Evaluating Large Language Models on the Classification of Different Technical Debt Types in Stack Overflow Discussions

Lucas Amaral
Universidade Estadual do Ceará
Fortaleza, Brasil
lucas.amaral@aluno.uece.br

Eliakim Gama
Universidade Estadual do Ceará
Fortaleza, Brasil
eliakim.gama@aluno.uece.br

Matheus Paixao
Universidade Estadual do Ceará
Fortaleza, Brasil
matheus.paixao@uece.br

Lucas Aguiar
Universidade Estadual do Ceará
Fortaleza, Brasil
lucas.aguiar@aluno.uece.br

ABSTRACT

Technical Debt (TD) refers to suboptimal decisions made during software development that offer short-term benefits at the cost of long-term maintainability. Managing TD is critical for ensuring the sustainability of software systems, especially as projects evolve. While prior research has leveraged machine learning techniques to identify TD in data from platforms such as Stack Overflow (SO), those approaches have shown limited performance. To address these limitations, this study investigates the effectiveness of transformer-based Large Language Models (LLMs) for the automated identification and classification of TD types in SO discussions. We evaluated three prominent LLMs: BERT, BART, and GPT-2, on their ability to classify multiple types of TD. Our contributions are: (i) a reproducible training/evaluation pipeline on an SO TD dataset, and (ii) a comparison against prior studies. LLMs reach up to 85% F1 and 78.6% average F1, outperforming previous results by 8–23%.

KEYWORDS

Technical Debt, Stack Overflow, Large Language Models

1 Introduction

The build-up of Technical Debt (TD) is a pervasive and often a unavoidable consequence of trade-offs and time pressures inherent in modern software development processes [19]. Developers may delay refactoring or compromise architecture to meet short-term goals, degrading long-term quality [8]. TD spans code, tests, documentation, design, build, and architecture [2].

As systems evolve and increase in complexity, the ability to monitor and manage TD becomes essential for maintaining software sustainability, architectural coherence, and development speed [6, 23]. Poor TD management can result in maintenance costs, delayed feature delivery, and diminished developer productivity [31].

To understand better TD in practice, several studies have turned to community platforms like SO, leveraging both manual analysis and traditional Machine Learning (ML) to detect TD-related content [12, 13, 15, 18]. While foundational, manual analyses do not scale and traditional ML often underperforms on noisy SO text.

Recent advancements in Natural Language Processing have positioned LLMs as state-of-the-art tools for a variety of software engineering tasks, including code summarization, bug localization, and developer intent classification [35]. Among the most prominent LLMs, models such as BERT[10], known for its robust contextual embeddings and precision in classification tasks, BART[22], valued for its versatile encoder–decoder design and balanced performance across tasks, and GPT[16], recognized for its fluent generative abilities and adaptability to varied contexts, have gained traction due to their strong empirical performance, adaptability to downstream tasks, and capability to capture intricate semantic relationships in both natural and programming languages. These models have been successfully used in prior software artifact analysis, with validated effectiveness across diverse domains, such as source code[7], code comments [32], commit messages [27], and issue discussions [21].

Leveraging their pretraining and contextual understanding, LLMs offer an advantage over classical ML techniques, particularly in tasks requiring semantic interpretation, classification, and information extraction from unstructured or informal developer content.

Previous studies on TD identification in SO discussions either relied on manual analysis, which is slow, non-scalable, and subjective [14], or on traditional ML models [12, 17], which achieved limited performance (often under 70% F1 for several TD types) and struggled with the nuanced, context-dependent language of developer discussions. Our approach addresses these limitations by fine-tuning transformer-based LLMs (BERT, BART, GPT) for per-type classification across five TD categories, directly comparing multiple architectures under a unified protocol, and analyzing performance across epochs to identify optimal training durations.

Hence, in this study, we propose an automated approach to classify TD types in SO discussions using LLMs.

Our investigation is guided by the following research questions:

- **RQ1:** To what extent can LLMs correctly classify different types of TD on SO discussions?
  Goal: Evaluate and compare the effectiveness of different LLM variants (e.g., BERT, BART, GPT) in the classification of different TD Types in SO discussions.
  Summary: Model performance varied across TD types, with the best results for Tests debt and Infrastructure debt, and lower performance for Design debt. BART showed the best overall balance with the highest average F1-Score (77.6%), while BERT achieved the highest recall (92.4%).
- **RQ2:** How do LLMs compare to state-of-the-art approaches in classifying different TD types in SO discussions?
  Goal: Compare LLMs and state-of-the-art approaches in classifying different types of TD in SO discussions.

Summary: LLMs achieved the highest average F1-Score (78.6%), precision (87.2%), and recall (93.2%), demonstrating their effectiveness in classifying different types of TD.

- **RQ3:** At which fine-tuning epochs do LLMs achieve peak performance for TD classification on SO discussions?
  Goal: Analyze LLMs learning curves and identify optimal training stages.
  Summary: Most models reached peak performance between epochs 2-6, varying across TD types. Our results suggest that optimal training duration is model and task-dependent.

2 Background

2.1 Technical Debt

TD describes shortcuts that compromise the long-term health of software to achieve short-term goals [9]. During the software lifecycle, TD can emerge in artifacts, depending on its occurrence timing and the activities it is related to. Considering these aspects, TD can be classified into the following types [26]: design, code, architecture, tests, documentation, defects, infrastructure, requirements, people, build, process, automated tests, usability, service, and versioning.

Unchecked TD accrues "interest" as added effort, defect risk, and reduced evolvability [24]. This interest typically manifests as increased effort and complexity in implementing new features, a higher probability of introducing new defects during maintenance activities, and degradation of the system's internal quality. In the long run, unchecked TD can lead to higher maintenance costs, decreased developer productivity, and a diminished capacity for the software to evolve in response to new business requirements [24].

2.2 Stack Overflow

SO is a Q&A platform that has established itself as a hub for computing professionals, facilitating discussions and problem-solving related to software development, due to its backlog of knowledge [34]. Threads include answers and comments that evolve into discussions, offering insight into technologies and developer practices [5].

SO has attracted research interest in many topics related to both community dynamics, human factors, and technical issues [25]. The data generated through SO's questions and answers has been utilised in research such as microservices [4], sentiment analysis [29], and code smells [30], to mention only a few.

2.3 Large Language Models

LLMs are a form of artificial intelligence that is trained on vast amounts of text data, which is why they are called "Large" [21], enabling them to understand, generate, and interact with human language in a remarkably nuanced way. LLMs can be classified into types based on their architecture:

Autoregressive Language Models, such as OpenAI Generative Pre-trained Transformer (GPT) [16] focus on the decoder part of the Transformer architecture [33]. These models generate text by predicting the next word based on the phrase context, making them suitable for text generation and summarization.

Auto Encoding Language Models, as exemplified by BERT [10] and BART [22], focus on the encoding part. These models use vector representations, called embeddings, to represent text and are tasked to reconstruct sentences based on corrupted or masked text. Since they focus on learning how to represent text, they excel especially in text classification and question answering.

For this study, we employ GPT, BART, and BERT, which have previously been widely used in the context of TD [1, 21, 28].

One of our objectives in this study was to evaluate the performance of each model throughout its training process, especially across different epochs. To that end, we chose to fine-tune the models ourselves, which allowed us to track their learning behavior over time and gain a deeper understanding of their adaptation to the task, something that would not be possible using prompt engineering.

3 Related Work

In this section, we review existing studies that have addressed the identification and classification of TD, particularly in SO, through both manual and automated approaches, including traditional ML.

Gama et al. [14] conducted a manual analysis of 140 Stack Overflow discussions to understand how developers identify TD in practice, resulting in a framework of 29 low-level and 13 high-level indicators categorized by TD type.

Kozanidis et al. [17] combined manual and automated analysis to characterize TD questions on SO. The authors built a labeled dataset and used Random Forest to classify TD presence, urgency, and type. Although the results were promising, the classification of specific Technical Debt types showed limited effectiveness, with F1-scores of 60% and 61% for Code and Infrastructure, respectively. The study serves as baseline for automatic TD classification on SO.

Edbert et al. [11] investigated TD in the context of software security by analyzing 117,000 security questions on SO using a RoBERTa-based classifier. They identified security TD questions and highlighted patterns in user profiles and question structure.

Gama et al. [12] extended earlier efforts by proposing a fully automated approach based on traditional ML algorithms (e.g., Random Forest, XGBoost, SVC). Using the manually labeled dataset of SO discussions from their previous work, the authors trained classifiers to detect and classify five TD types. Their results showed improvement over previous baselines.

Sheikhaei et al. [27] evaluated effectiveness of LLMs (BERT, CodeBERT, Flan-T5 variants) for SATD identification and classification in source code comments, different from our study, which analyzed TD in SO discussions. Fine-tuned Flan-T5-XL outperformed non-LLM baselines, though performance varied by model architecture.

Lambert et al. [21] explored the application of LLMs to the identification of Self-Admitted Technical Debt (SATD) in issue trackers, focusing on prompt engineering strategies. The results achieved competitive recall, but precision and MCC lagged behind specialized models without well-crafted prompts.

Prior works vary in focus, ranging from general TD detection [12, 17], to SATD in code comments and issue trackers [21, 27], and domain-specific studies such as security TD [11]. Traditional ML approaches have shown utility for TD detection but struggle with nuanced classification, particularly across TD types. Recent work leveraging LLMs highlights their potential but also exposes the need for fine-tuning and prompt optimization.

Our study builds upon and extends these efforts by (i) directly comparing the performance of LLMs and classical ML models on a unified TD detection and classification task in SO discussions;

(ii) conducting a fine-grained epoch-wise analysis of LLM fine-tuning performance; and (iii) evaluating multiple LLM architectures across specific TD types. Table 1 provides an overview of the main differences and similarities among the related works.

Table 1: Summary of related work on TD identification and classification

Reference	Year	Type of Analysis	Technique	LLM?	TD Task	SO?
Gama et al. (2020)	2020	Manual	Manual Analisys	No	Identification	Yes
Kozanidis et al. (2022)	2022	Manual and Automated	ML	No	Identification and Classification	Yes
Aldrich et al. (2023)	2023	Automated	ML	No	Identification	Yes
Gama et al. (2023)	2023	Automated	ML	No	Identification and Classification	Yes
Lambert et al. (2024)	2024	Automated	LLMs	Yes	Identification	No
Sheikhaei et al. (2024)	2024	Automated	LLMs	Yes	Identification and Classification	No
This Study	2025	Automated	LLMs	Yes	Classification	Yes

4 Experimental Setup

This section outlines the technical setup and process that we followed to obtain the results presented in this study. We describe the preprocessing of the dataset, the construction of balanced subsets for classification, and the subsequent execution of model training and evaluation. The complete dataset, along with the project code and models used, is available in our replication package [3].

4.1 Data Source

The data source for this work was originally produced by Gama et al. [12], who conducted a manual analysis of 372 Stack Overflow discussions to identify instances of TD. The resulting dataset represents 2,255 individual discussion comments, each annotated with a specific TD type. These TD comment instances are distributed across five types: Code (1,287), Infrastructure (410), Architecture (256), Test (224), and Design (78). The dataset is organized as a CSV file with three columns: Id (linking each comment to its discussion thread), Category (indicating the TD type), and Body (containing the text of the comment itself). The original comments often contained code snippets, special symbols, and programming syntax that could introduce noise into the models. To mitigate this, we applied a preprocessing pipeline, as explained in Section 4.2. Notably, the dataset exhibits significant class imbalance, with Code type instances making up more than half of all entries. The strategy to address this imbalance is detailed in Section 4.3.

4.2 Dataset Preprocessing

To prepare the textual data for classification, we applied a sequence of preprocessing operations aimed at reducing noise, eliminating code specific patterns, and standardizing the input format. Initially, all text was subjected to **lower-case normalization**. This transformation ensures consistency by preventing the same word from being treated as different tokens due to the case variations, such as 'Bug' and 'bug'. Next, we applied **special symbol removal** to eliminate programming specific symbols. Finally, **stopword removal** was performed, as such words typically carry little to no semantic weight and can dilute patterns in the data. The result of this process gave us a set of SO discussion comments properly adapted for being split into the datasets that will be used for the following steps.

4.3 Dataset Splitting

To enable the training and evaluation of classification models, we structured the dataset into appropriate subsets through a multi-step splitting process. This included the creation of balanced datasets for each TD type and the subsequent split of each dataset into training and testing partitions. These steps are important to ensure a fair evaluation and to mitigate the effects of class imbalance on model performance. Next, we detail the two phase process we employed.

4.3.1 *Phase 1: Dataset Balancing.*

To ensure an unbiased evaluation of our classification models, we adopted a binary classification strategy for each TD type, requiring the construction of balanced datasets, similar to the approach taken by Gama et al. [12]. As we mentioned in Section 4.1, it was necessary to address the strong class imbalance present in the original dataset, where some TD types had significantly fewer instances than others. In a multiclass setup, this imbalance could lead to models that favor the majority classes, resulting in misleading performance metrics.

For each TD type under investigation, we generated an individual dataset in which all instances labeled with that specific TD type were treated as positive examples (TD), while all other instances were treated as negative examples (Non-TD). To build a balanced dataset for each TD type, we selected all positive samples corresponding to the target TD type and all remaining samples as negatives. Since the positive and negative samples varied across types, we determined the examples to use from each class by selecting the minimum between the positive and negative instances. The resulting subsets were labeled and merged into a single balanced dataset. The balanced datasets were saved in separate files, one per TD type, and served as the input for the training and evaluation process described in the next stages. This procedure was repeated for each of the five TD types included in this study.

4.3.2 *Phase 2: Splitting of Training and Testing.*

Considering each balanced dataset, as described above, we proceeded to split the data into training and testing subsets. To ensure alignment with standard practices in data analysis [20], we adopted a split ratio of 70% for training and 30% for testing.

Each instance in the dataset consists of a preprocessed comment and a corresponding binary label, indicating whether it belongs to the target TD type. Before training, the textual input was tokenized using the pretrained tokenizer associated with the selected language model. This step converts the raw text into model-compatible token sequences while maintaining the semantic and syntactic context of the original input. The resulting tokenized datasets are then passed to the model's training pipeline, where the model learns to distinguish between the target TD type and all other types.

Three transformer-based models were selected for experimentation: **BERT**[1], **BART**[2], and **GPT**[3]. The rationale for selecting these LLMs is described in Section 2.3.

The execution begins by loading each pretrained model along with its corresponding tokenizer using the **Hugging Face Transformers** library[4]. The tokenizer is used to convert the preprocessed text into input token sequences, suitable for each model's architecture. Each model was configured for binary classification, where the label 1 indicates the presence of the target TD type and 0 its absence. Along with that, we made a preliminary study to evaluate the models performance in training epochs. This study lead us to consider 8 epochs as the limit for the training pipeline. After this, the evaluation metrics did not show considerable improvements. Training was performed over 8 epochs on the 70% training portion of the balanced dataset, with a batch size of 8 for both training and evaluation. The models were evaluated at the end of each epoch, enabling the capture of performance metrics across training.

We assessed model performance using standard classification metrics: Precision, Recall, and F1-Score. These metrics were computed on the 30% test split for each TD type. Precision measures the proportion of correct positive predictions among all positive predictions made by the model. Recall quantifies the model's ability to detect actual instances of the target class. F1-Score provides a harmonic mean of precision and recall, which is particularly valuable in binary classification tasks involving potential class imbalance.

5 Results

5.1 RQ1: To what extent can LLMs correctly classify different types of TD on SO discussions?

The results in Table 2 show that LLMs exhibit varying effectiveness in classifying different types of TD within SO discussions. In general, models achieved high F1-scores for test and infrastructure TD types, up to 85% and 84%, respectively. This highlights their ability to capture patterns in well-defined or explicitly described technical contexts. These TD types often rely on recognizable keywords or structured problem statements, which LLMs can leverage. In contrast, for design type, performance dropped significantly across models, with F1-scores ranging from 59% to 67%. This suggests that design-related discussions, often more abstract and less formalized, pose a greater challenge for automated classification using LLMs.

When comparing overall averages, BART and BERT demonstrated similar performance across metrics, but BART achieved higher average precision (82.4% vs. 77.8%) and F1-Score (77.6% vs. 76.0%). This suggests that while both models are effective choices for TD identification on SO discussions, BART may offer an overall advantage in precision and recall across diverse TD types.

BERT and BART showed consistent performance across TD types, achieving strong recall for Code (99% and 100%) and Design (100% for both models), though sometimes at the cost of precision.

The drop in performance for Design debt underscores the need for richer representations to handle subtle, conceptual descriptions. This suggests that, while LLMs are promising tools for TD identification on platforms like SO, refinement, such as incorporating domain-specific context or fine-tuning with curated examples, will be essential to achieve reliable results.

The lower performance for Design debt is partly explained by its abstract and context-dependent nature, which often lacks the explicit technical markers present in other TD types (e.g., API names, configuration paths). This challenge is compounded by the small number of available instances (only 78 comments), limiting the model's exposure to varied patterns. Design comments, such as "We should restructure the module to improve maintainability", overlap semantically with Code or Architecture debt, leading to misclassifications. These findings suggest that richer context and a larger dataset will be essential for future improvements.

5.2 RQ2: How do LLMs compare to state-of-the-art approaches in classifying different TD types in SO discussions?

Table 3 demonstrates that the LLM-based approach proposed in this study g5enerally outperforms prior state-of-the-art methods in classifying TD types in SO discussions. When comparing the average precision, recall, and F1-Score across all TD types, our study outperforms the baselines, with all average metrics exceeding those of previous works by over 10%. This improvement reinforces the strength of our LLM-based approach in achieving more reliable performance for TD identification in SO discussions. For most TD types, Tests, Architecture, Infrastructure and Code, our models achieved consistently higher F1-scores, with gains of more than 15-20% in some cases compared to Gama et al. [12] and Kozanidis et al. [18]. These improvements suggest that leveraging large language models provides a meaningful advantage in capturing the diverse and context-rich language typical of developer discussions.

However, the results also highlight some variation across TD types. For Design type, while our approach achieved perfect recall (100%), precision was somewhat lower, yielding an F1-Score of 67%, in comparison to previous results. Despite this, for most other TD types, including Infrastructure and Test, our models achieved clearly superior precision and recall, with F1-Scores of 84% and 85%, respectively. Improvements in individual metrics for these TD types exceeded those of prior studies by 8–23%, underscoring substantial advantage of leveraging LLMs for TD identification in SO.

Overall, these findings highlight the potential of LLMs to advance automated TD identification by significantly outperforming existing approaches on most TD types. The improvements in recall are particularly valuable for real-world applications, where failing to detect TD can lead to overlooked maintenance risks. At the same time, the slight precision-recall trade-off observed for Design type suggests that further fine-tuning or hybrid methods may be necessary to handle more abstract or ambiguous TD discussions. These insights reinforce the promise of LLMs as powerful tools for supporting developers in identifying TD in platforms like SO.

5.3 RQ3: At which fine-tuning epochs do LLMs achieve peak performance for TD classification on SO discussions?

The evolution of F1-Score across fine-tuning epochs, represented in Figure 1, reveals important insights into how LLMs learn to classify

[1] *BERT Transformer: https://huggingface.co/google-bert/bert-base-uncased*
[2] *BART Transformer: https://huggingface.co/facebook/bart-base*
[3] *GPT Transformer: https://huggingface.co/openai-community/gpt2*
[4] *Hughing Face Transformers*: https://huggingface.co/docs/transformers/quicktour

Table 2: Comparison of results between LLM models

TD Type	Precision			Recall			F1-Score		
	BART	BERT	GPT	BART	BERT	GPT	BART	BERT	GPT
Tests	**98%**	94%	93%	80%	81%	**85%**	83%	**85%**	82%
Design	53%	55%	**75%**	**100%**	**100%**	60%	**67%**	66%	59%
Architecture	91%	**93%**	79%	87%	**91%**	79%	81%	**83%**	78%
Infrastructure	**99%**	79%	93%	**90%**	**90%**	88%	**84%**	82%	82%
Code	**71%**	68%	**71%**	99%	**100%**	83%	73%	64%	**74%**
Average	**82.4%**	77.8%	82.2%	91.2%	**92.4%**	79.0%	**77.6%**	76.0%	75.0%

Table 3: Comparison of results between this and other studies

TD Type	Precision			Recall			F1-Score		
	This Study	Gama et al (2023)	Kozanidis et al (2022)	This Study	Gama et al (2023)	Kozanidis et al (2022)	This Study	Gama et al (2023)	Kozanidis et al (2022)
Tests	**98%**	71%	71%	**85%**	**85%**	**85%**	**85%**	77%	77%
Design	75%	**78%**	**78%**	**100%**	61%	61%	67%	**68%**	**68%**
Architecture	**93%**	63%	63%	**91%**	70%	70%	**83%**	66%	66%
Infrastructure	**99%**	63%	63%	**90%**	58%	58%	**84%**	61%	61%
Code	**71%**	57%	57%	**100%**	63%	63%	**74%**	60%	60%
Average	**87.2%**	66.4%	66.4%	**93.2%**	67.4%	67.4%	**78.6%**	66.4%	66.4%

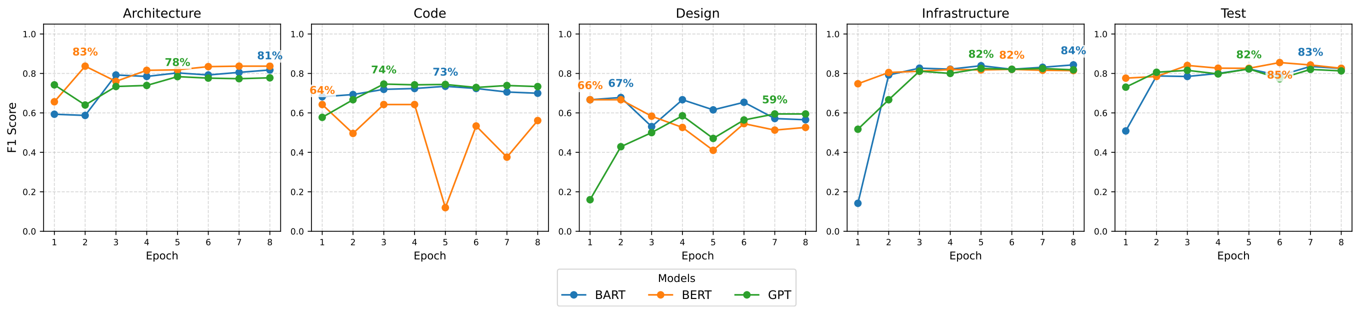

Figure 1: F1-score over training epochs for each TD type, comparing BART, BERT, and GPT models.

different TD types over time. For most TD types, the models show clear convergence within the first few epochs, with limited gains beyond epoch 4–5. For instance, in Architecture type, BERT rapidly peaks at 83% as early as epoch 2, while BART achieves its best performance (81%) only by epoch 8, suggesting different learning dynamics. GPT, meanwhile, demonstrates gradual convergence, stabilizing around 78%, which indicates consistent but lower performance. This pattern suggests benefits from fine-tuning, but optimal stopping varies by TD type.

Interestingly, the results expose notable differences in convergence behavior across TD types. Code type illustrates the instability of BERT, with severe drops in F1-score at later epochs, suggesting overfitting or sensitivity to noisy examples. In contrast, GPT and BART maintain smoother learning curves, with GPT reaching the highest F1-score (74%). For Infrastructure, all models show strong, early convergence around epoch 3, with BART achieving the top score of 84%, emphasizing that well-defined TD types with clear language cues benefit from shorter training. Conversely, Design type remains consistently challenging, staying in a F1-scores range around 66–67% for BART and BERT, while GPT lags at 59%, and learning curves indicate limited gains with further epochs, suggesting inherent ambiguity in design-related discussions that models struggle to capture regardless of training duration. The implication of these results is, while aggressive training (up to 8 epochs) can improve performance for certain TD types, many reach near-peak F1-scores after just 2–4 epochs, offering computational savings.

6 Conclusion

In this paper, we proposed an automated approach for identifying TD in SO discussions using LLMs. Our approach leverages transformer-based models, BERT, BART, and GPT, to detect TD in developer conversations across types including Code, Infrastructure, Architecture, Test, and Design. The results demonstrated strong performance, with F1-scores ranging from approximately 59% to 85%. Also, all average precision, recall, and F1-score metrics achieved in this study surpassed those reported in previous state-of-the-art methods, highlighting the effectiveness of LLMs in capturing the rich and varied language typical of developer discussions.

Our analysis revealed that while LLMs delivered excellent results in identifying certain types of TD, such as Test and Infrastructure type, they showed more limited effectiveness with design-related debt, likely due to the greater semantic ambiguity of such discussions. Additionally, our experiments indicated that peak model performance was typically reached within the first few fine-tuning epochs, highlighting opportunities to reduce training costs without sacrificing accuracy. These findings support the feasibility of applying LLMs to automate the identification of TD in large-scale developer forums, providing an efficient and replicable method to support software quality monitoring.

Although achieving promising results, a larger dataset could improve the results obtained in this study, specially for TD types that didn't achieve high results, such as Design. As future work, we plan to expand the dataset with additional discussions to provide richer coverage of TD-related contexts. We also intend to explore prompt engineering techniques to better guide LLM predictions, as well as investigate advanced training strategies and larger-scale pretraining data to enhance performance.

REFERENCES

[1] William Aiken, Paul K. Mvula, Paula Branco, Guy-Vincent Jourdan, Mehrdad Sabetzadeh, and Herna Viktor. 2023. Measuring Improvement of F1-Scores in Detection of Self-Admitted Technical Debt . IEEE Computer Society.

[2] Newton S. R. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman. 2016. Identification and management of technical debt: A systematic mapping study. *Information and Software Technology* (2016).

[3] Lucas Amaral. 2025. Replication package: "Evaluating Large Language Models on the Classification of Different Technical Debt Types in StackOverflow Discussions". https://anonymous.4open.science/r/Replication_Kit_ISE_2025-BC60/

[4] Alan Bandeira, Carlos Alberto Medeiros, Matheus Paixao, and Paulo Henrique Maia. 2019. We Need to Talk about Microservices: an Analysis from the Discussions on StackOverflow. *MSR* (2019).

[5] Anton Barua, Stephen W Thomas, and Ahmed E Hassan. 2014. What are developers talking about? an analysis of topics and trends in stack overflow. *Empirical software engineering* (2014).

[6] Nathan Brown, Yuanfang Cai, Yanyan Guo, Rick Kazman, Michael Kim, Philippe Kruchten, Eun-Young Lim, Alan MacCormack, Robert Nord, and Ipek Ozkaya. 2010. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of Software Engineering Research*.

[7] Weijia Chen et al. 2023. Code Llama: Open Foundation Models for Code. https://ai.meta.com/research/publications/code-llama-open-foundation-models-for-code/.

[8] Ward Cunningham. 1992. The WyCash Portfolio Management System. In *Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum)*.

[9] Ward Cunningham. 1992. The WyCash portfolio management system. *ACM Sigplan Oops Messenger* (1992).

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*.

[11] Joshua Aldrich Edbert, Sahrima Jannat Oishwee, Shubhashis Karmakar, Zadia Codabux, and Roberto Verdecchia. 2023. Exploring Technical Debt in Security Questions on Stack Overflow. *arXiv preprint arXiv:2307.11387* (2023).

[12] Eliakim Gama, Mariela I. Cortés, Matheus Paixao, and Adson Damasceno. 2023. Machine Learning for the Identification and Classification of Technical Debt Types on StackOverflow Discussions. In *Brazilian Workshop on Intelligent Software Engineering (ISE)*.

[13] Eliakim Gama, Sávio Freire, Manoel Mendonça, Rodrigo O. Spínola, Matheus Paixao, and Mariela I. Cortés. 2020. Using Stack Overflow to Assess Technical Debt Identification on Software Projects. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*.

[14] Eliakim Gama, Sávio Freire, Manoel Mendonça, Rodrigo O. Spínola, Matheus Paixão, and Mariela I. Cortés. 2020. Using Stack Overflow to Assess Technical Debt Identification on Software Projects. In *Proceedings of the 34th Brazilian Symposium on Software Engineering (SBES)*.

[15] Eliakim Gama, Matheus Paixao, Emmanuel Sávio Silva Freire, and Mariela Inés Cortés. 2019. Technical Debt's State of Practice on Stack Overflow: A Preliminary Study. In *Proceedings of the XVIII Brazilian Symposium on Software Quality*.

[16] Bhawna Jain, Gunika Goyal, and Mehak Sharma. 2024. Evaluating Emotional Detection Classification Capabilities of GPT-2 GPT-Neo Using Textual Data. In *2024 14th International Conference on Cloud Computing, Data Science Engineering (Confluence)*.

[17] Nicholas Kozanidis, Roberto Verdecchia, and Emitza Guzman. 2022. Asking about Technical Debt: Characteristics and Automatic Identification of Technical Debt Questions on Stack Overflow. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*.

[18] Nicholas Kozanidis, Roberto Verdecchia, and Emitzá Guzmán. 2022. Asking about Technical Debt Characteristics and Automatic Identification of Technical Debt Questions on Stack Overflow. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*.

[19] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. 2012. Technical debt: From metaphor to theory and practice. *IEEE Software* (2012).

[20] Max Kuhn and Kjell Johnson. 2013. Applied predictive modeling.

[21] Pedro Lambert, Lucila Ishitani, and Laerte Xavier. 2024. On the Identification of Self-Admitted Technical Debt with Large Language Models. In *Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software*. SBC.

[22] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461* (2019).

[23] Zengyang Li, Paris Avgeriou, and Peng Liang. 2015. A systematic mapping study on technical debt and its management. *Journal of Systems and Software* (2015).

[24] Erin Lim, Nitin Taksande, and Carolyn Seaman. 2012. A balancing act: What software practitioners have to say about technical debt. *IEEE software* (2012).

[25] Sarah Meldrum, Sherlock A Licorish, and Bastin Tony Roy Savarimuthu. 2017. Crowdsourced knowledge on stack overflow: A systematic mapping study. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*.

[26] Nicolli Rios, Manoel Gomes de Mendonça Neto, and Rodrigo Oliveira Spínola. 2018. A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. *Information and Software Technology* (2018).

[27] Mohammad Sadegh Sheikhaei, Yuan Tian, Shaowei Wang, and Bowen Xu. 2024. An Empirical Study on the Effectiveness of Large Language Models for SATD Identification and Classification. *arXiv preprint arXiv:2405.06806* (2024).

[28] Peeradon Sukkasem, Chitsutha Soomlek, and Chanon Dechsupa. 2025. Llm-Based Code Comment Summarization: Efficacy Evaluation and Challenges. In *2025 17th International Conference on Knowledge and Smart Technology (KST)*.

[29] Mark Swillus and Andy Zaidman. 2023. Sentiment overflow in the testing stack: Analyzing software testing posts on Stack Overflow. *Journal of Systems and Software* (2023).

[30] Amjed Tahir, Aiko Yamashita, Sherlock Licorish, Jens Dietrich, and Steve Counsell. 2018. Can you tell me if it smells? a study on how developers discuss code smells and anti-patterns in stack overflow. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*.

[31] Edith Tom, Aybüke Aurum, and Richard Vidgen. 2013. An exploration of technical debt. *Journal of Systems and Software* (2013).

[32] Michele Tufano, Cody Watson, Gustavo White-Martins, and Denys Poshyvanyk. 2023. An Empirical Study on the Use of Large Language Models for Code-related Tasks. *Empirical Software Engineering* (2023).

[33] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*.

[34] Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E Hassan, and Zhenchang Xing. 2017. What do developers search for on the web? *Empirical Software Engineering* (2017).

[35] Yaqin Zhang, Xiang Ren, Shihan Wang, Ziyue Chen, Hongyu Zhang, and David Lo. 2023. A Survey of Large Language Models in Software Engineering. *arXiv preprint arXiv:2307.04743* (2023).