, our work is different, as it does not investigate a broad range of issues related to SATD, but concentrates only on studying a specific perspective, i.e., the ability to perform binary classification and multiclass classification of SATD comments. *Second*, we conceive SALA, an enhanced **SA**TD detector on top of **LA**rge Language Models. By means of an empirical evaluation, we demonstrate that our proposed framework obtains promising performance and outperforms state-of-the-art approaches, implying its feasibility in practical use.

In summary, we perform both a quantitative analysis and a qualitative evaluation to answer the following research questions:

- ***RQ*$_1$***: To what extent has the existing literature explored the binary and multi-class classifications in SATD?* This research question investigates the extent to which state-of-the-art approaches address binary and multi-class classification of SATD. In addition, it aims to identify the most effective baselines for addressing the other research questions.
- ***RQ*$_2$***: How does SALA perform in binary classification compared to state-of-the-art baselines?* SATD has been extensively investigated in the literature, with numerous automated approaches proposed to detect SATD in code and comments. Most of these approaches focus on binary classification. In this study, we evaluate the performance of SALA by comparing it with different state-of-the-art baselines, i.e., DebtHunter [P14], GNNSI [P19], SATD-LN [P21], PILOT [P23], SATD-LN [P25], and Aiken et al. [P26].
- ***RQ*$_3$***: How does SALA perform in multi-class classification compared to state-of-the-art baselines?* The multi-class classification of SATD has recently started gaining attention within the research community. Several baselines, e.g., DebtHunter [P14], Qu et al. [P22], PILOT [P23], Yu et al. [P24], SCGRU [P27] have been proposed to classify SATD comments into five categories. In this research question, we evaluate SALA against these baselines to determine whether our proposed approach outperforms established methods.

The main contributions of this paper are summarized below:

1. A literature review on key Software Engineering venues to investigate how binary and multi-class classification of SATD have been addressed by state-of-the-art research.
2. SALA, a novel framework for detecting SATD from comments, leveraging NLP and ML techniques.
3. An empirical evaluation and comparison of SALA with well-founded approaches using real-world SATD datasets.
4. The replication package and the curated metadata have been published to facilitate future research.[1]

[1] https://github.com/gssi/SALA-SATD

**Organization.** In Section 2, we provide the background related to Technical Debt and Self-Admitted Technical Debt. In this section, we also present a qualitative study to investigate existing literature, thereby answering **RQ**$_1$. In Section 3, we describe in detail the proposed SALA approach with the underlying techniques and constituent components. Moreover, we also conduct an empirical evaluation, and report and analyze the quantitative and qualitative experimental results, addressing **RQ**$_2$ and **RQ**$_3$. Moreover, we discuss the threats that might impact the validity of our findings. The related work is reviewed in Section 4, and finally, the paper is concluded in Section 5.

## 2. Qualitative study

We start this section by reviewing basic terminology related to technical debt in Section 2.1. Afterward, the process conducted to perform a literature analysis is described in Section 2.2. The analysis results are reported in Section 2.3.

### 2.1. Terminology

During the development process, software teams might opt for sub-optimal solutions, which do not adhere to the best software development practices, aiming for a trade-off between short-term efficiency and long-term stability [1,2]. Unfortunately, such solutions may induce technical debt, thereby triggering maintenance issues. Essentially, technical debt is a term used to describe the situation where developers sacrifice long-term code quality for short-term objectives [7].

To signal the presence of any possible issues, developers manifest technical debt contained in source code with textual comments. These intentionally manifested comments are called self-admitted technical debt [3] (SATD), which signals further treatment. There are two types of SATD detection, i.e., binary and multi-class classification [2]. While binary classification tackles the detection of only two categories, i.e., SATD and non-SATD comments, by multi-class classification, there are five distinct SATD types, as shown in Table 1 [8].

Potdar et al. [3] were among the first authors who proposed the concept of self-admitted technical debt as a specific type of technical debt introduced directly by developers into source code comments. They manually analyzed 101,762 comments from 4 large open-source Java projects and extracted 62 comment patterns that commonly indicate the presence of SATDs.

### 2.2. Literature analysis (*RQ*$_1$)

This section presents a literature review to survey state-of-the-art research in detecting SATD. To collect the related studies, we followed a lightweight process defined by the set of guidelines for a systematic literature review (SLR), conceived by Kitchenham et al. [9]. Moreover, we also incorporated the search strategy proposed by Zhang et al. [10], aiming to cover more relevant state-of-the-art work. Such a search process has been successfully applied in a recent study for a similar purpose [11]. Section 2.2.1 describes the search process, whereas Section 2.2.2 clarifies the required steps to select suitable studies for the literature review.

#### 2.2.1. Search process

Zhang et al. [10] defined a quasi-gold standard-based systematic search approach, collecting relevant studies by means of four "*W*"-questions, i.e., *Which?*, *Where?*, *What?*, and *When?*. This work applies the same strategy to look for related papers, explained as follows.

▷ **Which?** We conducted both manual and automatic searches as suggested by Zhang et al. [10].

▷ **Where?** Concerning the manual search, we employed the same methodology used by Sierra et al. in their SLR [5]. In particular, we searched for papers that cited the seminal paper in SATD detection [3]

**Table 1**
Categories of SATD comments.

| Name | Description | Example |
|---|---|---|
| DEFECT | Comments referring to code with possible defects, bugs, unexpected behavior, or failures. | "*// FIXME formatters are not thread-safe*" from Apache Ant |
| DESIGN | Debt that are detected by means of source code analysis and violations of the object-oriented design principles | "*//TODO: Return content encoding according to the mime part in the mail*" from Columba |
| DOCUMENTATION | Issues found in software project documentation, i.e., comments that signal missing, outdated, inadequate, or incomplete documentation | "*// FIXME: Document difference between warn and warning (or rename one better)*" from JRuby |
| REQUIREMENT | Trade-offs for what requirements need to be implemented or how to implement them; a method, class or program that presents an incompleteness | "*// TODO save hostname; save sample type (plain or http)*" from Apache JMeter |
| TEST | Issues related to testing activities that can impact on the quality of testing process | "*// TODO: This requires that the tests be run with assertions*" from ArgoUML |

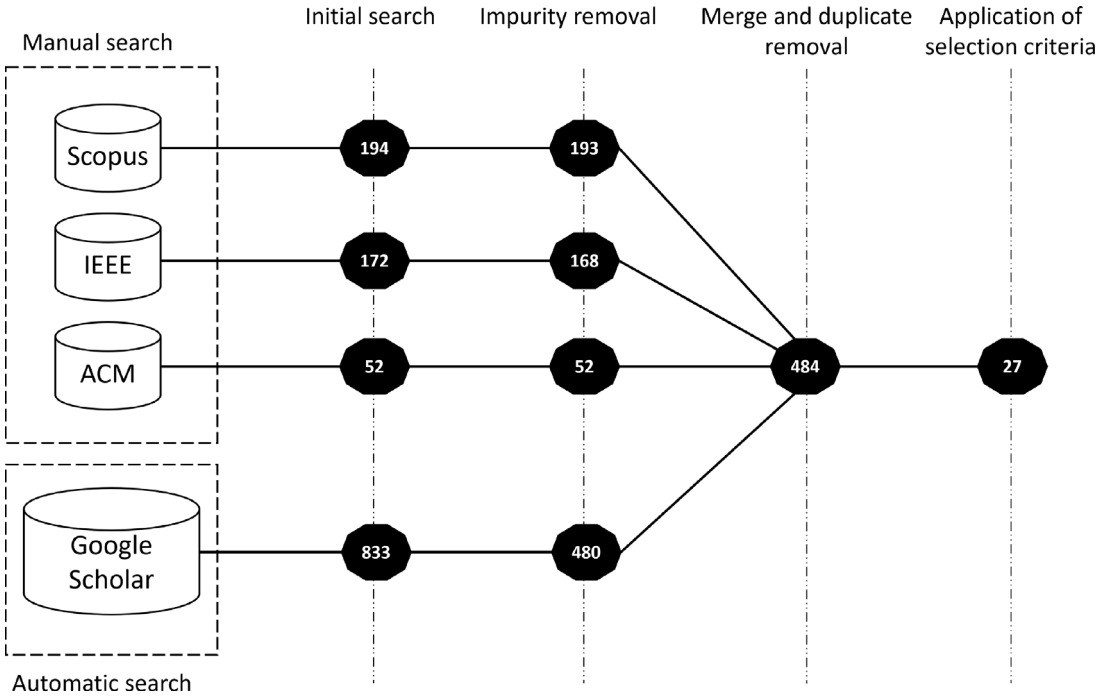



**Fig. 1.** The search and selection process to identify relevant publications.

from the following databases: Scopus,[2] IEEE,[3] and ACM.[4] For the automatic search, we leveraged the Google Scholar engine,[5] using its advanced search and export features.

▷ **What?** For the manual search, we extracted the cited papers from the following links: https://tinyurl.com/36c7z9he (Scopus), https://ieeexplore.ieee.org/document/6976075/citations#citations (IEEE), and https://dl.acm.org/doi/10.1109/ICSME.2014.31 (ACM). Then, for the automatic search, we employed Google Scholar to identify the papers related to self-admitted technical debt by using the following search string ‘‘`self AND admitted AND technical AND debt`’’.

▷ **When?** Since the SATD term was first defined in the seminal paper by Potdar and Shihab [3] about self-admitted technical debt, published in October 2014, we limit our search to the period from October 1st, 2014, to April 15th, 2024.

### 2.2.2. Selection process

Fig. 1 depicts the search and selection process that was applied to collect the relevant studies from the four aforementioned databases, i.e., Scopus, IEEE, ACM, and Google Scholar. In the search process, we obtained 194 studies for Scopus, 172 for IEEE, 52 for ACM, and 833 for Google Scholar, respectively. The stages of the selection process applied to the items retrieved from the *initial search* are described as follows.

▷ **Impurity removal.** In this stage, we manually removed the documents that are clearly not research papers, including thesis, books, reports, and unavailable articles, e.g., those published in a journal or presented at a conference. Concerning the manual databases, we obtained 193 and 168 studies for the Scopus and IEEE databases, respectively, whereas, for the ACM database, we kept all papers. For the Google Scholar database, we got 480 studies after executing the impurity removal step.

[2] https://www.scopus.com
[3] https://ieeexplore.ieee.org/Xplore/home.jsp
[4] https://dl.acm.org/search/advanced
[5] https://scholar.google.com/

**Table 2**
Inclusion and exclusion criteria.

| Inclusion criteria | Exclusion criteria |
|---|---|
| Studies exploiting pattern-based techniques to identify SATD instances among all the source code comments automatically and to classify them into binary or multi classes | Studies not containing SATD as the main topic |
| Studies introducing machine learning classifiers to identify SATD instances among all the source code comments automatically and to classify them into binary two or multi classes | Studied being related to the comprehension and repayment of SATD |
| Studies analyzing source-code comments | Studies analyzing commit and/or issue comments |
| Studies published in English | Duplicated studies, i.e., we only considered the extended journal version instead of the conference/workshop one |
| – | Studies being related to secondary or tertiary studies, e.g., surveys, mapping studies, systematic literature reviews, to name a few |
| – | Studies being not available as full-text |

▷ **Merge and duplicate removal.** We merged all papers from manual and automatic sources into a single corpus. After removing all the duplicated articles, we obtained 484 studies.

▷ **Application of selection criteria.** In the last stage, we filtered all the papers using the inclusion and exclusion criteria in Table 2. In particular, we are interested in only studies that deal with the binary and multi-class classification of SATD comments extracted from source code while ignoring those that deal with SATD in commits or issues. Moreover, only papers written in English and available as full-text are incorporated in the selection.

By applying the inclusion and exclusion criteria, we retrieved 27 final studies listed in Appendix. We read and analyzed them to understand the type of supported detection and the corresponding classification type.

### 2.3. Existing research on the detection of SATD

**RQ$_1$**

To what extent has the existing literature explored the binary and multi-class classifications in SATD?

The results related to the type of classification that the considered papers deal with are shown in Table 3: it is evident that almost all of the studies (i.e., 25 papers) use Machine Learning techniques to classify SATD comments, and only 2 of them, i.e., HATD [P9] and Jitterbug [P10] are pattern-based approaches. Concerning classification type, 18 out of 27 papers were dedicated to dealing only with binary classification, 7 out of 27 papers worked with multi-class classification, and the remaining two studies employed both binary and multi-class classification. It is worth noting that even a recent paper, i.e., Aiken et al. [P26] published in 2023, concentrates only on binary classification. Overall, Table 3 suggests that while the research community has well-investigated binary classification, the multi-class classification remains largely unexplored. Among the approaches, DebtHunter [P14] and PILOT [P23] are the only ones capable of performing both types of classification.

Next, we show an analysis of the papers to identify baselines for binary and multi-class classification used to compare our approach.

▷ **Binary classification.** Table 3 shows that 18 studies refer to binary classification. We decided to employ the most recent studies, i.e., starting in 2022, as baselines. The only paper we did not consider in 2022 is [P20], i.e., DebtFree. We excluded it from the baselines since the approach aims to reduce the labeling effort by human experts for technical debt. Moreover, we also discarded the two studies that use pattern-based and machine-learning techniques since our work only focuses on machine-learning techniques. We considered two other studies

**Table 3**
Details of the considered papers and classification types.

| ID | Detection type | | Classification type | | Year |
|---|---|---|---|---|---|
| | Pattern-based | Machine learning | Binary | Multi-class | |
| [P1] | | ✓ | ✓ | | 2017 |
| [P2] | | ✓ | | ✓ | 2017 |
| [P3] | | ✓ | ✓ | | 2018 |
| [P4] | | ✓ | | ✓ | 2019 |
| [P5] | | ✓ | ✓ | | 2019 |
| [P6] | | ✓ | ✓ | | 2019 |
| [P7] | | ✓ | ✓ | | 2019 |
| [P8] | | ✓ | ✓ | | 2020 |
| [P9] | ✓ | ✓ | ✓ | | 2020 |
| [P10] | ✓ | ✓ | ✓ | | 2022 |
| [P11] | | ✓ | | ✓ | 2020 |
| [P12] | | ✓ | ✓ | | 2020 |
| [P13] | | ✓ | ✓ | | 2021 |
| [P14] | | ✓ | ✓ | ✓ | 2021 |
| [P15] | | ✓ | ✓ | | 2021 |
| [P16] | | ✓ | ✓ | | 2021 |
| [P17] | | ✓ | | ✓ | 2022 |
| [P18] | | ✓ | ✓ | | 2021 |
| [P19] | | ✓ | ✓ | | 2022 |
| [P20] | | ✓ | ✓ | | 2022 |
| [P21] | | ✓ | ✓ | | 2022 |
| [P22] | | ✓ | | ✓ | 2023 |
| [P23] | | ✓ | ✓ | ✓ | 2022 |
| [P24] | | ✓ | | ✓ | 2023 |
| [P25] | | ✓ | ✓ | | 2023 |
| [P26] | | ✓ | ✓ | | 2023 |
| [P27] | | ✓ | | ✓ | 2023 |

that classify SATD as binary and multi-class. Therefore, the considered studies as baselines for binary classification are: DebtHunter [P14], GNNSI [P19], GGSATD [P21], PILOT [P23], SATD-LN [P25], and Aiken et al. [P26].

▷ **Multi-class classification.** As we pointed out in Table 3, there are 9 studies that perform multi-class classification Maldonado et al. [P2], Wattanakriengkrai et al. [P4], Meneses Santos et al. [P11], DebtHunter [P14], Chen et al. [P17], Qu et al. [P22], PILOT [P23], Yu et al. [P24], and SCGRU [P27]. However, not all classify the data into five categories listed in Table 1. In particular, Maldonado et al. [P2], Wattanakriengkrai et al. [P4], and Meneses Santos et al. [P11] conducted the classification only with two categories, i.e., `DESIGN` and `REQUIREMENT`. Maldonado et al. [P2] got an F1-score of 0.620 and 0.403 for `DESIGN` and `REQUIREMENT`, respectively. Similarly, Wattanakriengkrai et al. [P4] obtained 0.203, 0.894, 0.404 as Precision, Recall, and F1-score for the `DESIGN` category. Meanwhile by the `REQUIREMENT` category, the performance is substantially improved,

i.e., Wattanakriengkrai et al. [P4] got 0.754, 0.603, 0.667 as Precision, Recall, and F1-score, respectively. Meneses Santos et al. [P11] compared their approach with auto-sklearn and Maximum entropy techniques. The results show that their model enhanced the recall and f-measure in DESIGN classification and recall in REQUIREMENT. We did not select these studies as baselines since they only classified two categories out of five.

Chen et al. [P17] and Yu et al. [P24] applied eXtreme Gradient Boosting (XGBoost) and Jensen–Shannon divergence Generative Adversarial Network (JSD-GAN) techniques, respectively, to classify SATD into three categories, i.e., DESIGN, DEFECT, and REQUIREMENT.[6] Yu et al. compared their approach with XGBoost, finding that the JSD-GAN model outperforms the XGBoost except for the Recall metric in the DESIGN category, i.e., 0.9140 versus 0.8753. Therefore, we selected the Yu et al. [P24] approach as a baseline in our experiment.

The remaining studies, i.e., DebtHunter [P14], Qu et al. [P22], PILOT [P23], and SCGRU [P27], use ML approaches to classify SATD in all five categories. Therefore, we also selected these studies as baselines for multi-class classification.

**Answer to $RQ_1$.** The literature review reveals that most existing studies have dealt with binary classification, while multi-class classification has received less attention. We selected the following studies for binary and multi-class classification, respectively:

- **Binary:** GNNSI [P19], GGSATD [P21], SATD-LN [P25], and Aiken et al. [P26]
- **Multi-class:** Qu et al. [P22], Yu et al. [P24], and SCGRU [P27]

We also identified as baselines the only two approaches classifying binary and multi-class, i.e., DebtHunter [P14] and PILOT [P23].

## 3. Quantitative study

**Motivation and Focus**. The qualitative findings from $\mathbf{RQ}_1$ (Section 2.3) indicate that prior studies on Self-Admitted Technical Debt (SATD) have predominantly treated SATD detection as a binary classification task, with multi-class classification receiving significantly less attention. This gap in the literature motivates us to develop an approach that addresses both binary and multi-class classification of SATD comments by leveraging the capabilities of Large Language Models (LLMs). In doing so, we aim to leverage LLMs' strong natural language understanding to improve classification performance in both types of tasks.

**Summary of the Contributions**. In this section, we introduce our proposed tool — named SALA— for the detection of SATD using LLMs. More importantly, we report the study conducted using real word datasets to evaluate the effectiveness of the proposed tool, as well as to compare it with state-of-the-art approaches. Different from most of the state-of-the-art work, which often examines a broad array of SATD-related issues, our research narrows down the scope to this specific classification perspective. In particular, we focused on evaluating how effectively SATD comments can be identified (binary classification, distinguishing SATD vs. non-SATD) and further categorized into distinct technical debt types (multi-class classification of SATD comments). By concentrating exclusively on these classification capabilities, our approach provides a targeted analysis that is unique in the SATD literature. This perspective allows us to thoroughly investigate and demonstrate the effectiveness of LLMs-based techniques for SATD classification, offering a deeper understanding of SATD detection and categorization that complement the broader investigations found in existing studies.

**Structure**. We introduced the newly conceived tool in Section 3.1. Afterward, we explain in detail the empirical conducted to study the proposed tool. In particular, Sections 3.2 and 3.3 present the datasets and metrics, respectively, while Section 3.4 describes the experimental configurations. In Sections 3.5 and 3.6, we report and analyze the obtained results related to the comparison with the baselines. The possible threats to validity are highlighted in Section 3.7.

### 3.1. Sala: A dual-purpose machine learning SATD classifier

In this work, we develop a tool for the detection of SATD using a large language model. The proposed framework is shown in Fig. 2 with two main phases, i.e., training and testing. During training, labeled SATD comments are used as input, and the Data Parser module parses the input data using different text processing steps, i.e., stopword removal, stemming, lemmatization to produce features for the classification engine.

The CLASSIFIER component is used to perform the main classification task, and it can be built on top of various neural networks. Large language models (LLMs) have the ability to produce coherent and contextually relevant text based on input prompts or cues.

LLMs are able to capture intrinsic language patterns and contextual nuances, semantic relationships, and syntactic structures inherent in human language. The primary innovation of LLMs lies in their ability to learn rich, contextual representations of language through unsupervised pre-training on massive datasets. We leverage these characteristics to construct the classification engine of SALA.

This generative aspect facilitates a wide range of applications, including language translation, text summarization, question answering, sentiment analysis, and dialogue generation, to name but a few. LLMs have been conceptualized with sophisticated deep learning architectures, and pre-trained in vast textual corpora, thus, in this work, we utilized an LLM named LLaMA-2 to build the classification engine. To guide the learning process, we leverage different prompt engineering techniques. Essentially, prompt engineering is the process of designing effective queries or input patterns to guide the behavior of LLMs during inference, and it is used to steer the CLASSIFIER component. Users craft input text so as to elicit the desired response or behavior from the model. Prompt engineering is crucial for leveraging the capabilities of LLMs in various applications, including text generation, question answering, and problem-solving [12–14].

We make use of prompt engineering to provide the model with context and constraints that steer its output towards the desired outcome. This is related to specifying the task, providing relevant examples or instructions, and shaping the input to encourage the desired behavior while minimizing undesirable outputs such as biases or inaccuracies. Fig. 3 shows the prompt used to guide LLaMA-2 in generating predictions.

Once the training phase has been completed, i.e., the learning model is built, the deployment phase takes place. In this phase, SALA extracts comments from source code, using a state-of-the-art library for parsing Java source code such as Java Parser.[7] Then the other components can classify the comments SATD or non-SATD (binary classification) or one of the SATD categories (multi-class classification) defined in Table 1.

### 3.2. Dataset

To evaluate SALA, we make use of state-of-the-art dataset curated by existing study [8]. The dataset was curated by Maldonado et al. [8], and it contains $259K$ labeled source code comments belonging to the following 10 open source Java projects: Apache Ant, Apache JMeter, ArgoUML, Columba, Eclipse EMF, Hibernate, JEdit,

[6] Both papers used the IMPLEMENTATION class instead of the REQUIREMENT one.

[7] https://javaparser.org/

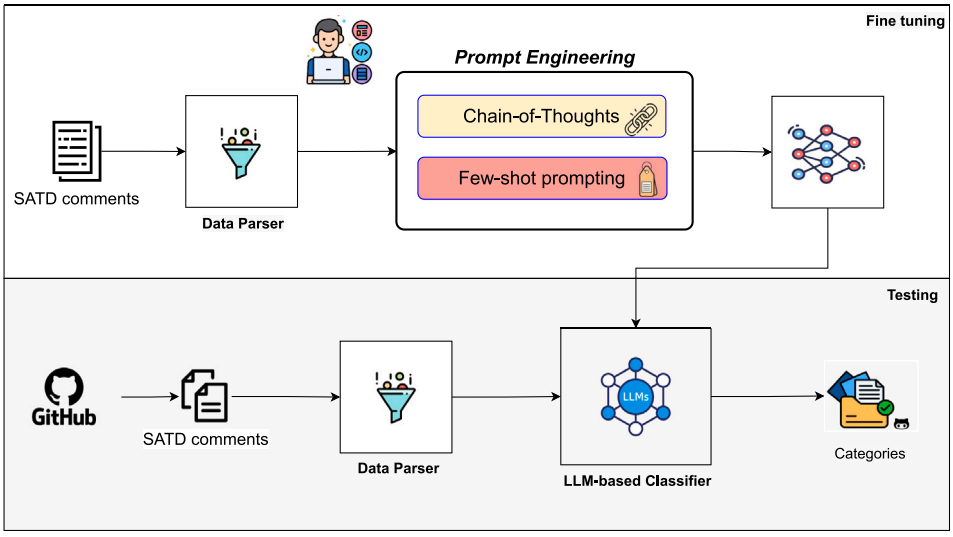



**Fig. 2.** The overall architecture.

```
"You are an AI assistant specialized in classifying code comments."
"Your task is to analyze the code comments. Classify the following code comment as
    DEFECT, DESIGN, DOCUMENTATION, IMPLEMENTATION, or TEST. Use just one class. Do not
     include any additional text."
{code comment}
```

**Fig. 3.** Prompt used in getting the response from LLMs.

**Table 4**
Distribution of the sets for binary classification in $D_1$ [8].

| Project | DEFECT | DESIGN | DOC | IMPL | TEST | NO SATD | TOTAL |
|---|---|---|---|---|---|---|---|
| ant-1.7.0 | 13 | 95 | 0 | 13 | 10 | 3,967 | 4,098 |
| jmeter-2.10 | 22 | 316 | 3 | 21 | 12 | 7,683 | 8,057 |
| argouml | 127 | 801 | 30 | 411 | 44 | 8,039 | 9,452 |
| columba | 13 | 126 | 16 | 43 | 6 | 6,264 | 6,468 |
| emf | 8 | 78 | 0 | 16 | 2 | 4,286 | 4,390 |
| hibernate | 52 | 355 | 1 | 64 | 0 | 2,496 | 2,968 |
| jEdit | 43 | 196 | 0 | 14 | 3 | 10,066 | 10,222 |
| jfreechart | 9 | 184 | 0 | 15 | 1 | 4,199 | 4,408 |
| jruby | 161 | 343 | 2 | 110 | 6 | 4,275 | 4,897 |
| sql12 | 24 | 209 | 2 | 50 | 1 | 6,929 | 7,215 |
| **Total** | **472** | **2,703** | **54** | **757** | **85** | **58,604** | **62,775** |
| SATD | | | **4,171** | | | | |

`JFreeChart`, `JRuby`, and `SQuirrel`. The authors used an open-source Eclipse plug-in to parse source code and extract code comments for each project. In particular, five heuristic rules have been applied to filter out comments that were obviously impossible to be SATD comments. Afterwards, 62,775 comments were manually classified. Table 4 provides a summary of the data collected by Maldonato et al.. There are 4,071 comments identified as containing SATD, which are further labeled into five independent categories, i.e., `DESIGN`, `DEFECT`, `DOCUMENTATION`, `REQUIREMENT`, and `TEST`. The `Non SATD` category accounts for the major of data instances, i.e., 93.35%, and the remaining 6.65% are `SATD` comments. This unbalanced data problem could manifest itself in suboptimal performance measures, especially for the class with a small number of samples. A summary of the training and testing data for binary and multi-class classification is shown in Tables 5 and 6.

**Table 5**
Distribution of the sets for binary classification in $D_1$ [8].

| Classification | Training set | Testing set | Total |
|---|---|---|---|
| `SATD` | 3,234 | 837 | 4,071 |
| `NO-SATD` | 46,586 | 11,618 | 58,204 |
| **Total** | 49,820 (80%) | 12,455 (20%) | 62,275 |

**Table 6**
Distribution of the sets for multiclass classification in $D_1$ [8].

| Category | Training set | Test set | Total |
|---|---|---|---|
| `DEFECT` | 382 | 90 | 472 |
| `DESIGN` | 2,189 | 514 | 2,703 |
| `DOCUMENTATION` | 44 | 11 | 54 |
| `REQUIREMENT` | 613 | 152 | 757 |
| `TEST` | 69 | 17 | 85 |
| **Total** | 3,297 (80%) | 784 (20%) | 4,071 |

**Table 7**
Evaluation features of different approaches [P25], [P23], [P14], [P26], [P19], and [P21].

| | SATD-LN [P25] | PILOT [P23] | DebtHunter [P14] |
|---|---|---|---|
| Evaluation methodology | intra-project and cross-project validation | 10 fold cross-validation | stratified 10 fold cross-validation |
| Metrics reported | $F_1$-score | precision, recall, $F_1$-score | precision, recall, $F_1$-score |
| Result aggregation | average for projects | SATD/Non-SATD | average for project |
| | **Aiken et al.** [P26] | **GNNSI** [P19] | **GGSATD** [P21] |
| Evaluation methodology | cross- and intra-project validation | 10-fold cross-validation | cross- and intra-project validation |
| Metrics Reported | $F_1$-score | precision, recall, $F_1$-score | precision, recall, $F_1$-score |
| Result aggregation | average for projects | average for projects | average for projects |

### 3.3. Metrics

We classify a set of SATD comments into distinct categories in the evaluation. There are the following definitions: *(i) True positive (TP)*: the comments that are correctly classified; *(ii) True negative (TN)*: comments classified to other categories, and they do not belong to the ground-truth category; *(iii) False positive (FP)*: comments classified but they do not belong to the ground-truth data; *(iv) False negative (FN)*: comments wrongly classified to other categories, and they belong to the ground-truth data.

We measure how well the predicted categories match with the manually classified ones using

*Precision*, *Recall*, and $F_1$*-score* defined as follows.

- **Precision.** Precision measures the fraction of the number of correctly classified comments to a category to the total number of classified comments in the category: $P = TP/(TP + FP)$;
- **Recall.** It is the ratio of the number of correctly classified items to the total number of items in the ground-truth data. Recall is also named as true positive rate (TPR): $R = TPR = TP/(TP + FN)$.
- **$F_1$-score**. The metric is a combination of precision and recall: $F_{1c} = (2 \cdot P \cdot R)/(P + R)$.

### 3.4. Settings

We conducted our experiments by fine-tuning the LLaMA-2 model, using an 80–20 split in the data set, with 80% allocated for fine-tuning the llama models and 20% for testing. This division aims to evaluate the performance of the model by reserving a substantial portion of the data for model training. In contrast, the remaining data serve as an independent test set to assess generalization. Tables 5 and 6 provide details of the splits. It is important to note that the selected testing data reflect the overall distribution of classes in the dataset, whether binary or multi-class. As reported in the following sections, many of the considered baselines lack available replication packages; therefore, we utilized the data provided in their respective papers to ensure a consistent and fair comparison. This approach allows for a straightforward and reliable assessment of the accuracy of the model across comparable datasets.

### 3.5. Comparison with binary classification approaches

**RQ$_2$**

How does SALA perform in binary classification compared to state-of-the-art baselines?

To answer this question, we investigated how well SALA identifies SATD comments. Since the seminal work [8] has been published, different approaches have been proposed to identify SATD comments. As observed in RQ$_1$, several recent approaches have been proposed to identify SATD comments and the referenced works, including DebtHunter [P14], PILOT [P23], SATD-LN [P21], Aiken et al. [P26] and GNNSI [P19]. A detailed analysis of the selected studies revealed only that PILOT [P23] and DebtHunter provide a replication package available. To assess the performance of SALA in binary SATD classification, we compare it against state-of-the-art approaches, analyzing their evaluation methodologies and experimental settings. Since some baselines provide replication packages while others do not, we replicated experiments for DebtHunter [P14] and PILOT [P23], while relying on reported results for GNNSI [P19], SATD-LN [P21], and Aiken et al. [P26]. Notably, for Aiken et al. [P26], we considered the extra-project experimental results, as they align more closely with our evaluation approach.

Table 7 summarizes the evaluation methodologies of the selected approaches. DebtHunter [P14] employs 10-fold cross-validation across the entire dataset, ensuring robust performance estimation. The dataset used is the Maldonado et al. dataset [8], and training and testing data are randomly split across the folds. The primary evaluation metrics include Precision (P), Recall (R), and $F_1$-score, aggregated over SATD and Non-SATD classes. Since DebtHunter provides a replication package, we successfully replicated its results using the same dataset and training configurations. Similarly to DebtHunter, PILOT [P23] applies a 10-fold cross-validation on the Maldonado dataset. The data is randomly partitioned across the folds, and performance is assessed using Precision, Recall, and $F_1$-score, aggregated by SATD and Non-SATD classes. Given that PILOT provides a replication package, we replicated its experiments to ensure a fair comparison with SALA.

GNNSI [P19] follows a fixed training-validation-test split rather than cross-validation. The dataset is divided into 80% training, 10% validation, and 10% testing. The performance is measured using Precision, Recall, and $F_1$-score, aggregated across SATD and Non-SATD classes. Unlike DebtHunter and PILOT, GNNSI does not provide a replication package, and we therefore rely on reported results for comparison. Aiken et al. [P26] conduct both cross-project and intra-project experiments using the Maldonado dataset. In the cross-project setup, data from nine projects is used for training, while the remaining project serves as the test set. In contrast, intra-project evaluation applies 10-fold cross-validation within each project. Since this evaluation paradigm better aligns with our methodology, we considered the extra-project results. Unlike previous baselines, Aiken et al. emphasize Forced Minority Re-Sampling (FMR) and Data Augmentation to address class imbalance. Their primary metric is $F_1$-score, aggregated at the project level. However, due to the lack of a replication package, we relied on the results reported in the paper achieved by the best configuration, i.e., FMR. Finally, is worth noting that all the results reported in these three last papers are shown in weighted averages between SATD and Non-SATD classification. Table 7 provides a structured comparison of the evaluation methodologies adopted by each approach. In particular, the table reveals that the methodological difference belongs to different training strategies, handling of class imbalance, and cross-in-projects evaluation. Finally, GGSATD [P21] adopts both within-project and

**Table 8**
Precision, Recall, and $F_1$-score for binary classification with PILOT, DebtHunter, and SALA.

| | DebtHunter [P14] | | | PILOT [P23] | | | SALA | | |
|---|---|---|---|---|---|---|---|---|---|
| | P | R | $F_1$ | P | R | $F_1$ | P | R | $F_1$ |
| Non-SATD | 0.981 | 0.988 | 0.985 | 0.985 | 0.985 | 0.985 | **0.992** | **0.993** | **0.992** |
| SATD | 0.824 | 0.750 | 0.785 | 0.788 | 0.783 | 0.785 | **0.910** | **0.914** | **0.912** |
| Weighted avg. | 0.971 | 0.972 | 0.972 | 0.968 | 0.975 | 0.972 | **0.987** | **0.972** | **0.987** |
| Accuracy | | 0.971 | | | 0.976 | | | **0.988** | |

cross-project evaluation strategies. In our comparison (Table 9), we report results from the cross-project setting, where nine projects are used for training and the remaining one for testing, following a leave-one-project-out protocol. For each round, 10% of the training data is used for validation. Performance is evaluated using the Precision, Recall, and $F_1$-score, and the results are aggregated at the project level rather than by the SATD and non-SATD classes. Since the authors did not release a replication package and benchmark dataset, we reused the reported cross-project scores to ensure alignment with our evaluation set-up.

We replicated the experiments for DebtHunter and PILOT using the same training and testing data as the tool listed in Table 5. Table 8 presents the performance comparison of SALA against two state-of-the-art baselines, DebtHunter [P14] and PILOT [P23], for binary classification of SATD. The evaluation considers Precision (P), Recall (R), and $F_1$-score ($F_1$) for both SATD and Non-SATD classes, along with the weighted average scores and overall accuracy. The results show that SALA significantly outperforms both DebtHunter and PILOT when classifying SATD comments. Specifically, SALA achieves an $F_1$-score of 0.912 for the SATD class, substantially higher than PILOT (0.785) and DebtHunter (0.785). This improvement is driven by both higher precision (0.910 vs. 0.788 for PILOT and 0.824 for DebtHunter) and higher recall (0.914 vs. 0.783 for PILOT and 0.750 for DebtHunter), indicating that SALA detects SATD instances more accurately while reducing false negatives. In contrast, for the Non-SATD class, the improvement of SALA over the baselines is less pronounced. While SALA achieves the highest scores (P = 0.992, R = 0.993, $F_1$ = 0.992), PILOT and DebtHunter also perform well, both reaching an $F_1$-score of 0.985.

This discrepancy in performance gains between the SATD and Non-SATD classes suggests that SALA, powered by an LLM, is better equipped to handle imbalanced training data. Traditional approaches like DebtHunter and PILOT may struggle with the inherent imbalance in SATD datasets, where Non-SATD comments are often much more frequent than SATD ones. LLMs, in contrast, demonstrate greater robustness to data imbalance, allowing SALA to excel particularly in detecting SATD comments, where previous methods have suffered from lower recall. In terms of weighted average performance, SALA still leads with P = 0.987, R = 0.972, and $F_1$ = 0.987, slightly outperforming the baselines (0.972). Similarly, the overall accuracy of 0.988 further confirms the superior classification capability of SALA over PILOT (0.976) and DebtHunter (0.971).

These findings demonstrate that SALA is particularly effective in identifying SATD, significantly outperforming the baselines in this class while maintaining high performance for Non-SATD comments. This improvement highlights the advantage of LLM-based approaches in handling imbalanced datasets, which is a critical limitation of existing SATD classification techniques.

Table 9 presents the $F_1$ scores for the binary classification task of SALA compared with established baselines: SATD-LN [P25], Aiken et al. [P26], GNNSI [P19], and GGSATD [P21]. The results are reported across ten open-source projects and highlight the varying performance of each method. Overall, GGSATD achieves the highest average score among the baselines (0.862), followed by Aiken et al. (0.858) and SATD-LN (0.815), while GNNSI performs less consistently (0.795). These baselines often demonstrate strong results on specific datasets — e.g., GGSATD reaches 0.939 on Columba and 0.926 on JRuby — but no single baseline dominates across all projects. In contrast, SALA consistently ranks among the top two performers in nearly all cases, achieving the highest $F_1$ scores in 6 out of 10 projects, including a perfect score (1.000) on SQuirreL. Its average performance (0.879) surpasses that of all other approaches, underlining its robustness across heterogeneous codebases. These results suggest that SALA effectively leverages LLMs' contextual reasoning to identify SATD comments more reliably than existing techniques, while maintaining strong generalization across projects of varying complexity.

**Table 9**
$F_1$ for binary classification of SALA with the existing baselines.

| Project | SATD-LN [P25] | Aiken et al. [P26] | GNNSI [P19] | GGSATD [P21] | SALA |
|---|---|---|---|---|---|
| Apache Ant | 0.730 | **0.804** | 0.658 | 0.811 | 0.450 |
| ArgoUML | 0.911 | 0.921 | 0.894 | 0.926 | **0.937** |
| Columba | 0.859 | 0.850 | 0.930 | **0.939** | 0.757 |
| emf | 0.728 | 0.745 | 0.685 | 0.802 | **0.886** |
| Hibernate | 0.877 | 0.898 | 0.866 | 0.909 | **0.916** |
| JEdit | 0.778 | 0.767 | 0.616 | 0.782 | **0.882** |
| JFreeChart | 0.829 | 0.879 | 0.778 | 0.786 | **0.929** |
| JMeter | 0.829 | **0.923** | 0.864 | 0.902 | 0.882 |
| JRuby | 0.849 | **0.922** | 0.901 | 0.926 | 0.899 |
| SQuirreL | 0.755 | 0.869 | 0.758 | 0.834 | **1.000** |
| Average | 0.815 | 0.858 | 0.795 | 0.862 | **0.879** |

It is important to note that the results presented in Table 9 are averaged values, which are influenced by the imbalanced nature of the dataset. Since Non-SATD comments are more prevalent than SATD comments, they contribute more significantly to the average $F_1$ score. Nevertheless, this table serves as an aggregated summary of the detailed evaluations reported in Table 8, where $F_1$ scores were reported separately for SATD and Non-SATD categories. This broader aggregation supports the interpretation of SALA's consistent performance across diverse project settings. Moreover, we report only the $F_1$ score as the basis for comparison, as it effectively balances precision and recall and is sufficient to illustrate overall classification capability. However, the results should not be interpreted as a direct competitive analysis. Rather, Table 9 provides a high-level overview of how existing methods perform relative to SALA. The findings suggest that SALA consistently delivers strong results and demonstrates the viability of LLM-based solutions for automating SATD detection in software projects.

Table 10 reports the precision and recall values for SALA, GNNSI [P19], and GGSATD [P21], providing a more granular view of their performance beyond the $F_1$ scores. This is a further investigation of the performance of SALA with respect to the baselines. As shown in Table 7, SATD-LN [P25] and Aiken et al. [P26] are excluded from this comparison, as their original evaluations did not report these metrics. The results show that SALA consistently achieves high precision and recall across all projects, with particularly strong recall values—reaching 1.000 on the SQuirreL project and exceeding 0.9 in several others (e.g., Hibernate, ArgoUML, JFreeChart). GGSATD shows competitive precision (e.g., 0.942 on Columba and 0.921 on JRuby), but its recall is generally lower than that of SALA. GNNSI offers a more balanced trade-off between precision and recall but still underperforms relative to SALA. These findings reinforce the effectiveness of SALA's LLM-based strategy in identifying SATD comments, showing a superior ability to balance precision and recall across diverse software projects.

**Table 10**
Precision and recall for binary classification of SALA with the existing baselines.

| Project | GNNSI [P19] | GGSATD [P21] | SALA | GNNSI [P19] | GGSATD [P21] | SALA |
|---|---|---|---|---|---|---|
| | Precision | | | Recall | | |
| Apache Ant | 0.758 | **0.806** | 0.360 | 0.581 | **0.818** | 0.450 |
| ArgoUML | 0.847 | 0.909 | **0.941** | **0.947** | 0.946 | 0.928 |
| Columba | 0.961 | **0.936** | 0.694 | 0.902 | **0.942** | 0.758 |
| emf | 0.781 | 0.823 | **0.852** | 0.610 | 0.786 | **0.886** |
| Hibernate | 0.921 | **0.927** | 0.877 | 0.818 | 0.894 | **0.936** |
| JEdit | 0.648 | **0.864** | 0.846 | 0.587 | 0.720 | **0.882** |
| JFreeChart | 0.785 | 0.831 | **0.907** | 0.771 | 0.753 | **0.929** |
| JMeter | 0.904 | **0.905** | 0.835 | 0.827 | **0.899** | 0.873 |
| JRuby | 0.902 | **0.938** | 0.869 | 0.900 | **0.914** | 0.899 |
| SQuirreL | 0.839 | 0.851 | **1.000** | 0.693 | 0.820 | **1.000** |
| Average | 0.834 | **0.879** | 0.873 | 0.764 | 0.849 | **0.906** |

**Answer to $RQ_2$.** SALA demonstrates a superior performance in binary SATD classification compared to state-of-the-art baselines. In a fair comparison with DebtHunter [P14] and PILOT [P23], for which replication was feasible, SALA substantially outperforms both approaches, particularly in detecting SATD comments. Additionally, a broader comparison with other baselines — including SATD-LN [P25], Aiken et al. [P26], GNNSI [P19], and GGSATD [P21] — confirms the consistent and reliable performance of SALA across diverse projects. These results highlight the potential of LLM-based approaches for improving SATD detection, particularly in imbalanced and cross-project scenarios.

### 3.6. Comparison with multi-class classification approaches

**$RQ_3$**

How does SALA perform in multi-class classification compared to state-of-the-art baselines?

We investigate how well SALA performs in classifying SATD comments into five categories. The multi-class classification of SATD aims to categorize SATD comments into distinct types such as DEFECT, DESIGN, DOCUMENTATION, REQUIREMENT, and TEST. Despite its importance, this area has received less attention compared to binary classification. To address this gap, SALA was developed as a dual-purpose tool leveraging LLMs to classify SATD comments. In this section, we evaluate SALA's performance in multi-class classification, providing insights into its strengths and highlighting areas for improvement. Recent approaches have been proposed to address this gap, as indicated by $RQ_1$ and the references DebtHunter [P14], Qu et al. [P22], PILOT [P23], Yu et al. [P24], and SCGRU [P27]. After carefully analyzing the extracted papers, we were able to replicate the exact experiments with DebtHunter and PILOT.

Our analysis revealed that the approaches differ significantly in their evaluation methodologies. In particular, they differ in *(i)* Evaluation strategies, i.e., DebtHunter [P14] and PILOT [P23], employed 10-fold cross-validation on the entire dataset, while SCGRU [P27], used fixed splits (e.g., 10% training and 1% testing), *(ii)* subset selection, i.e., PILOT [P23] and Yu et al. [P24] restricted their analysis to specific SATD categories, and *(iii)* metrics, i.e., most studies reported precision, recall, and F1-scores, but aggregation methods varied (e.g., by project or category). It is important to note that all the proposed approaches evaluate their performances on the same seed dataset, i.e., Maldonado et al. [8], the one also involved in the SALA experimentations.

DebtHunter [P14] has been evaluated by means of 10-fold cross-validation applied to the entire dataset, reporting results as an average across all folds. SCGRU [P27] employed a training set comprising 10% of $D_1$ (6,227 code comments) and evaluated performance on a testing set consisting of 1% (622 code comments). The results were aggregated at the project level, i.e., the ten projects described in Section 3.2, and across the SATD categories. PILOT [P23] focused exclusively on SATD-classified comments, totaling 4,071 entries. The evaluation involved ten-fold cross-validation, with results aggregated by SATD categories. Yu et al. [P24] restricted the analysis to comments within three SATD categories, i.e., DESIGN, REQUIREMENT, and DEFECT. Then, a ten-fold cross-validation was performed, and SATD categories and projects aggregated the results. Finally, [P22] used ten-fold cross-validation, in which nine software projects were randomly selected, and data from these projects were randomly sampled and combined to form the training dataset. The remaining project served as the testing set to evaluate the model's performance. Excluding Qu et al. [P22] which involves only the $F_1$-score metrics, all the other approaches measure their classification performances by precision, recall, and $F_1$-score metrics. Table 11 resumes the main differences in the evaluation of the approaches considered.

Due to the availability of replication packages, we replicated the experiments for DebtHunter and PILOT using the same training and testing data as the tool listed in Table 6. This approach ensures a fair comparison. As a result, the findings for these two baselines can be directly compared to those of our method. Table 12 presents the results in terms of precision, recall, and $F_1$ scores, aggregated by SATD class. We opted for this aggregation strategy because different projects in the dataset contain very few occurrences of certain SATD classes. Additionally, our experiment does not aim to assess the performance of our approach based on intra-project relationships, where both the testing and training data originate from the same project, nor on extra-project relationships, where the testing data comes from a specific project while the language model is fine-tuned using data from other projects.

As seen in Table 12, SALA exhibits competitive performance in terms of precision, recall, and F1-score. Notably, SALA achieves near-perfect precision across all categories, with values consistently at or near 1.000. This performance significantly surpasses that of DebtHunter [P14] and PILOT [P23] in this regard, indicating that SALA is highly confident in its predictions and effectively minimizes false positives. However, recall varies across categories. For instance, SALA gets a recall of 0.914 for DESIGN and 0.850 for TEST. In contrast, its recall is lower for DEFECT (0.660) and REQUIREMENT (0.553). This discrepancy suggests that while SALA demonstrates high precision, it occasionally misses relevant instances, resulting in some classifications being overlooked.

In contrast, because of methodological differences, direct comparisons between the results of SCGRU [P27], Yu et al. [P24], and Qu et al. [P22] are not feasible. Therefore, the precision, recall, and F1 scores reported in Tables 13, 14, and 15 for these baselines should not be interpreted as a direct competitive analysis to determine the "best" approach.

Instead, these tables provide a high-level overview of the current landscape of SATD multi-classification, illustrating the potential of LLMs in this domain. The results should be analyzed with the objective of identifying scenarios where LLMs excel—demonstrating how SALA, by leveraging the contextual understanding and generalization capabilities of LLMs, achieves significant results across diverse SATD categories and projects.

**Table 11**
Comparison of Evaluation Features of different approaches [P14], [P27], [P23], [P24], and [P22].

| Feature | DebtHunter [P14] | SCGRU [P27] | PILOT [P23] | Yu et al. [P24] | Qu et al. [P22] |
|---|---|---|---|---|---|
| Categories Evaluated | All | All | All | DESIGN, REQUIREMENT, DEFECT | All |
| Evaluation Methodology | stratified 10-fold cross-validation | Fixed split 10/1 | 10-fold cross-validation | 10-fold cross-validation | Extra project cross-validation |
| Metrics Reported | Precision, Recall, $F_1$-score | Precision, Recall, $F_1$-score | Precision, Recall, $F_1$-score | Precision, Recall, $F_1$-score | $F_1$-score |
| Pre-filtering | No | No | Yes | Yes | No |
| Result aggregation | Category | Category and Project | Category | Category and Project | by category and project |

**Table 12**
Comparison of SALA with PILOT [P23] and DebtHunter [P14].

| | PILOT [P23] | | | DebtHunter [P14] | | | SALA | | |
|---|---|---|---|---|---|---|---|---|---|
| | P | R | $F_1$ | P | R | $F_1$ | P | R | $F_1$ |
| DEFECT | 0.629 | 0.424 | 0.507 | 0.659 | 0.377 | 0.478 | **0.999** | **0.660** | **0.702** |
| DESIGN | 0.799 | 0.896 | 0.845 | 0.794 | **0.927** | 0.855 | **1.000** | 0.914 | **0.887** |
| DOCUMENTATION | **1.000** | 0.241 | 0.388 | **1.000** | 0.204 | 0.339 | **1.000** | **0.833** | **0.625** |
| REQUIREMENT | 0.596 | 0.532 | 0.562 | 0.634 | 0.515 | 0.568 | **0.999** | **0.553** | **0.634** |
| TEST | 0.914 | 0.376 | 0.533 | 0.909 | 0.235 | 0.373 | **1.000** | **0.850** | **0.810** |
| **Accuracy** | | 0.754 | | | 0.740 | | | 0.782 | |

These tables should be interpreted with the aim of highlighting the circumstances under which LLMs excel, i.e., demonstrating how SALA, leveraging the contextual understanding and generalization capabilities of LLMs, achieves significant results across diverse SATD categories and projects. Tables 13, 14, and 15 break down the evaluation by project. To aid in understanding the tables, we clarify the notation used as follows: a "–" indicates that no instances of the corresponding SATD category were present in the testing set, while "NA" signifies that the respective approach did not report results for that category (i.e., Yu et al. [P24] did not evaluate the DOCUMENTATION and TEST categories). These results highlight that SALA consistently outperforms existing approaches in categories with complex, context-dependent text, such as DESIGN and DOCUMENTATION. With their strong contextual understanding, this suggests that LLMs can better capture the nuances of such SATD categories. In contrast, categories like REQUIREMENT and TEST exhibit high variance in recall across projects, where traditional baselines sometimes achieve a better balance between precision and recall. This can be attributed to the nature of the textual patterns within these categories — potentially more structured or domain-specific — which rule-based or deep learning models trained on domain-specific embeddings might capture more effectively than general-purpose LLMs.

It is important to note that Qu et al. [P22] only presented the performance of their approach using the $F_1$-score. Consequently, Table 15 includes the results from [P22]. A key takeaway message from Table 15 (F1-measure) is that SALA's advantage over baselines is more pronounced in projects with diverse SATD examples, indicating that LLMs generalize well across heterogeneous datasets. However, in projects with limited training data or where SATD comments are highly structured, some baseline approaches still yield competitive or better recall.

The results indicate that SALA's performance advantages stem from its ability to leverage contextual understanding and generalization capabilities inherent in LLMs. This advantage is particularly evident in the following scenarios:

- Handling Imbalanced Datasets: In categories with fewer examples, such as DOCUMENTATION, traditional approaches like DebtHunter [P14] struggle with low recall. SALA, however, benefits from pre-trained knowledge, leading to a better balance between recall and precision.
- Categories with High Variability: By the DESIGN and REQUIREMENT categories, SALA maintains stable performance, indicating robustness to category-specific variations in SATD annotations.
- Limited Training Data: When training data is sparse (e.g., TEST category), SALA's pre-trained LLM foundation allows it to generalize better than baselines relying solely on traditional machine learning techniques.

The results highlight that LLMs, such as SALA, present a promising avenue for multi-class SATD classification, particularly in handling diverse, unstructured, and complex textual descriptions. Their ability to generalize across different projects and SATD categories makes them a valuable tool for SATD detection. However, integrating hybrid approaches — such as combining LLMs with domain-specific embeddings or fine-tuned classifiers — could further enhance performance, particularly by improving recall while preserving the high precision that SALA demonstrates.

**Answer to RQ$_3$.** SALA achieves a good performance in multi-class SATD classification, particularly excelling in precision and handling imbalanced datasets due to its LLM-driven contextual understanding. Although direct comparisons are challenging due to methodological differences, the results suggest that SALA outperforms existing baselines, especially in terms of recall across diverse SATD categories.

### 3.7. Threats to validity

This section describes the validity threats to our evaluation following the guidelines proposed by Easterbrook et al. [15].

- **Internal validity.** This concerns the factors internal to our evaluation that can impact the findings. The selection of hyperparameters such as temperature, the number of tokens might influence the final experimental outcomes. To mitigate such an issue, we experimented with different settings to obtain a stable measurement of performance. Internal validity also involves the robustness of the study design, particularly in establishing that the outcomes are a direct derivation of the collected data and misapplication of statistical analyses [15]. In the evaluation, we adopted a well-known dataset manually curated and classified by existing research [8]. The quality of the curated data depends very much on the evaluators' expertise. We anticipate that comments might be misclassified and unbalanced, thus negatively impacting the

**Table 13**
Precision for multiclass classification of SALA with SCGRU [P27] and Yu et al. [P24].

| Project | DEFECT | | | DESIGN | | | DOCUMENTATION | | | REQUIREMENT | | | TEST | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | SALA | [P27] | [P24] | SALA | [P27] | [P24] | SALA | [P27] | [P24] | SALA | [P27] | [P24] | SALA | [P27] | [P24] |
| ant | 1.00 | 0.43 | 0.70 | 1.00 | 0.59 | 0.86 | – | – | NA | 0.00 | 0.61 | 0.57 | 0.00 | 0.29 | NA |
| argouml | 1.00 | 0.46 | 0.49 | 1.00 | 0.77 | 0.70 | – | 0.52 | NA | 1.00 | 0.66 | 0.81 | 1.00 | 0.88 | NA |
| columba | 1.00 | 0.75 | 0.53 | 1.00 | 0.77 | 0.85 | 1.00 | 0 | NA | 1.00 | 0.83 | 0.55 | 1.00 | 1.00 | NA |
| emf | 0.00 | 1.0 | 0.83 | 1.00 | 0.82 | 0.86 | – | – | NA | 0.00 | 1.00 | 0.80 | – | 1.00 | NA |
| hibernate | 1.00 | 0.85 | 0.45 | 1.00 | 0.84 | 0.82 | – | 0 | NA | 1.00 | 0.86 | 0.45 | – | – | NA |
| jEdit | 1.00 | 0.58 | 0.41 | 1.00 | 0,71 | 0.86 | – | – | NA | 1.00 | 0.417 | 0.52 | 0.00 | 1.00 | NA |
| jfreechart | 1.00 | 0.75 | 0.34 | 1.00 | 0.77 | 0.96 | – | – | NA | 1.00 | 0.90 | 0.22 | – | 0 | NA |
| jmeter | 0.00 | 0.67 | 0.19 | 1.00 | 0.86 | 0.92 | – | 1.00 | NA | 1.00 | 0.57 | 0.19 | 1.00 | 1.00 | NA |
| jruby | 1.00 | 0.9 | 0.65 | 1.00 | 0.80 | 0.69 | 1.00 | 1.00 | NA | 1.00 | 0.77 | 0.73 | 1.00 | 0.67 | NA |
| SQuirrel | 1.00 | 0.53 | 0.65 | 1.00 | 0.85 | 0.85 | – | 1.00 | NA | 1.00 | 0.87 | 0.65 | – | 0 | NA |

**Table 14**
Recall for multiclass classification of SALA with SCGRU [P27] and Yu et al. [P24].

| Project | DEFECT | | | DESIGN | | | DOCUMENTATION | | | REQUIREMENT | | | TEST | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | SALA | [P27] | [P24] | SALA | [P27] | [P24] | SALA | [P27] | [P24] | SALA | [P27] | [P24] | SALA | [P27] | [P24] |
| ant | 1.00 | 0.46 | 0.43 | 1.00 | 0.63 | 0.95 | – | – | NA | 0.00 | 0.61 | 0.38 | 0 | 0.4 | NA |
| argouml | 0.53 | 0.67 | 0.21 | 0.76 | 0.94 | 0.94 | – | 0.33 | NA | 0.91 | 0.90 | 0.42 | 1.00 | 0.68 | NA |
| columba | 1.00 | 0.69 | 0.52 | 1.00 | 0.82 | 0.74 | 1.00 | 0.00 | NA | 1.00 | 0.93 | 0.69 | 1.00 | 0.67 | NA |
| emf | 0.00 | 0.25 | 0.67 | 0.96 | 0.47 | 0.97 | – | – | NA | 0 | 0.44 | 0.36 | – | 0.5 | NA |
| hibernate | 0.53 | 0.67 | 0.29 | 0.87 | 0.79 | 0.85 | – | 0.00 | NA | 0.25 | 0.87 | 0.48 | – | – | NA |
| jEdit | 0.83 | 0.16 | 0.51 | 0.92 | 0.53 | 0.88 | – | – | NA | 1.00 | 0.37 | 0.37 | 0 | 0.67 | NA |
| jfreechart | 1.00 | 0.67 | 0.29 | 0.94 | 0.34 | 0.82 | – | – | NA | 0.50 | 0.6 | 0.65 | – | 0 | NA |
| jmeter | 0.0 | 0.54 | 0.37 | 0.79 | 0.75 | 0.82 | – | 1.00 | NA | 0.66 | 0.57 | 0.31 | 1.0 | 0.83 | NA |
| jruby | 0.99 | 0.58 | 0.27 | 0.85 | 0.61 | 0.89 | 1.00 | 0.50 | NA | 0.54 | 0.64 | 0.55 | 1.0 | 0.33 | NA |
| SQuirrel | 1.00 | 0.37 | 0.57 | 0.83 | 0.54 | 0.90 | – | 0.05 | NA | 0.50 | 0.87 | 0.51 | – | 0 | NA |

**Table 15**
$F_1$ score for multiclass classification of SALA with SCGRU [P27], Yu et al. [P24], and Qu et al. [P22].

| Project | DEFECT | | | | DESIGN | | | | DOCUMENTATION | | | | REQUIREMENT | | | | TEST | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | SALA | [P27] | [P24] | [P22] | SALA | [P27] | [P24] | [P22] | SALA | [P27] | [P24] | [P22] | SALA | [P27] | [P24] | [P22] | SALA | [P27] | [P24] | [P22] |
| ant | 1.00 | 0.44 | 0.53 | 0.75 | 1.00 | 0.61 | 0.90 | 0.55 | – | – | NA | 0.00 | 0.00 | 0.61 | 0.46 | 0.78 | 0.00 | 0.33 | NA | 0.84 |
| argouml | 0.69 | 0.54 | 0.29 | 0.41 | 0.86 | 0.85 | 0.80 | 0.66 | – | 0.41 | NA | 0.69 | 0.95 | 0.76 | 0.55 | 0.63 | 1.00 | 0.77 | NA | 0.66 |
| columba | 1.00 | 0.72 | 0.51 | 0.72 | 1.00 | 0.80 | 0.74 | 0.65 | 1.00 | 0 | NA | 0.68 | 1.00 | 0.88 | 0.57 | 0.56 | 1.00 | 0.8 | NA | 0.88 |
| emf | 0.00 | 0.4 | 0.73 | 0.92 | 0.98 | 0.60 | 0.91 | 0.66 | – | – | NA | 0.00 | 0.00 | 0.61 | 0.49 | 0.91 | – | 0.67 | NA | 0.99 |
| hibernate | 0.70 | 0.75 | 0.35 | 0.42 | 0.93 | 0.82 | 0.84 | 0.66 | – | 1.0 | NA | 0.96 | 0.40 | 0.87 | 0.46 | 0.52 | – | – | NA | 0.00 |
| jEdit | 0.90 | 0.25 | 0.45 | 0.99 | 0.96 | 0.61 | 0.87 | 0.98 | – | – | NA | 0.00 | 1.00 | 0.38 | 0.41 | 0.99 | 0.00 | 0.80 | NA | 0.99 |
| jfreechart | 1.00 | 0.71 | 0.30 | 0.67 | 0.97 | 0.47 | 0.88 | 0.54 | – | – | NA | 0.00 | 0.67 | 0.72 | 0.33 | 0.49 | – | 0 | NA | 0.95 |
| jmeter | 0.00 | 0.60 | 0.25 | 0.47 | 0.88 | 0.80 | 0.86 | 0.78 | – | 1.0 | NA | 0.95 | 0.80 | 0.57 | 0.23 | 0.45 | 1.00 | 0.91 | NA | 0.75 |
| jruby | 0.99 | 0.71 | 0.37 | 0.19 | 0.92 | 0.70 | 0.77 | 0.80 | 1.00 | 0.67 | NA | 0.90 | 0.70 | 0.70 | 0.63 | 0.50 | 1.00 | 0.44 | NA | 0.75 |
| SQuirrel | 1.00 | 0.44 | 0.60 | 0.80 | 0.91 | 0.66 | 0.88 | 0.67 | – | 0.67 | NA | 0.99 | 0.67 | 0.84 | 0.57 | 0.74 | – | 0 | NA | 0.99 |

overall prediction performance. To avoid such pitfalls, evaluating the systems with a larger dataset is necessary. Regarding the used metrics, we considered the standard ones used in evaluating the ML-based approaches.

- **External validity.** This is related to the generalizability of the findings beyond the scope of this study. The conclusions drawn from this paper may be applicable only to the considered datasets, i.e., SATD comments collected from Java projects. Future work should be conducted to study the performance of SALA on additional datasets curated from other programming languages. Moreover, some of the baseline results reported in this study are adopted from prior publications due to the unavailability of replication packages. While we strived to align our evaluation setting with those of the original studies, e.g., using the same dataset splits and focusing on cross-project evaluations when applicable, differences in experimental environments, data preprocessing, or implementation details could affect the comparability of results. We mitigated this threat by including a fair comparison with DebtHunter [P14] and PILOT [P23], for which we replicated experiments using the same dataset and evaluation procedure as for SALA. Nevertheless, comparisons involving GNNSI [P19], SATD-LN [P25], Aiken et al. [P26], and GGSATD [P21] should be interpreted with caution, as they rely on published metrics that may have been produced under slightly different conditions.

## 4. Related work

By answering **RQ**$_1$ we already identified state-of-the-art studies on the automatic binary and multi-class classifications of SATD using source comments within open-source Java projects. Therefore, in this section we focus on reviewing some of the most significant studies related to the problem of identifying and classifying SATD in other source artifacts as well as other languages.

### 4.1. Detection of SATD in other source artifacts

Recently, an approach has been proposed to identify SATD using issues extracted from seven open-source projects automatically [16]. First, the authors created a dataset by collecting and manually analyzing 4,200 issues. Then, various ML techniques were employed to perform the classification. The approach was evaluated using the previously defined dataset and the results showed that Text CNN with default settings using random word embedding gains the highest F1-score (0.597). Moreover, the authors experimented with several Text CNN configurations, i.e., handling imbalanced data, refining word embeddings, and CNN hyperparameters, to identify the best configuration. The results indicated that the F1-score improved by 14.9% from 0.597 to 0.686 for the Text CNN with default settings. The authors leveraged the knowledge of existing SATD datasets to attempt to improve the accuracy of Text CNN. The results showed that the F1-score

slightly increases from 0.686 to 0.691 using the Jira issue sentiment dataset [17].

Three studies [18–20] investigated SATD[8] to identify and repay SATD in issues. In particular, Bellomo et al. conducted the study using the issues coming from four projects, two open-source and two government IT projects [18]. They extracted 1,264 issues and manually labeled 109 issues as SATD issues. The results indicated that developers are aware of concepts related to TD, and they embed them within issues to communicate TD. Dai and Kruchten [19] extracted 8,149 issues of a commercial software product. The issues were written in Mandarin since the related product comes from China. The authors analyzed and manually tagged, obtaining 331 SATD issues. Then a Naive Bayes classifier was applied to identify issues as SATD or non-SATD automatically. Moreover, they derived text patterns from recognizing TD. The results indicate the 20 most informative features strongly interconnected to TD. Xavier et al. [20] created an initial dataset containing 286 SATD issues using five open-source projects. Through three research questions, the authors studied *(i)* which types of SATD are paid in the found issues; *(ii)* why developers introduce; and *(iii)* pay SATD in the found issues. Concerning the first RQ, the results showed that the DESIGN category is the most frequent one, with almost 60%, followed by UI (10%), TESTS (9%), and PERFORMANCE (8%). The results of the other two RQs indicated that SATD was introduced at the first stages for about 45% of the surveyed developers. Moreover, 65.5% of the respondents declared that SATD payment is used to reduce TD interest, followed by *To clean code* (27.6%). Another contribution of the paper was developing the AdmiTD tool to help developers document SATD within source comments by automatically generating issues.

Xiao et al. [21] performed an empirical study to identify elements of the project object model (POM) in maven-based systems affected by the SATD through the related comments. The results showed that "plugin" and "external dependencies" configurations are the most frequently occurring pom's elements with, respectively, 49% and 32% of the related SATD comments left by developers. Another contribution was how to recognize the most frequent reasons and purposes for developers to leave SATD comments usually. The results depicted that developers wrote SATD comments for the "limitation" reason with 50%. Moreover, "document for later fix" and "document workaround" are the most frequently occurring purposes, with 39% and 22%, respectively. The authors conducted another study on automatically classifying the SATDs comments in build systems for the reasons and purposes previously identified, using different ML techniques, i.e., auto-sklearn, Naive Bayes, Support Vector Machine, and k-Nearest Neighbor. The auto-sklearn classifier obtained the best value for the F1-score metrics for both reasons and purposes.

The studies mentioned above use different issues and build artifacts, i.e., pom, to detect SATD automatically. All approaches extract comments from these sources to create a dataset by manually labeling them. Then, they use ML techniques to identify the binary or multi-class SATDs. Since they use comments, our proposed techniques can be deployed to detect technical debt in the aforementioned artifacts. Regardless of the sources, once SATD comments are extracted, various NLP and embedding techniques will be used to parse the input data, and PILOT is able to run the training and deployment phases, as shown in this paper.

Li et al. [22] proposed and evaluated an approach for automated SATD identification that integrates four sources: source code comments, commit messages, pull requests, and issue tracking systems. They characterized SATD in 103 Apache open-source projects by analyzing the four sources, and observed that SATD is spread among all sources. Moreover, issues and pull requests are the two most similar sources regarding the number of shared SATD keywords, followed by commit messages, and then by code comments. The proposed SATD identification is trained and tested, and is based on a convolutional neural network and leverages the multitask learning technique. They created two available SATD datasets, one containing 5,000 commit messages and 5,000 pull request sections from the 103 open-source projects manually tagged as non-SATD or SATD (including the types of SATD) and another one containing 23.7M code comments, 1.3M commit messages, 0.3M pull requests, and 0.6M issues from the same 103 Apache open-source projects.

### 4.2. Detection of SATD in other programming languages

An initial attempt was made to detect technical debt in R packages [23]. The authors applied several techniques, i.e., traditional Machine Learning (ML), deep learning, and Pre-Trained Language Models (PTMs), to identify and classify SATD from a dataset containing 164K R source code comments with 4,962 instances classified into 12 TD types and 159,299 Non-SATD [24]. The results demonstrated that PTMs (RoBERTa model) exceed the other considered ML models for categorizing SATD R source comments.

Liu et al. [25] accomplished a study to identify and classify SATD within deep learning frameworks manually. The authors created a dataset containing 234,260 source comments from TensorFlow, Caffe, PyTorch, MXNet, CNTK, and DL4J. All the frameworks are written in Python and C++ except Dl4J written in Java. Then they manually classified comments using a well-defined process. The results showed that 2.93%, i.e., 7,159, of all comments are SATD and TensorFlow contains the most considerable amount of SATD (3,775 instances), whereas Keras has only 54 SATD instances. Another result indicated that design debt is the most prevalent, with 24.07% (in Keras) to 65.27% (in CNTK), followed by requirement debt (7.09%–31.48%) and algorithm debt (5.62%–20.67%).

The detection of SATD types in ML software has been investigated in recent work [26]. In particular, the authors extracted 68,820 source comments from 2,641 GitHub projects related to ML tools and applications written in Python. Then they sampled 856 source comments and manually classified them. The results pointed out that the most frequent SATD type is requirement debt, with 40.68%, followed by code debt (26.24%), test debt (10.65%), defect debt (10.39%), and design debt (10.14%). Moreover, the authors created an ad-hoc taxonomy of ML SATDs, named "ML SATD Groups" consisting of 23 ML SATD Types, where 13 come from pre-existing TD in ML software types. The remaining five are new types, and the other five are specializations of the pre-existing Configuration debt.

Unlike the studies discussed in Section 4.1, the previous approaches attempt to identify SATD automatically using comments from different languages, such as R and Python. Therefore, we could investigate whether our models trained for detecting SATD can be re-used to identify SATDs in these languages.

## 5. Conclusion

This study included a literature review to identify representative approaches for SATD classification, covering both binary and multi-class settings. The review considered recent work published in leading software engineering venues and led to the selection of baselines used in our empirical evaluation. The analysis shows that both binary and multi-class classification have been addressed in the literature, although with varying methodological choices and levels of replication support. To this end, we designed and implemented SALA, a dual-purpose framework to detect SATD from comments, applying different NLP and ML techniques. To assess the performance of SALA, we conducted an empirical evaluation against existing baselines for both classification tasks. The evaluation shows that the fine-tuned LLaMA model used in SALA achieves competitive results in identifying SATD comments in the binary setting and performs reliably in categorizing

[8] Although the authors utilized the more general TD term, we used SATD as this was defined in the original study [3].

SATD comments. These findings suggest that LLM-based approaches, such as the one adopted in SALA, can be a viable option for addressing both binary and multi-class SATD classification tasks. Several directions can be explored to extend this work. First, future studies could investigate the generalizability of LLM-based SATD classification across different programming languages and domains beyond those included in the Maldonado dataset. Second, while SALA focuses on comment-level classification, integrating contextual information from surrounding code or issue trackers may further improve performance, particularly in multi-class settings. Third, a promising direction involves exploring continual or active learning strategies to adapt the model over time as projects evolve.

## CRediT authorship contribution statement

**Francesca Arcelli Fontana:** Writing – review & editing, Writing – original draft, Validation, Supervision, Methodology. **Juri Di Rocco:** Writing – review & editing, Writing – original draft, Visualization, Software, Data curation, Conceptualization. **Davide Di Ruscio:** Writing – review & editing, Writing – original draft, Supervision, Conceptualization. **Amleto Di Salle:** Writing – review & editing, Writing – original draft, Validation, Investigation, Data curation. **Phuong T. Nguyen:** Writing – review & editing, Writing – original draft, Supervision, Software, Methodology, Conceptualization.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Acknowledgments

This work has been partially funded by (a) by the MUR (Italy) Department of Excellence 2023–2027 for GSSI; (b) by the European Union - NextGenerationEU under the Italian Ministry of University and Research (MUR) National Innovation Ecosystem grant ECS00000041 - VITALITY – CUP: D13C21000430001; (c) by the European Union - NextGenerationEU under the Italian Ministry of University and Research (MUR) National Innovation Ecosystem, grant PE0000020 – CHANGES – CUP: D53C22002560006; (d) by the European HORIZON-KDT-JU research project MATISSE "Model-based engineering of Digital Twins for early verification and validation of Industrial Systems", HORIZON-KDT-JU-2023-2-RIA, Proposal number: 101140216-2, KDT232RIA_00017; (e) the MOSAICO project (Management, Orchestration and Supervision of AI-agent COmmunities for reliable AI in software engineering) that has received funding from the European Union under the Horizon Research and Innovation Action (Grant Agreement No. 101189664); and (f) "PRIN 2022" project TRex-SE: *"Trustworthy Recommenders for Software Engineers"*, grant n. 2022LKJWHC.

## Appendix. The selected papers

[P1] F. Zampetti, C. Noiseux, G. Antoniol, F. Khomh, and M. Di Penta. Recommending when design technical debt should be self-admitted. In 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 216–226, 2017. doi: 10.1109/ICSME.2017.44.

[P2] E. da S. Maldonado, E. Shihab, and N. Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. IEEE Transactions on Software Engineering, 2017. doi: 10.1109/tse.2017.2654244.

[P3] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li. Identifying self-admitted technical debt in open source projects using text mining. Empirical Software Engineering, 23(1):418–451, 2018.

[P4] S. Wattanakriengkrai, N. Srisermphoak, S. Sintoplertchaikul, M. Choetkiertikul, C. Ragkhitwetsagul, T. Sunetnanta, H. Hata, and K. Matsumoto. Automatic classifying self-admitted technical debt using n-gram idf. In 2019 26th Asia-Pacific Software Engineering Conference (APSEC), pages 316–322, 2019. doi: 10.1109/APSEC48747.2019.00050.

[P5] M. Yan, X. Xia, E. Shihab, D. Lo, J. Yin, and X. Yang. Automating change-level self-admitted technical debt determination. IEEE Transactions on Software Engineering, 45(12):1211–1229, 2019. doi: 10.1109/TSE.2018.2831232.

[P6] J. Flisar and V. Podgorelec. Identification of self-admitted technical debt using enhanced feature selection based on word embedding. IEEE Access, 7:106475–106494, 2019. doi: 10.1109/ACCESS. 2019.2933318.

[P7] X. Ren, Z. Xing, X. Xia, D. Lo, X. Wang, and J. Grundy. Neural network-based detection of self-admitted technical debt: From performance to explainability. ACM Trans. Softw. Eng. Methodol., 28(3), jul 2019. ISSN 1049-331X. doi: 10.1145/3324916. URL https://doi.org/10.1145/3324916.

[P8] R. Maipradit, B. Lin, C. Nagy, G. Bavota, M. Lanza, H. Hata, and K. Matsumoto. Automated identification of on-hold self-admitted technical debt. In 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), pages 54–64, 2020. doi: 10.1109/SCAM51674. 2020.00011.

[P9] X. Wang, J. Liu, L. Li, X. Chen, X. Liu, and H. Wu. Detecting and explaining self-admitted technical debts with attention-based neural networks. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, ASE '20, page 871–882, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367684. doi: 10.1145/3324884.3416583. URL https://doi.org/10.1145/3324884.3416583.

[P10] Z. Yu, F. M. Fahid, H. Tu, and T. Menzies. Identifying self-admitted technical debts with jitterbug: A two-step approach. IEEE Transactions on Software Engineering, 2020.

[P11] R. Meneses Santos, I. Santos, M. Júnior, and M. Mendonça. Long term-short memory neural networks and word2vec for self-admitted technical debt detection. pages 157–165, 05 2020.

[P12] L. Rantala and M. Mäntylä. Prevalence, Contents and Automatic Detection of KL-SATD, 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2020, pp. 385–388, doi: 10.1109/SEAA51224.2020.00069.

[P13] M. Sridharan, M. Mantyla, L. Rantala, and M. Claes. Data balancing improves self-admitted technical debt detection. In 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), pages 358–368. IEEE, 2021.

[P14] I. Sala, A. Tommasel, and F. Arcelli Fontana. Debthunter: A machine learning-based approach for detecting self-admitted technical debt. In Evaluation and Assessment in Software Engineering, EASE 2021, page 278–283, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450390538. doi: 10.1145/3463274.3464455. URL https://doi.org/10.1145/3463274.3464455.

[P15] K. Zhu, M. Yin, and Y. Li. Detecting and classifying self-admitted of technical debt with cnn-bilstm. In Journal of Physics: Conference Series, volume 1955, page 012102. IOP Publishing, 2021.

[P16] Z. Guo, S. Liu, J. Liu, Y. Li, L. Chen, H. Lu, and Y. Zhou. How far have we progressed in identifying self-admitted technical debts? a comprehensive empirical study. 30(4), July 2021. ISSN 1049-331X. doi:10.1145/3447247. URL https://doi.org/10.1145/3447247.

[P17] X. Chen, D. Yu, X. Fan, L. Wang, and J. Chen. Multiclass classification for self-admitted technical debt based on xgboost. IEEE Transactions on Reliability, pages 1–16, 2021. doi:10.1109/TR.2021. 3087864.

[P18] D. Yu, L. Wang, X. Chen, and J. Chen. Using bilstm with attention mechanism to automatically detect self-admitted technical debt. Frontiers of Computer Science, 15(4):1–12, 2021.

[P19] Hui Li, Yang Qu, Yong Liu, Rong Chen, Jun Ai, and Shikai Guo. Self-admitted technical debt detection by learning its comprehensive semantics via graph neural networks. Software: Practice and Experience, 2022. URL https://doi.org/10.1002/spe.3117

[P20] Q. H. Tu and T. Menzies. Debtfree: minimizing labeling cost in self-admitted technical debt identification using semi-supervised learning. Empirical Software Engineering, 27, 07 2022. doi:10.1007/s10664-022-10121-w.

[P21] J. Yu, K. Zhao, J. Liu, X. Liu, Z. Xu, and X. Wang. Exploiting gated graph neural network for detecting and explaining self-admitted technical debts. Journal of Systems and Software, 187:111219, 2022. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2022.111219.

[P22] Yubin Qu, Tie Bao, Meng Yuan, Long Li. Deep Learning-Based Self-Admitted Technical Debt Detection Empirical Research. Journal of Internet Technology, vol. 24, no. 4, pp. 975–987, Jul. 2023.

[P23] A. Di Salle, A. Rota, P. T. Nguyen, D. Di Ruscio, F. Arcelli Fontana, and I. Sala. PILOT: Synergy between text processing and neural networks to detect self-admitted technical debt. In Proceedings of the 2022 IEEE/ACM International Conference on Technical Debt (TechDebt2022), 2022. doi: 10.1145/3524843.3528093

[P24] J. Yu, X. Zhou, X. Liu, J. Liu, Z. Xie, K. Zhao, Detecting multi-type self-admitted technical debt with generative adversarial network-based neural networks. Journal of Information and Software Technology, 2023. doi: https://doi.org/10.1016/j.infsof.2023.107190

[P25] A. Gong, F. Fukumoto, P. Muangkammuen, L. Jiyi, D. Yu, Identifying Self-admitted Technical Debt with Context-Based Ladder Network, International Conference on Neural Information Processing, 2023. doi: https://doi.org/10.1007/978-981-99-8184-7_7

[P26] W. Aiken, P.K. Mvula, P. Branco, G.V. Jourdan, M. Sabetzadeh, H. Viktor, Measuring Improvement of F1-Scores in Detection of Self-Admitted Technical Debt, in 2023 ACM/IEEE International Conference on Technical Debt, TechDebt 2023. doi: https://doi.org/10.1109/TechDebt59074.2023.00011

[P27] K. Zhu, M. Yin, D. Zhu, X. Zhang, C. Gao, J. Jiang, SCGRU: A general approach for identifying multiple classes of self-admitted technical debt with text generation oversampling, Journal of Systems and Software, 2023. doi: https://doi.org/10.1016/j.jss.2022.111514

## Data availability

The authors do not have permission to share data.

## References

[1] I. Sala, A. Tommasel, F. Arcelli Fontana, DebtHunter: A machine learning-based approach for detecting self-admitted technical debt, in: Evaluation and Assessment in Software Engineering, in: EASE 2021, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450390538, 2021, pp. 278–283, http://dx.doi.org/10.1145/3463274.3464455, URL https://doi.org/10.1145/3463274.3464455.

[2] J.C. Carver, X. Larrucea, A. Serebrenik, M. Staron, Technical debt problems and concerns, IEEE Softw. 39 (3) (2022) 116–119, http://dx.doi.org/10.1109/MS.2021.3133734.

[3] A. Potdar, E. Shihab, 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, IEEE Computer Society, 2014, pp. 91–100, http://dx.doi.org/10.1109/ICSME.2014.31, URL https://doi.org/10.1109/ICSME.2014.31.

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

[5] G. Sierra, E. Shihab, Y. Kamei, A survey of self-admitted technical debt, J. Syst. Softw. (ISSN: 0164-1212) 152 (2019) 70–82, http://dx.doi.org/10.1016/j.jss.2019.02.056, URL https://www.sciencedirect.com/science/article/pii/S0164121219300457.

[6] A. Di Salle, A. Rota, P.T. Nguyen, D. Di Ruscio, F. Arcelli Fontana, I. Sala, PILOT: Synergy between text processing and neural networks to detect self-admitted technical debt, in: Proceedings of the 2022 IEEE/ACM International Conference on Technical Debt, TechDebt2022, 2022, http://dx.doi.org/10.1145/3524843.3528093.

[7] Z. Guo, S. Liu, J. Liu, Y. Li, L. Chen, H. Lu, Y. Zhou, How far have we progressed in identifying self-admitted technical debts? A comprehensive empirical study, ACM Trans. Softw. Eng. Methodol. (ISSN: 1049-331X) 30 (4) (2021) http://dx.doi.org/10.1145/3447247, URL https://doi.org/10.1145/3447247.

[8] E.d.S. Maldonado, E. Shihab, N. Tsantalis, Using natural language processing to automatically detect self-admitted technical debt, IEEE Trans. Softw. Eng. 43 (11) (2017) 1044–1062, http://dx.doi.org/10.1109/TSE.2017.2654244.

[9] B. Kitchenham, S. Charters, Guidelines for performing systematic literature reviews in software engineering, 2007.

[10] H. Zhang, M.A. Babar, P. Tell, Identifying relevant studies in software engineering, Inf. Softw. Technol. 53 (6) (2011) 625–637, http://dx.doi.org/10.1016/j.infsof.2010.12.010, URL https://doi.org/10.1016/j.infsof.2010.12.010.

[11] P.T. Nguyen, C. Di Sipio, J. Di Rocco, M. Di Penta, D. Di Ruscio, Adversarial attacks to API recommender systems: Time to wake up and smell the coffee? in: 2021 36th IEEE/ACM International Conference on Automated Software Engineering, ASE, 2021, pp. 253–265, http://dx.doi.org/10.1109/ASE51524.2021.9678946.

[12] L. Henrickson, A. Meroño-Peñuela, Prompting meaning: a hermeneutic approach to optimising prompt engineering with ChatGPT, AI Soc. (ISSN: 0951-5666) (2023) Publisher Copyright: © 2023, The Author(s). http://dx.doi.org/10.1007/s00146-023-01752-8.

[13] T. Taulli, Prompt engineering, in: ChatGPT and Bard for Business Automation: Achieving AI-Driven Growth, A Press, Berkeley, CA, ISBN: 978-1-4842-9852-7, 2023, pp. 51–64, http://dx.doi.org/10.1007/978-1-4842-9852-7_4, URL https://doi.org/10.1007/978-1-4842-9852-7%5F4.

[14] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, et al., Prompt tuning in code intelligence: An experimental evaluation, IEEE Trans. Softw. Eng. (ISSN: 1939-3520) (01) (2023) 1–17, http://dx.doi.org/10.1109/TSE.2023.3313881.

[15] S. Easterbrook, J. Singer, M.-A. Storey, D. Damian, Selecting empirical methods for software engineering research, in: F. Shull, J. Singer, D.I.K. Sjøberg (Eds.), Guide to Advanced Empirical Software Engineering, Springer London, London, ISBN: 978-1-84800-044-5, 2008, pp. 285–311, http://dx.doi.org/10.1007/978-1-84800-044-5_11, URL https://doi.org/10.1007/978-1-84800-044-5_11.

[16] Y. Li, M. Soliman, P. Avgeriou, Identifying self-admitted technical debt in issue tracking systems using machine learning, Empir. Softw. Eng. 27 (6) (2022) 131, http://dx.doi.org/10.1007/s10664-022-10128-3, URL https://doi.org/10.1007/s10664-022-10128-3.

[17] M. Ortu, A. Murgia, G. Destefanis, P. Tourani, R. Tonelli, M. Marchesi, B. Adams, The emotional side of software developers in JIRA, in: M. Kim, R. Robbes, C. Bird (Eds.), Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14–22, 2016, ACM, 2016, pp. 480–483, http://dx.doi.org/10.1145/2901739.2903505, URL https://doi.org/10.1145/2901739.2903505.

[18] S. Bellomo, R.L. Nord, I. Ozkaya, M. Popeck, Got technical debt?: surfacing elusive technical debt in issue trackers, in: M. Kim, R. Robbes, C. Bird (Eds.), Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14–22, 2016, ACM, 2016, pp. 327–338, http://dx.doi.org/10.1145/2901739.2901754, URL https://doi.org/10.1145/2901739.2901754.

[19] K. Dai, P. Kruchten, Detecting technical debt through issue trackers, in: H. Lichter, T. Sunetnanta, T. Anwar (Eds.), Proceedings of the 5th International Workshop on Quantitative Approaches To Software Quality Co-Located with 24th Asia-Pacific Software Engineering Conference (APSEC 2017), Nanjing, Jiangsu, China, December 4, 2017, in: CEUR Workshop Proceedings, vol. 2017, CEUR-WS.org, 2017, pp. 59–65, URL http://ceur-ws.org/Vol-2017/paper10.pdf.

[20] L. Xavier, J.E. Montandon, F. Ferreira, R. Brito, M.T. Valente, On the documentation of self-admitted technical debt in issues, Empir. Softw. Eng. 27 (7) (2022) 163, http://dx.doi.org/10.1007/s10664-022-10203-9, URL https://doi.org/10.1007/s10664-022-10203-9.

[21] T. Xiao, D. Wang, S. McIntosh, H. Hata, R.G. Kula, T. Ishio, K. Matsumoto, Characterizing and mitigating self-admitted technical debt in build systems, IEEE Trans. Softw. Eng. 48 (10) (2022) 4214–4228, http://dx.doi.org/10.1109/TSE.2021.3115772.

[22] Y. Li, M. Soliman, P. Avgeriou, Automatic identification of self-admitted technical debt from four different sources, Empir. Softw. Eng. 28 (3) (2023) 65, http://dx.doi.org/10.1007/S10664-023-10297-9, URL https://doi.org/10.1007/s10664-023-10297-9.

[23] R. Sharma, R. Shahbazi, F.H. Fard, Z. Codabux, M.C. Vidoni, Self-admitted technical debt in R: detection and causes, Autom. Softw. Eng. 29 (2) (2022) 53, http://dx.doi.org/10.1007/s10515-022-00358-6, URL https://doi.org/10.1007/s10515-022-00358-6.

[24] M. Vidoni, Self-admitted technical debt in R packages: An exploratory study, in: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories, MSR, 2021, pp. 179–189, http://dx.doi.org/10.1109/MSR52588.2021.00030.

[25] J. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, S. Li, Is using deep learning frameworks free?: characterizing technical debt in deep learning frameworks, in: G. Rothermel, D. Bae (Eds.), ICSE-SEIS '20: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Society, Seoul, South Korea, 27 June–19 July, 2020, ACM, 2020, pp. 1–10, http://dx.doi.org/10.1145/3377815.3381377, URL https://doi.org/10.1145/3377815.3381377.

[26] D. OBrien, S. Biswas, S. Imtiaz, R. Abdalkareem, E. Shihab, H. Rajan, 23 shades of self-admitted technical debt: An empirical study on machine learning software, in: ESEC/FSE'2022: The 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022.