On the Identification of Self-Admitted Technical Debt with Large Language Models

Pedro Lambert
pedro.lambert@sga.pucminas.br
PUC Minas
Belo Horizonte, MG, Brazil

Lucila Ishitani
lucila@pucminas.br
PUC Minas
Belo Horizonte, MG, Brazil

Laerte Xavier
laertexavier@pucminas.br
PUC Minas
Belo Horizonte, MG, Brazil

ABSTRACT

Self-Admitted Technical Debt (SATD) refers to a common practice in software engineering involving developers explicitly documenting and acknowledging technical debt within their projects. Identifying SATD in various contexts is a key activity for effective technical debt management and resolution. While previous research has focused on natural language processing techniques and specialized models for SATD identification, this study explores the potential of Large Language Models (LLMs) for this task. We compare the performance of three LLMs - Claude 3 Haiku, GPT 3.5 turbo, and Gemini 1.0 pro - against the generalization of the state-of-the-art model designed for SATD identification. Additionally, we investigate the impact of prompt engineering on the performance of LLMs in this context. Our findings reveal that LLMs achieve competitive results compared to the state-of-the-art model. However, when considering the Matthews Correlation Coefficient (MCC), we observe that the LLM performance is less balanced, tending to score lower than the state-of-the-art model across all four confusion matrix categories. Nevertheless, with a well-designed prompt, we conclude that the models' bias can be improved, resulting in a higher MCC score.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; **Software evolution**; *Documentation*; *System administration*; *Software maintenance tools.*

KEYWORDS

Self-Admitted Technical Debt, Large Language Models, Prompt Engineering

ACM Reference Format:

Pedro Lambert, Lucila Ishitani, and Laerte Xavier. 2024. On the Identification of Self-Admitted Technical Debt with Large Language Models. In *Proceedings of Brazilian Symposium on Software Engineering (SBES'24).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

SBES'24, September 30 – October 04, 2024, Curitiba, PR

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In software engineering, the metaphorical concept of Technical Debt (TD) refers to the future costs incurred in software development due to sub-optimal solutions [10]. Such decisions often stem from factors such as time constraints, inadequate software quality, and flawed processes [25]. The identification and categorization of technical debt have been largely studied in the literature. Particularly, most previous papers have led to efforts to explicitly document technical debt instances [14, 30, 32, 42] and to propose techniques for characterizing technical debt based on how and why it accrues [16, 27, 40].

The term Self-Admitted Technical Debt (SATD) gained formal recognition in the software engineering literature in 2014 [32]. Initially, research primarily focused on SATD through code comments, analyzing keyword patterns to identify TD admission [6, 32]. Over time, researchers have observed that developers use different documentation methods, for example, issues and pull requests [23]. While much attention is still directed towards studying SATD in code comments, recent investigations have delved into SATD in issue tracking systems [22–24, 35, 41, 42, 44].

When examining SATD in issues, the initial step normally involves identifying instances of TD admissions. Traditionally, researchers have used issue labels to identify SATD in issues [41]. However, recent studies have shifted towards employing natural language processing (NLP) algorithms for this purpose [22–24, 35]. The initial results suggest the potential to recognize Self-Admitted Technical Debt even in the absence of explicit labels. This paves the way to more flexible solutions for SATD management and enables the exploration of the impact of explicitly acknowledging the debt with a label.

However, current results also highlight a challenge: although NLP-based models, particularly those leveraging transformers [35], have been effective in identifying Self-Admitted Technical Debt, they often experience a decline in accuracy when applied to data from projects different from their training set. The necessity of training the model and conducting fine-tuning on a per-project basis may diminish its attractiveness for widespread adoption.

Concurrently, Large Language Models (LLMs) have gained significant attention in both commercial and research contexts. For instance, ChatGPT reached remarkable user adoption rates [37], making LLMs well-known. Researchers also started to explore LLM applications across various domains, including SATD [29]. Despite this growing interest, LLMs have not yet shown positive results in addressing technical debt, as indicated by previous studies related to TD payment [29]. Nonetheless, observations of LLMs' ability to manage technical debt may become benchmarks over time, assessing their performance in handling complex software engineering

tasks that require a deep understanding of the field's practices and reasoning capabilities.

Additionally, prompt engineering has also emerged in the literature as the process of crafting effective prompts for LLMs to improve its outputs [17]. It involves designing the input prompts to leverage the capabilities of the models and guide it towards producing relevant and accurate responses [18]. In the context of this study, we took advantage of prompt engineering to guide LLMs to effectively identify instances of SATD in issues.

In light of this context, our study aims to explore the performance of LLMs in identifying instances of SATD documented in issues. We believe that LLMs may be used as a more versatile and easier-to-use approach compared to the state-of-the-art models specifically designed for this task. To accomplish that we evaluated three LLMs, and four prompting techniques. In particular, we aim to address the following research questions:

RQ1: How does the performance of LLMs compare to the state-of-the-art models for SATD identification? By analyzing and comparing the effectiveness of the models selected in this study against the best-known model proposed in the literature, we aim to determine whether LLMs can achieve comparable performance in identifying SATD in software projects. With the motivation of being able to leverage these easier-to-use, and versatile tools that do not require training.

RQ2: What is the impact of prompt engineering on the performance of LLMs in the task of SATD identification? By employing four different prompt engineering strategies, we observe their effect on the performance of LLMs in SATD identification. We aim to identify not only the most effective prompt engineering approach for this task, but also, to understand the broader capabilities and limitations of prompt engineering.

Structure of the paper. This study is organized as follows. Section 2 introduces the background knowledge necessary for understanding the most used concepts in this paper. After that, Section 3 outlines the study design adopted to accomplish the research. Lastly, Section 4 and Section 5 presents and delves into the interpretation of this study's findings, with Section 6 concluding the paper.

2 BACKGROUND

This section aims to provide an understanding of the foundational concepts and context crucial to this study. It delves into the evolution and significance of Technical Debt in Section 2.1. We also introduce Large Language Models and Prompt Engineering in the Sections 2.2 and 2.3, respectively.

2.1 Technical Debt

The concept of Technical Debt was first introduced by Cunningham in 1992 [10], and its use quickly spread across both academic and non-academic domains. In the early 2000s, Martin Fowler used the term often through his blog posts [12, 13, 15], illustrating the trade-off between initial productivity gains from disregarding sound design practices and the eventual decrease in productivity in a poorly designed software environment. The academic exploration of Technical Debt began in the 2000s [9], leading to a formal definition that recognized Technical Debt as a collection of design or implementation decisions that are beneficial in the short term but can make future changes more costly [5]. This definition not only established the field of study but also broadened the scope of TD beyond source code, recognizing that concerns related to TD are present throughout the entire software development cycle.

The progress of research around Technical Debt led to the topic of how developers deal with a system's debt and the emergence of Self-Admitted Technical Debt [32] as a field focused on the explicit documentation of TD instances. Early studies on SATD aimed to identify debt through source code comments as a means of documentation [6, 32]. These studies followed a list of commonly used keywords to describe Self-Admitted Technical Debt in code comments and confirmed that this was a common practice in the software industry. With this confirmation, research began to focus on comprehending TD [34], identifying it [22], and understanding how it was introduced [41] and paid [26]. Figure 1 demonstrates an instance of SATD documented in MICROSOFT/VSCODE [31]. In this example, it documents a functionality that works, but due to it being implemented in a sub-optional way it needs refactoring.

Recently, the field then expanded to explore other means of documenting SATD, with recent studies delving into Self-Admitted Technical Debt in issues [22, 24, 29, 35]. Some studies have also aimed to understand where to document each type of TD and have investigated the interplay between different means of documentation [23, 43].

2.2 Large Language Models

Large Language Models (LLMs) are systems capable of processing natural language to generate human-like responses. The term *Large* refers to the billions of tokens used to train these models. The success of ChatGPT, a commercially available LLM, has drawn attention to the area. These models have continued to grow in size, and unexpected abilities have begun to emerge [38], enhancing the perception of the models' intelligence, performance, and viability.

This development has marked a "paradigm shift" [11] in Natural Language Processing (NLP) systems, transitioning from being primarily built for specific tasks to being built on a LLM base or "foundational" model. Instead of each task having a statistically optimized model, such as separate systems for SATD identification and categorization, LLMs have paved the way for language-related tasks to be built on top of the same underlying deep learning model, optimized through fine-tuning, prompt engineering, or other approaches that enhance the model's understanding of the task.

Moreover, the largest publicly available models have been fine-tuned to provide the most human-like answers possible. With the vast amount of data they possess, they can respond accurately to a wide range of topics. This ability to generalize and adapt to different contexts and areas is evident in scientific publications spanning various fields, from psychiatry [21] to finance [20]. Fine-tuned and easily accessible models like OpenAI's ChatGPT [2], Google's Gemini [19] and Anthropic's Claude 3 models [4], have, like many others, enabled automatizing tasks that would otherwise require human evaluation, including classification tasks [7].

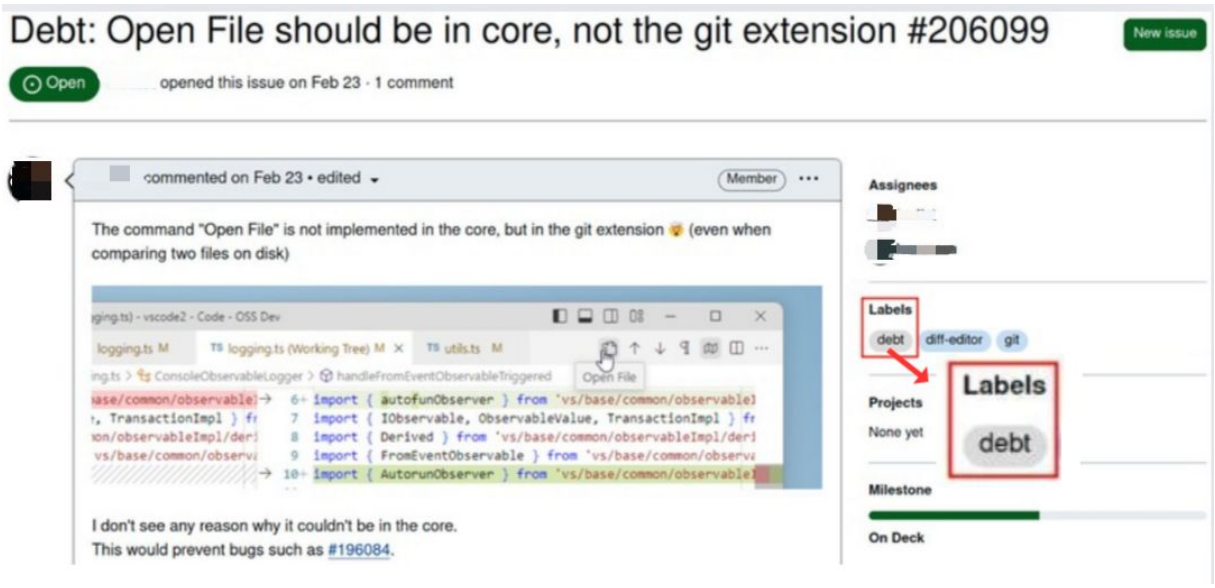

Figure 1: SATD instance in the VSCode repository

2.3 Prompt Engineering

Prompt engineering is an emerging field that explores the importance of constructing well-defined prompts to enhance the performance of large language models (LLMs). Although the field is not yet fully consolidated, its effect and importance have been discussed in recent literature [3, 17, 18, 28, 33, 36, 39]. By bridging the gap between the intention behind the prompt and the machine's actions, well-engineered prompts help overcome the complexity of human communication. Since the task of identifying SATD involves recognizing patterns that are not always explicit, reasoning and knowledge of software engineering theory and practice, crafting a prompt that fully encapsulates the context is a important step of using LLMs as a research tool.

In this context, we use the definition of a prompt being a collection of data and instructions that aim to guide a language model towards a desired output [18]. A prompt typically consists of four elements: (1) The instruction that guides the model's behavior; (2) The template guiding the model's output; (3) The input data that the model should process; (4) The context given, aiming to guide the model's understanding;

Prompt engineering offers a wide array of strategies to optimize the effectiveness of prompts in various ways. These strategies range from providing more context in a specific manner to teach the model how to act, to specifying how the model should respond in an effort to encourage reasoning. Several studies have systematically classified and categorized these strategies [17, 33]. For the purpose of this research, we selected a subset of these strategies to employ and analyze, focusing on those that are most relevant to the task of identifying SATD using LLMs and would not need to be crafted towards a single, specific, software project since we are pursuing a generalizable approach. The prompts used are as follows:

- Zero-shot prompt: A prompt that provides no examples or additional context.
- Think step-by-step prompt: A prompt that encourages the model to reason in multiple steps.
- Few-shot prompt: A prompt supplemented with a few examples to guide the response.
- Chain-of-Thought prompt: A prompt that explicitly sequences thoughts logically.

3 STUDY DESIGN

In this section, we outline the key components of our study's design, detailing our methodology and data. We discuss the selection criteria for models and datasets, provide an overview of our prompt engineering strategies and describe the evaluation metrics used to measure the efficacy of the models.

Model Selection. To conduct the analysis of LLMs' performance in identifying SATD, our study begins with the selection of models and prompts. We based our selection on the Multitask Multi-Modal Language Understanding Benchmark (MMLU) [1], ensuring compatibility within our budget constraints. Consequently, we chose Claude 3 Haiku, Google Gemini 1.0 Pro, and gpt-3.5-turbo-0125 as our models.

Dataset. To assess the efficacy of our selected models, we adopted a appropriate dataset. To compare our approach with existing ones, we adopted the dataset employed in evaluating the generalization capacity of the current state-of-the-art identification model described in Skryseth et al. [35]. These datasets are composed of issues mined from publicly available repositories. They were categorized as either "TD" or "Not_TD" based on the presence or absence of a label containing the term "debt". The adopted dataset include issues from the repositories outlined in Table 1.

In total we analyse 8,663 issues, 3,910 being classified as TD and 4,753 issues being classified as not TD.

Repository	Total Issues	TD Issues	Not_TD Issues
UBC Thunderbolts	847	538	309
Apache Traffic	616	362	254
Owncloud	1,200	553	647
VA Gov	6,000	2,457	3,543

Table 1: Summary of datasets used for analysis

Prompt Engineering Considering the classifications and elements that compose a prompt defined in Section 2.3, we engineered four prompts. The output's template and the input data have limited room for change, so we focus on optimizing the instructions given to the model and the context provided to inform its understanding. Initially, we tested each model's ability to identify SATD using a zero-shot prompt, as shown in Figure 2.

```
Does the issue bellow should have a "Self Admitted Technical Debt"
 label?
Your answer should be "YES" or "NO".
issue{
   "title": {issue_title}
   "body": {issue_body}
}
```

Figure 2: Zero-Shot Prompt

We subsequently crafted the prompts based on methodologies detailed in prior studies [17, 33]. We began with the straightforward "think step-by-step" strategy, which has demonstrated its effectiveness in enhancing model performance on reasoning tasks by explicitly guiding the model to reason through each step of the answer. The resulting prompt is illustrated in Figure 3.

```
Does the issue bellow should have a "Self Admitted Technical Debt"
 label? Think step-by-step
issue text{
    {issue}
}
Your answer should start with "YES" or "NO".
```

Figure 3: Think Step-by-Step Prompt

We also employed the "Few-shot Learning" strategy. To achieve this, we selected a balanced set of examples from the GitHub Technical Debt Dataset [35] which was used to train the state of the art model to identify SATD but are not included in the issues used to test the models' performance, ensuring an equal number of SATD and non-SATD examples to avoid biasing the LLMs. The SATD examples were extracted for the reason of clearly stating the admission of technical debt in either code duplication or sub-optimal implementation due to time pressure. The non-SATD examples encompass a question and a feature request, both not related to Technical Debts. The resulting prompt is shown in Figure 4.

The last strategy we employed was "Chain-of-thought reasoning" (CoT). Describing the exact steps to identify SATD is a complex task, and these steps might be subjective. To ensure that we were instructing the model properly, we first described anything related to explicitly discussing something that would fit the formal definition of Self-Admitted Technical Debt in the software engineering

```
Does the issue below should have a "Self Admitted Technical Debt"
label?
Examples:
SATD Issue{{
 "I wrote up the polling solution really quickly when I was late
 for work and never went back to fix it. Instead of saving message
  IDs or keeping track of the text of the poll as one variable, it
  saves a new copy of the poll every time someone adds or removes
 a vote. This is horrible and I'm embarrassed that I haven't fixed
  it yet"
}}
Answer: YES

Non-SATD Issue{{
    "### Search before asking

- [X] I have searched the YOLOv5 [issues](https://github.com/
ultralytics/yolov5/issues) and [discussions](https://github.com/
ultralytics/yolov5/discussions) and found no similar questions.


### Question

Why is it set like this?
    hyp['box'] *= 3 / nl  # scale to layers
    hyp['cls'] *= nc / 80 * 3 / nl  # scale to classes and layers
    hyp['obj'] *= (imgsz / 640) ** 2 * 3 / nl  # scale to image
    size and layers
default  hyp['box'] : hyp['cls'] :  hyp['obj'] = 1:1:1
if my data nc is 3, hyp['box'] : hyp['cls'] :  hyp['obj'] = 1:
3/80 :1????

### Additional

_No response_"
}}
Answer: NO

issue{{
    "Logic for manipulating resource path (and paths generally,
    for that matter) is scattered and duplicated all over the
    application. This needs to be tracked and unified into a set
    of standard general-purpose path utilities."
}}
Answer: YES

issue{{
"There's often confusion whenever daylight savings changes,
because people forget to change their timezone on the website.
We can detect when the browser's timezone is different from the
timezone on the player's profile. In that case, we can prompt the
user if they want to change their profile timezone."
}}
Awnser: NO

Does the issue below should have a "Self Admitted Technical Debt"
label? YOUR ANSWER SHOULD BE JUST "YES" OR "NO".
issue text{{
    {issue}
}}
```

Figure 4: Few-shot Prompt

literature[32]. Additionally, we listed keywords that were found to be the most common in issue descriptions[23] and keywords generally used to describe technical debt[32], to inform the model about the common elements in issue descriptions. The final result is shown in Figure 5.

Evaluation Metrics. To be able to compare the performance of each model in our analysis we used the F1-score, precision, recall, and the Matthews Correlation Coefficient (MCC) metric. The F1-score is the harmonic mean of precision and recall, providing a measure of a model's performance. Equation 1 represents the F1-score formula. Precision represents the proportion of true positive predictions among all positive predictions, and it's formula is shown on Equation 2. Recall represents the proportion of true positive predictions among all actual positive instances, Equation 3 shows

```
Analyze the given issue description to identify the presence of
Self-admitted technical debt (SATD). Follow these steps to
determine if SATD is present:

Step 1: Identify any mentions of shortcuts, workarounds, or
temporary solutions in the issue description. These may indicate
the presence of SATD.

Step 2: Look for phrases that suggest the implementation is
incomplete, suboptimal, or requires future refactoring. Examples
include "typo", "leak", "flaky", "unnecessary", "performance", "
checkstyle", "spelling", "unused", "cleanup", "coverage", "TODO",
"FIXME", "hack", "not ideal", "needs improvement" or similar
expressions.

Step 3: Check if the issue description acknowledges any design or
architectural limitations that may incur technical debt in the
future.

Step 4: Determine if the issue description mentions any time
constraints, pressure to deliver, prioritization of speed over
quality or if the issue description discusses any compromises made
 in the implementation, such as using a less efficient algorithm,
hardcoding values, or skipping necessary validations, which may
lead to SATD.

Step 5: Consider if the issue description indicates any planned or
 required refactoring, code cleanup, or performance optimizations
in the future.

Based on the presence or absence of the above indicators, conclude
 whether the issue contains Self-admitted technical debt or not.
Provide a detailed analysis of each identified instance of SATD,
including the relevant excerpt from the issue, an explanation of
why it is considered SATD, and any additional context or insights.

Start your answer with "YES" or "NO".

issue text{{
    {issue}
}}
```

Figure 5: Chain-of-Thought Prompt

the formula for calculating it. Lastly, the MCC metric is a reliable metric for binary classification [8], producing a high score only if the prediction obtained good results in all four confusion matrix categories (true positives - TP, false negatives - FP, true negatives - TN, and false positives - FP). The formula for MCC is shown on Equation 4

The formulas for each metric are as follows:

$$F1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{1}$$

$$\text{precision} = \frac{TP}{TP + FP} \tag{2}$$

$$\text{recall} = \frac{TP}{TP + FN} \tag{3}$$

$$MCC = \frac{TP + TN - FP - FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{4}$$

4 RESULTS

In this section, we present the findings of our study, detailing the performance of various combinations of models and prompts in identifying SATD. We provide an overview of the results for each research question, discuss the impact of different prompt engineering strategies, and compare the models using the evaluation metrics detailed in Section 3.

RQ1: How does the performance of LLMs compare to the state-of-the-art models for SATD identification?

To answer RQ1, we used the same dataset used to test the state-of-the-art generalization performance. We've identified this model as "Skryseth" in Table 2, referring to the author of the paper related to the state-of-the-art model.

Table 2: Comparison between models and state of the art

	Skryseth	Claude	GPT	Gemini
F1 Score	0.66	**0.68**	0.48	0.61
Precision	**0.74**	0.45	0.57	0.57
Recall	0.54	**0.88**	0.42	0.67
MCC	**0.39**	0.15	0.08	0.12

As shown in Table 2, Claude 3 achieved the best performance among the LLMs, with an F1 score of 0.68, which is comparable to the state-of-the-art model by Skryseth (0.66). However, it is important to note that Skryseth's model demonstrates higher precision (0.74) and MCC (0.39) compared to Claude 3 (0.45 and 0.15, respectively). This indicates that while Claude 3 has a similar overall performance, the state-of-the-art model is more precise in its predictions and generates less false positives.

Additionally, when examining the recall metric, we observe that LLMs tend to have higher recall values. Claude 3, in particular, achieves a recall of 0.88, which is 34 percentage points higher than Skryseth's model (0.54). This suggests that LLMs may be biased towards providing positive results, leading to a higher recall but relatively lower precision.

GPT 3.5 turbo and Gemini 1.0 pro had lower F1 scores of 0.48 and 0.61, respectively, indicating that their performance in identifying SATD is not as strong as Claude 3 Haiku or Skryseth. The MCC scores for these models were also lower, suggesting that their predictions are less balanced.

Summary

Claude 3 achieved performance comparable to the state-of-the-art model Skryseth, but with lower precision and MCC, indicating a tendency towards more false positives.

RQ2: What is the impact of prompt engineering on the performance of LLMs in the task of SATD identification?

To answer RQ2, we averaged out the performance of each prompt across every dataset per model. We used the MCC metric to compare the performance of each prompt/model, since it produces a high score only if the prediction obtained good results in all four confusion matrix categories. With that, we are able to better tell when a prompt was able to prune a model's bias towards an answer and give a more balanced result.

Table 3 highlights that in each model there was at least one engineered prompt that showed better performance than the zero-shot prompt. This improvement was especially notable in Claude, where the best prompt (Few-shot) achieved a 40% increase in MCC score from the original prompt. However, it is important to note that not all engineered prompts outperformed the original prompt.

Table 3: MCC Scores per prompt

	Claude	GPT	Gemini
Zero-shot	0.15	0.08	0.12
Think step-by-step	0.11	0.06	0.13
Few-shot	**0.21**	0.07	**0.15**
Chain-of-thought	0.04	**0.09**	0.10

For instance, the worse prompt performance in Claude represented a 73% decrease in performance when compared to the zero-shot prompt.

It is also worth noting that each model benefited differently from prompt engineering. Claude 3 showed the most significant improvement (40%), while GPT and Gemini had more modest gains of 12.5% and 25%, respectively, from their original prompts to their best-performing engineered prompts.

Summary

Prompt engineering significantly enhanced model performance, with Claude 3 showing the most improvement (40%), followed by Gemini (25%) and GPT (12.5%). However, the effectiveness of prompts varied widely.

5 DISCUSSION

In this section, we interpret the results of our study, examining the performance of LLMs in identifying SATD and evaluating the influence of prompt engineering. We discuss the comparative performance of the models, analyze the impact of different prompting strategies, and situate our findings within the broader context of existing research.

RQ1: How does the performance of LLMs compare to the state-of-the-art models for SATD identification?

Our study reveals that among the evaluated LLMs, Claude 3 demonstrated the highest performance in identifying SATD, with an F1 score of 0.68, comparable to the state-of-the-art model by Skryseth, which achieved an F1 score of 0.66. However, the state-of-the-art model showed higher precision (0.74) and MCC (0.39) compared to Claude 3's precision of 0.45 and MCC of 0.15. This indicates that Claude 3, while performing well in general, tends to generate more false positives than the state-of-the-art model. Other models, GPT 3.5 turbo and Gemini 1.0 pro, exhibited lower overall performance, with F1 scores of 0.48 and 0.61, respectively. Overall, Claude 3's competitive F1 score demonstrates the potential of LLMs as general-purpose models that can adapt to SATD identification without domain-specific training which could be seen as a alternative for models specifically designed for SATD identification.

RQ2: What is the impact of prompt engineering on the performance of LLMs in the task of SATD identification?

The study also shows that prompt engineering substantially impacts the performance of LLMs in the task of SATD identification, particularly, the Few-shot prompt was the most effective. For instance, Claude 3 had a 40% increase in performance. GPT and Gemini also benefited from prompt engineering, though to a lesser extent, with improvements of 12.5% and 25%, respectively, when looking at the best prompt for each one. Not all engineered prompts were beneficial, with some even reducing the model's performance.

In general, our findings align with recent studies exploring the potential of LLMs in software engineering tasks [29]. The comparable performance of Claude 3 to the state-of-the-art model in SATD identification supports the notion that LLMs can be effective in this domain. However, the lower precision and MCC scores of LLMs compared to the state-of-the-art model are consistent with observations from previous studies, which have highlighted the limitations of LLMs in independently resolving SATD [29].

The varying impact of prompt engineering on LLM performance in our study resonates with the growing body of literature emphasizing the importance of crafting effective prompts [17, 18]. Our results demonstrate that well-designed prompts can enhance LLM performance in SATD identification, but the effectiveness of specific prompt engineering strategies may vary across different models.

6 CONCLUSION AND FUTURE WORK

In this study, we explored the potential of LLMs for identifying SATD and compared their performance against a state-of-the-art model specifically designed for this task. Our findings demonstrate that LLMs, particularly Claude 3 Haiku, can achieve competitive results in SATD identification. However, the performance of LLMs varies depending on the specific model and the prompt used, emphasizing the importance of prompt engineering in optimizing their effectiveness.

The impact of prompt engineering on the performance of LLMs in SATD identification was evident from our experiments. Different prompts led to varying levels of performance for each model, with the Few-shot prompt yielding the most balanced predictions across all models. This highlights the need for careful prompt design and experimentation to find the most effective prompt for a given model and task.

Our study contributes to the growing body of research on the application of LLMs in software engineering tasks, specifically in the context of SATD identification. The findings suggest that LLMs have the potential to be valuable tools for managing and addressing technical debt in software projects. However, further research is needed to fully understand the capabilities and limitations of LLMs in this domain and to develop best practices for prompt engineering.

Future work could explore the use of LLMs for other aspects of technical debt management, such as prioritization, payment and its documentation. Additionally, investigating the interpretability and explainability of LLMs in the context of SATD identification could provide valuable insights for practitioners and researchers alike. As a final note, the replication package with all our data is publicly available at: https://zenodo.org/records/11406252

REFERENCES

[1] 2024. *MMLU Benchmark*. https://paperswithcode.com/sota/multi-task-language-understanding-on-mmlu
[2] Open ai. 2024. *Chat GPT*. https://Chat.openai.com
[3] Mohammad Aljanabi, Mohanad Ghazi Yaseen, Ahmed Hussein Ali, and Mostafa Abdulghafoor Mohammed. 2023. Prompt Engineering: Guiding the Way to Effective Large Language Models. *Iraqi Journal For Computer Science and Mathematics* 4, 4 (Nov. 2023), 151–155. https://doi.org/10.52866/ijcsm.2023.04.04.012

[4] Anthropic. 2024. *Anthropic Console.* https://console.anthropic.com/dashboard

[5] Paris Avgeriou, Philippe Kruchten, Ipek Ozkaya, and Carolyn Seaman. 2016. Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162). *Dagstuhl Reports* 6 (01 2016). https://doi.org/10.4230/DagRep.6.4.110

[6] Gabriele Bavota and Barbara Russo. 2016. A Large-Scale Empirical Study on Self-Admitted Technical Debt. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. 315–326.

[7] Loredana Caruccio, Stefano Cirillo, Giuseppe Polese, Giandomenico Solimando, Shanmugam Sundaramurthy, and Genoveffa Tortora. 2024. Claude 2.0 large language model: Tackling a real-world classification problem with a new iterative prompt engineering approach. *Intelligent Systems with Applications* 21 (2024), 200336. https://doi.org/10.1016/j.iswa.2024.200336

[8] Davide Chicco and Giuseppe Jurman. 2020. The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. *BMC Genomics* 21, 1 (2020), 6. https://doi.org/10.1186/s12864-019-6413-7

[9] Marcus Ciolkowski, Valentina Lenarduzzi, and Antonio Martini. 2021. 10 Years of Technical Debt Research and Practice: Past, Present, and Future. *IEEE Software* 38, 6 (2021), 24–29. https://doi.org/10.1109/MS.2021.3105625

[10] Ward Cunningham. 1992. The WyCash portfolio management system. *SIGPLAN OOPS Mess.* 4, 2 (dec 1992), 29–30. https://doi.org/10.1145/157710.157715

[11] Rishi Bommasani et al. 2022. On the Opportunities and Risks of Foundation Models. arXiv:2108.07258 [cs.LG]

[12] Martin Fowler. 2005. *Detestable.* https://martinfowler.com/bliki/Detestable.html

[13] Martin Fowler. 2006. *CodeSmells.* https://martinfowler.com/bliki/CodeSmell.html

[14] Martin Fowler. 2007. *DesignStaminaHypothesis.* https://martinfowler.com/bliki/DesignStaminaHypothesis.html

[15] Martin Fowler. 2008. *EstimatedInterest.* https://martinfowler.com/bliki/EstimatedInterest.html

[16] Martin Fowler. 2009. *TechnicalDebtQuadrant.* https://martinfowler.com/bliki/TechnicalDebtQuadrant.html

[17] Andrew Gao. 2023. Prompt Engineering for Large Language Models. *SSRN* (8 July 2023), 8. https://doi.org/10.2139/ssrn.4504303 8 Pages Posted: 17 Jul 2023.

[18] Louie Giray. 2023. Prompt Engineering with ChatGPT: A Guide for Academic Writers. *Annals of Biomedical Engineering* 51, 12 (01 12 2023), 2629–2633. https://doi.org/10.1007/s10439-023-03272-4

[19] Google. 2024. *Gemini.* https://gemini.google.com/

[20] Kemal Kirtac and Guido Germano. 2024. Sentiment trading with large language models. *Finance Research Letters* 62 (2024), 105227. https://doi.org/10.1016/j.frl.2024.105227

[21] Oscar N.E. Kjell, Katarina Kjell, and H. Andrew Schwartz. 2024. Beyond rating scales: With targeted evaluation, large language models are poised for psychological assessment. *Psychiatry Research* 333 (2024), 115667. https://doi.org/10.1016/j.psychres.2023.115667

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

[23] Y. Li, M. Soliman, and P. Avgeriou. 2023. Automatic identification of self-admitted technical debt from four different sources. *Empir Software Eng* 28 (2023), 65. https://doi.org/10.1007/s10664-023-10297-9

[24] Y. Li, M. Soliman, P. Avgeriou, and M. Van Ittersum. 2023. DebtViz: A Tool for Identifying, Measuring, Visualizing, and Monitoring Self-Admitted Technical Debt. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE Computer Society, Los Alamitos, CA, USA, 558–562. https://doi.org/10.1109/ICSME58846.2023.00072

[25] Erin Lim, Nitin Taksande, and Carolyn Seaman. 2012. A Balancing Act: What Software Practitioners Have to Say about Technical Debt. *IEEE Software* 29, 6 (2012), 22–27. https://doi.org/10.1109/MS.2012.130

[26] Everton Da S. Maldonado, Rabe Abdalkareem, Emad Shihab, and Alexander Serebrenik. 2017. An Empirical Study on the Removal of Self-Admitted Technical Debt. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 238–248. https://doi.org/10.1109/ICSME.2017.8

[27] Robert C. Martin. 2009. *A mess is not technical debt.* https://sites.google.com/site/unclebobconsultingllc/a-mess-is-not-a-technical-debt

[28] Ggaliwango Marvin, Nakayiza Hellen, Daudi Jjingo, and Joyce Nakatumba-Nabende. 2024. Prompt Engineering in Large Language Models. In *Data Intelligence and Cognitive Informatics*, I. Jeena Jacob, Selwyn Piramuthu, and Przemyslaw Falkowski-Gilski (Eds.). Springer Nature Singapore, Singapore, 387–402.

[29] A. Mastropaolo, M. Di Penta, and G. Bavota. 2023. Towards Automatically Addressing Self-Admitted Technical Debt: How Far Are We?. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Los Alamitos, CA, USA, 585–597. https://doi.org/10.1109/ASE56229.2023.00103

[30] Steve McConnell. 2008. *Managing Technical Debt.* http://www.construx.com/uploadedfiles/resources/whitepapers/Managing%20Technical%20Debt.pdf

[31] Microsoft. [n. d.]. *Visual Studio Code.* https://github.com/microsoft/vscode

[32] Aniket Potdar and Emad Shihab. 2014. An Exploratory Study on Self-Admitted Technical Debt. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 91–100. https://doi.org/10.1109/ICSME.2014.31

[33] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. 2024. A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications. *arXiv preprint arXiv:2402.07927* (2024). https://doi.org/10.48550/arXiv.2402.07927 Submitted on 5 Feb 2024.

[34] Giancarlo Sierra, Emad Shihab, and Yasutaka Kamei. 2019. A survey of self-admitted technical debt. *Journal of Systems and Software* 152 (2019), 70–82. https://doi.org/10.1016/j.jss.2019.02.056

[35] Daniel Skryseth, Karthik Shivashankar, Ildikó Pilán, and Antonio Martini. 2023. Technical Debt Classification in Issue Trackers using Natural Language Processing based on Transformers. In *2023 ACM/IEEE International Conference on Technical Debt (TechDebt)*. 92–101. https://doi.org/10.1109/TechDebt59074.2023.00017

[36] Yu Tian, Ang Liu, Yun Dai, Keisuke Nagato, and Masayuki Nakao. 2024. Systematic synthesis of design prompts for large language models in conceptual design. *CIRP Annals* (2024). https://doi.org/10.1016/j.cirp.2024.04.062

[37] The Verger. 2023. *ChatGPT continues to be one of the fastest-growing services ever.* https://www.theverge.com/2023/11/6/23948386/chatgpt-active-user-count-openai-developer-conference

[38] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022. Emergent Abilities of Large Language Models. arXiv:2206.07682 [cs.CL]

[39] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. arXiv:2302.11382 [cs.SE]

[40] Laerte Xavier, Rodrigo dos Santos, Sandalo Bessa, and Marco Tulio Valente. 2020. Ltd: Less Technical Debt Framework. *SSRN* (2020). https://ssrn.com/abstract=4397233 Available at SSRN: https://ssrn.com/abstract=4397233 or http://dx.doi.org/10.2139/ssrn.4397233.

[41] Laerte Xavier, Fabio Ferreira, Rodrigo Brito, and Marco Tulio Valente. 2020. Beyond the Code: Mining Self-Admitted Technical Debt in Issue Tracker Systems. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 137–146. https://doi.org/10.1145/3379597.3387459

[42] Laerte Xavier, Jean Eduardo Montandon, Fabio Ferreira, et al. 2022. On the documentation of self-admitted technical debt in issues. *Empir Software Eng* 27 (2022), 163. https://doi.org/10.1007/s10664-022-10203-9

[43] Laerte Xavier, João Eduardo Montandon, Fabio Ferreira, Rodrigo Brito, and Marco Tulio Valente. 2022. On the documentation of self-admitted technical debt in issues. *Empirical Software Engineering* 27, 7 (2022), 163. https://doi.org/10.1007/s10664-022-10203-9

[44] Laerte Xavier, João Eduardo Montandon, and Marco Tulio Valente. 2022. Comments or Issues: Where to Document Technical Debt. *IEEE Software* 39, 5 (2022), 84–91. https://doi.org/10.1109/MS.2022.3170825