

Prescriptive and Contextual Technical Debt Management with LLM and SonarQube

Abstract—Technical Debt (TD) is a short-term technical decision that compromises long-term quality and stability. Static analysis tools, such as SonarQube, can identify TD but only provide diagnostic analysis without prescriptive recommendations or project-specific context. However, prescriptive and contextual recommendations are essential because the conditions and priorities of each project are different. This research aims to develop an LLM-based TD management system integrated with SonarQube and GitHub to provide prescriptive and contextual analysis. Evaluation is carried out using the Rule2Text LLM-as-a-judge framework to measure the prescriptive quality of the system’s output. We found that LLM shows a significant improvement from SonarQube and can reliably provide prescriptive analysis, but falls a bit short contextually as it tends to hallucinate.

Index Terms—LLM, technical debt management, SonarQube, software maintenance, continuous integration

I. INTRODUCTION

Technical debt (TD) arises when teams take sub-optimal shortcuts that speed delivery but increase future costs, the “interest” paid through rework, schedule slippage, and reduced business value [1]–[3]. To manage TD, practitioners rely on static analysis tools, backlogs, coding standards, and documentation [4], [5]. These approaches are valid but essentially diagnostic. SonarQube, one of the leading static analysis tools, has proven to flag issues without explaining why they matter in this context or what concrete action best suits the project’s needs [6]. SonarQube’s findings typically stop at “what,” leaving developers to infer the why and how, which slows remediation and weakens prioritisation [6]. Recent work in TD calls for analyses that are not only diagnostic but also contextual and prescriptive, grounded in project realities and accompanied by actionable guidance [7]–[9].

The term “contextual” means that TD management varies across different projects and companies [8]. While “prescriptive” means there is justification and the ability to explain the effects of the detected TD [6]. Prescriptive and contextual analyses would also help data-driven decision making, increasing trust and transparency with stakeholders [6]. To fulfill these prescriptive and contextual requirements of TD management, we leverage recent advancements in artificial intelligence, specifically Large Language Models (LLMs). LLM offers a promising avenue for exploration, especially in the Software Engineering field. Models such as GPT-4 can understand code context, explain code, and give adaptive refactoring suggestions based [10]. Furthermore, GPT-4 is one of the most recent LLMs from OpenAI that shows high performance in reasoning and mathematics [11].

Identify applicable funding agency here. If none, delete this.

We propose an LLM-centric approach using SonarQube, integrated with GitHub, to manage TD at the pull request (PR) level. Leveraging LLM’s high reasoning and code understanding capabilities, we intend to integrate LLM’s analysis directly into the software development cycle as a PR comment. When developers intend to push an update, the LLM would provide a precise analysis of potential TD(s) that would arise if the changes are committed. Highlighting the severity, short and long-term effects, and suggested action on each TD issue. We raise these research questions (RQs) to evaluate our approach:

- **RQ1:** To what extent do LLM-based systems provide contextually relevant explanations for SonarQube-detected TD?
- **RQ2:** How effective are LLM-based systems in providing prescriptive guidance for addressing TD?

RQ1 and RQ2 are answered using a separate LLM-as-Judge module that evaluates the agent’s output Contextual and Prescriptive scores computed on PR-level analyses.

This study contributes: (1) an end-to-end, CI-friendly pipeline that augments SonarCloud findings with LLM-generated, PR-scoped explanations and remedies; and (2) implementation artefacts, including GitHub Action workflow, prompts, and evaluation harnesses to enable replication.

The remainder of this paper is structured as follows: a survey of related works on TD management and the tools proposed by previous works. The proposed approach is described in detail, from the individual parts of the system to the evaluation design. The result would then be presented and analysed further, discussing the implications and answers for RQs. The discussion also follows the limitations we faced during our approach. Finally, a conclusion is drawn from our discussion, highlighting the results of our research and calling for further study in our area with specific directions.

II. RELATED WORK

TD management is a primary concern at the PR level. Introducing TD in a PR increases the risk of that PR being rejected [12]. At the same time, Karmakar et al. found that nearly 40% of comments in PR are related to identifying or debating TD [13]. Calixto et al. revealed that TD is neglected in approximately 75% of cases, either remaining unchanged or increasing [14]. Developers struggle to identify and address TD effectively during the manual code review process because the discussions tend to be unstructured and lack context [12].

Recent works have produced several tools and solutions for managing code TD. Martini et al. developed Anacondet, a

tool that tracks and evaluates TD through systematic visualisation and refactoring plans [15]. Mendes et al. introduced VisMinerTD, which identifies TD, code smells, defects, and more within a project [16]. Dornauer et al. noticed that while SonarQube is effective in detecting TD and code smells, it cannot prioritise debt based on project impact or explain when and why certain debts must be addressed or deferred [17]. This limitation can lead to inefficiencies in managing TD. To address this, they proposed SoHist, a tool designed to overcome SonarQube’s weaknesses. Tsoukalas et al. emphasised that while these tools are effective at identifying TD, they still fall short in providing explanations of causes, impacts, and strategies for contextual resolution [6]. Even efforts like SoHist do not fully address the need for reasoning and recommendations.

Unlike prior tools that emphasise detection, visualisation, or history tracking of TD, our contribution is a CI-centred framework that augments SonarQube results with LLM-driven, *contextual and prescriptive* explanations at the PR level. This closes the gap between purely diagnostic findings and actionable remediation steps within developers’ existing workflows. Our work extends the previously mentioned approaches by implementing an LLM-based method for TD management. This is important because there is a need for a method to provide not only a diagnostic solution to TD but also a prescriptive solution to TD [17].

III. PROPOSED APPROACH

This section provides a detailed explanation of our LLM-centric system.

A. System Workflow Overview

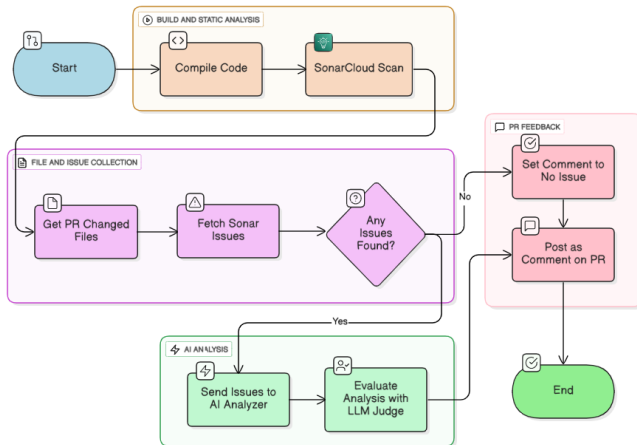


Fig. 1. The full integration flow with GitHub, SonarCloud, and AI Analyser agent.

Figure 1 details our workflow in great detail. The workflow begins when a developer opens or updates a PR on GitHub. This event acts as the trigger for the automated analysis pipeline. After a successful build, the code undergoes static

analysis using SonarCloud, a cloud-hosted version of SonarQube. Then, the system fetches changed files from the most recent commit, along with Sonar issues from SonarCloud. If no problems are present, the flow will end with a message. Otherwise, the issues and file changes are sent directly to the AI agent analyzer via input message. Another AI agent would then judge the analysis, and the final result will be sent as a PR comment.

B. Build and Static Analysis

The trigger will follow up on another process of compiling and scanning the code. Firstly, we set up the environment on our GitHub Action. Secondly, we prepare to scan with SonarCloud. Compiling the code is required before any SonarCloud scan, so we do so while skipping tests for simplicity. Finally, we initiate the scan to find sonar issues that would be analysed as potential TD.

C. File and Issue Collection

We intend to leverage GPT-4’s code understanding capabilities. Therefore, we need to inject some of the troubled code snippets into the AI analyser agent. For this, we need to get the changed files within that PR. Firstly, we list the changed files for this PR. Then, we build a list of files mentioned by the Sonar issues. Secondly, we find the intersection of changed files in PR and in the Sonar issues. Finally, the intersect will list the changed files that would then be sent to another process. After we get the changed files, we set up another action to fetch Sonar issues and their code snippets in one block. We set up configurable max issues, max issues per rule, sonar severities, etc., to limit or extend the sonar issues fetched from the previous scan. Secondly, we iterate the Sonar issues through SonarCloud’s API, appending them to a JSON file. We enforce per-rule and per-file caps, then trim the issues to the previously mentioned caps. Finally, we build the compact snippets with a tiny context and a character cap, then we output them. After this process is done, there are two ways this could go. If no sonar issues are found, we skip the AI analyser agent and comment that there is no potential TD in the current PR. Otherwise, it would be sent to the AI analyser agent to address whether there are TD(s) and what their effects are.

D. AI Analysis

We send the previous snippet to the AI analyser agent through Flowise’s prediction API. Firstly, we set up another configurable payload settings, such as max summary chars, max question chars, max payload bytes, etc. Secondly, we format the snippet into a more readable markdown summary and set up the question text, instructing the agent to analyse Sonar Cloud issues from the current PR number in this repository, targeting this PR title and message. Finally, we create the payload, which consists of snippets of rules, severity, location, and message for each Sonar issue. Then we send it through Flowise’s API to the analyzer agent Agentflow.

This Agentflow consists of the start node and the AI Agent node. We set up the prompt using a persona prompt pattern

[18]. We let the Agent use GitHub MCP, which it can use to search for code, issues, etc., in a repository. We also set up its knowledge base using Flowise’s document store. The knowledge base contains Jira issues from the public Jira board. We then upsert it to Pinecone Vector DB and use Supabase to act as the record manager. The judge agent also follows this same set-up, although using a Flowise chatflow instead. This setup aims to allow the AI agent to maximise the quality of the analysis generated.

After the analysis has finished generating, we also incorporated the LLM judge agent in this part. We load the analysis output from the previous step, reuse the sonar summary text for context, and build the judge prompt, which essentially instructs the judge agent to evaluate the LLM’s output and return only JSON. The judge’s output would be appended to the PR comment.

E. PR Feedback

This is the final part of the flow where the final payload is sent out as a PR comment. If the sonar scan doesn’t find any issue, this comment will state that there are no potential TDs in the current PR. Otherwise, it will take the previous AI analyzer agent’s analysis and append it with the judge’s output as a comment. The final comment would include the TD issue number, issue description, including the sonar rule, affected code, is a technical debt field, justifications, rationale, TD severity, long and short-term effects, and suggested actions. The judge’s evaluation details the final scores for both contextual and prescriptive aspects. Figure 2 shows the sample SonarQube’s message that is used as the input, while figure 3 shows the sample output of our system. It generates an analysis from the Sonar issue to explain TD’s effect in great detail.

IV. VALIDATION

This section will provide an explanation of our validation system as well as its results.

A. Validation Overview

Figure 4 shows the overview of our validation process. We first extract TD issues from the Technical Debt Dataset V2.0 [19]. We extract TD issues created between 2015 and 2022, querying the dataset. For each TD issue, we run the system that generates AI analysis. If the sonar scan did not detect any problem, or if the AI analysis deemed that there are no TD available, then further investigation is unnecessary. We marked the said issue as no TD and skipped it. Otherwise, the analysis generated by the AI analyser agent will be passed to the judge agent for evaluation. The judge agent will evaluate

```

/** Marks a constant pool entry as a Module Reference.
 * Base: http://code.adoptopenjdk.java.net/~m/ijgson/spec/Lang-vm.html#ijgson-2.6
 * JRS: Modules in the Java Language and JVM/CP
 * Note: Early access Java 9 support- currently subject to change */
public static final byte CONSTANT_Module = 19;

Rename this constant name to match the regular expression "[A-Z][A-Z0-9]*([A-Z0-9]+)?$".

```

Fig. 2. Sample of the PR feedback input from the SonarCloud that the AI analyzer agent would analyze.

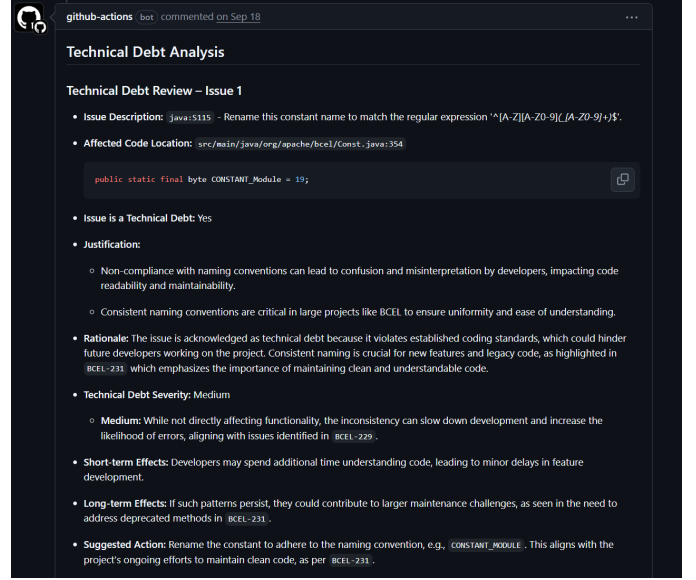


Fig. 3. Sample of the PR feedback, analysis of potential TD by the AI analyzer agent.

the analysis’s prescriptive and contextual criteria and assign a final score ranging from one to five, one being very poor and five being very good. Finally, we record the scores of each TD item and visualise them in our paper.

B. The Technical Debt Dataset

We use the Technical Debt Dataset, Version 2.0, for our case study and experiments [19]. This dataset contains TD issues that are linked with GitHub commits and SonarQube. There are a total of 31 Apache projects, each with a different number of TD issues found. We randomly picked two projects for our experimentation: Apache Commons Collections and Apache Commons BCEL. Each of them contains 55 and 53 TD issues, respectively. We also assigned a time range of 2015 - 2022, from which TD issues were found. This generates a fault-inducing commit hash, its parent commit hash, the author’s name, and the commit message for our validation

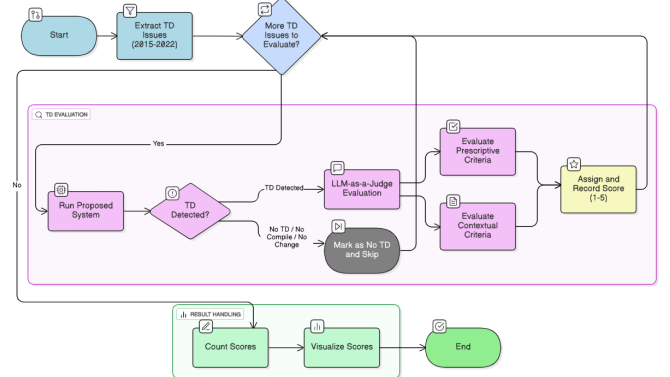


Fig. 4. The overview of how the evaluation is handled using the LLM-as-a-judge framework.

process. These issues would then be used as the basis for the experiment, where the judge agent would evaluate the AI analysis.

We found that the three most common rules that were identified as TDs were squid: S1166, squid: S1213, and squid: S00117. Therefore, we selected the three issues in table I that align with the most common rules listed in this paper as a reference for the types of TD issues examined in this research. Interestingly enough, out of all 25 issues, only three matched the most common rules. This means that the most frequent TD is not necessarily the most impactful.

C. TD Analysis Evaluation with LLM-as-a-judge

We evaluate the analysis produced by the analyzer agent using an LLM-as-a-judge framework. This framework would allow scalable and human-free evaluation, which correlates with human assessment on reasoning and instruction-following tasks, whilst providing logical scores to the analysis [20]–[22]. We implemented the judge agent within Flowise’s chatflow. The judge agent has the same project’s context to evaluate its contextual score, but using a different LLM base model. As specified in Rule2Text, this evaluation method is considerably biased [20]. To minimise this bias, we used a different LLM model, specifically Gemini 2.5 Flash. To further reduce bias and overconfidence, we enforce JSON-only outputs and evaluate per criterion. We follow the Rule2Text framework to create our own LLM-as-a-judge, using Gemini as a base model, although it is an older Gemini 2.0 Flash. First, we make our criterion utilising a chain of thought prompt pattern for the judge agent.

Step 1: Verify Contextual Analysis

- (C1) Cites a relevant Jira key from the project context? (Yes/No)
- (C2) Provides a logical rationale linked to project goals? (Yes/No)
- (C3) Specifies a technical debt severity? (Yes/No)
- (C4) Describes contextual short- and long-term effects? (Yes/No)

Step 2: Verify Prescriptive Depth

- (P1) Explains short- and long-term effects of not fixing? (Yes/No)
- (P2) Are effects compelling and specific? (Yes/No)
- (P3) Suggests what to do if the issue is identified as technical debt? (Yes/No)

Step 3: Verify Quality of Generation

- (Q1) Is the analysis clear and well-written? (Yes/No)
- (Q2) Includes all required structural components? (Yes/No)

Step 4: Final Scoring

- *Contextual Score*: 1–5 based on rules C1–C4.
- *Prescriptive Score*: 1–5 based on rules P1–P3.

Secondly, we integrate each judge agent of each project with its own knowledge base, similar to the analyzer agent’s

knowledge. Finally, for each issue we previously extracted, we run the judge agent after the analyzer agent has finished analyzing. This judge agent also follows previous studies, where the result of the LLM-as-a-judge evaluation is a Likert scale of 1 to 5 [20], [21].

Figure 5 shows a comparison in the contextual score of our LLM-centric system and SonarQube’s default message against the judge agent. SonarQube’s default messages show a mostly very poor performance, with 88% falling on the poor/very poor scores and 12% falling to the fair score. In contrast, our LLM-centric approach to contextual TD management generates somewhat good results. The results show 64% of scores were good / very good, 28% were fair, and 8% were poor. This shows a significant increase in performance from the default SonarQube messages to our LLM-based analysis. However, this score also indicates that LLM can hardly provide consistent, reliable contextual explanations, usually suffering from hallucination.

Figure 6 details the prescriptive scores of our system’s analysis and SonarQube’s messages. We found a poor performance for SonarQube, with 100% of scores falling into poor/very poor categories. Our LLM-based analysis performed excellently, achieving 96% very good scores and 4% good scores. This highlights a significant improvement that our system generates for TD management from SonarQube’s messages. LLM demonstrates its ability to create consistent prescriptive analysis in managing TD, guiding developers on how to handle TDs.

V. DISCUSSION

This section will discuss our findings and their correlation to our research questions.

RQ1. To what extent do LLM-based systems provide contextually relevant explanations for SonarQube-detected TD?

The contextual scores we found in figure 5 are generally high, but some lower values pull the mean down. Contextual explanations were considered good but slightly less consistent

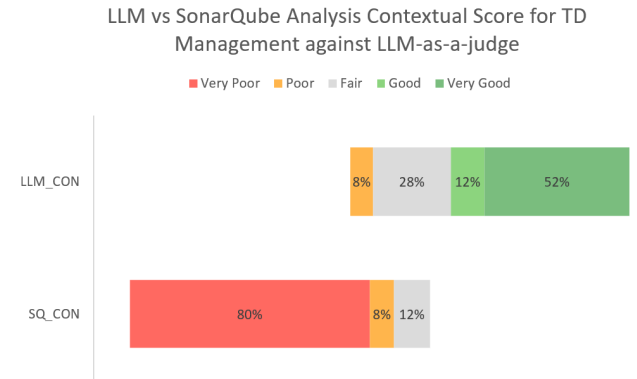


Fig. 5. AI analyzer agent VS SonarQube Analysis Contextual Score against LLM-as-a-judge. 64% of LLM outputs were rated good/very good; 0% of SonarQube outputs fell into good/very good.

TABLE I
EXAMPLE OF FAULT-INDUCING COMMITS AND THEIR PARENT COMMITS. NAMES ARE NOT SHOWN FOR PRIVACY.

Fault Inducing Commit Hash	Fault Inducing Commit Parent Hash	Commit Message
7a2...7103	e05...51fe	Use final on local vars when possible (which already do on fields and parameters when possible.) git-svn-id: https://svn.apache.org/repos/asf/commons/proper/bcel/trunk@1749603 13f79535-47bb-0310-9956-ffa450edef68
feb...8b40	f3f...4a17	BCEL-221 BCELifier is not working for Java8Example Workround for crash; may need further work git-svn-id:https://svn.apache.org/repos/asf/commons/proper/bcel/trunk@1702447 13f79535-47bb-0310-9956-ffa450edef68
02f...28c9	bad...ccad	Gradually working towards restoring binary compatibility Revert Constants back to interface git-svn-id: https://svn.apache.org/repos/asf/commons/proper/bcel/trunk@1702355 13f79535-47bb-0310-9956-ffa450edef68

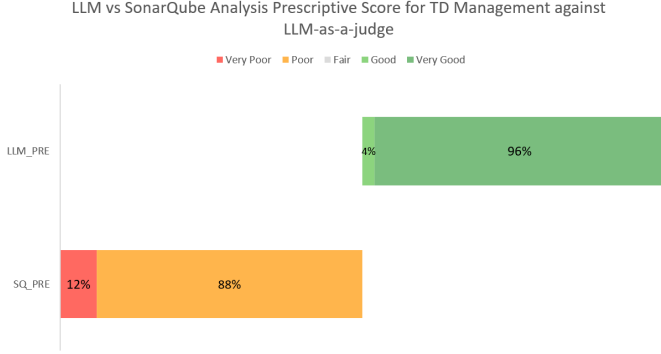


Fig. 6. AI analyzer agent VS SonarQube Analysis Prescriptive Score against LLM-as-a-judge. 100% of LLM outputs were good/very good; 0% of SonarQube outputs fell into good/very good.

than prescriptive ones. Furthermore, LLM offers a substantial increase in contextually relevant TD management compared to the base SonarQube messages. Contextual scores also provide more variation, as some instances were identified where the judge agent saw missing context in the explanations. LLM can provide contextually relevant explanations to a great degree, but it is not perfect. This highlights potential areas where LLM contextual reasoning could be improved, such as project-specific requirements, deadlines, or more references to Jira issues.

RQ2. How effective are LLM-based systems in providing prescriptive guidance for addressing TD?

Almost all prescriptive scores we found in figure 6 are five, indicating that the judge agent consistently found the LLM’s prescriptive analysis highly effective in guiding the addressing of TD. Compared to SonarQube, LLM provides a significant score improvement. Furthermore, we found that the judge agents mostly agree, with minimal variation in their judgments for all prescriptive scores. This means that the LLM is highly effective at generating prescriptive recommendations. It reliably informs developers on what to do, aligning with our goal of making TD handling actionable.

The results that we found are interesting. While the LLM-centric system works perfectly for prescriptive analysis, it shows a significant improvement in score against SonarQube’s messages. Contextually, the AI analyzer system might produce mixed-quality analysis. Some analysis may correctly reference

a logical Jira issue from the knowledge base. While other times, it hallucinates and cites its own non-existing Jira issue, or even doesn’t cite one at all. At the same time, the scores show that the AI analyzer agent can provide good, relevant contextual analysis. Thus, this approach may have improved SonarQube’s abilities to manage TD, but further work is needed to solidify our contextual management of TD.

We faced several hallucination issues when experimenting on the dataset. Thus, we also implemented several mitigation attempts to address this: **F1: Weak citation to project context.** Sometimes the analyzer mentions issues without explicit IDs. *Mitigation:* enforce a “must include BCEL-XXX/Collections-XXX” rule in prompts and filter responses lacking IDs. **F2: Over-general advice.** Fix suggestions too generic. *Mitigation:* require “file: line” references and one concrete diff-style suggestion when applicable.

We also face a challenge due to OpenAI’s token limit, which restricts the number of Sonar issues, code snippets, and additional context that can be sent to the agents. This, in particular, could directly impact the outcome of the analysis in a contextual sense. Furthermore, not all extracted issues from the dataset can be detected as Sonar issues by SonarQube, and not all matters extracted from the dataset actually contain code changes; some detected fault-inducing commits only show changes.xml, and we only experimented with issues that have code changes in the codebase and are detected as an issue by SonarCloud.

A. Threats to Validity

We consider two main threats:

External Validity: Refers to the generalizability of our results. Experiments were conducted on only two Apache Java projects, Commons Collections and Commons BCEL; the results may not apply to other software ecosystems or projects. This is because we are limited in both time and cost. Broader validation across different projects, larger TD issues, and even different programming languages would improve confidence in the generalizability of our findings. Future work may expand the experiments to include a larger set of projects and issues to validate our approach to TD management further.

Conclusion Validity: Threats that affect the reliability of inferences. Our study did not include human evaluation and samples due to limited time and resources. The results we

found may not fully represent the spectrum of issues or judgments. Future studies could address this issue by incorporating human evaluation as a base reference for the LLM-as-a-judge evaluation, solidifying the validation of our research.

VI. DATA AVAILABILITY

The artefact collected in this research, such as the dataset, Flowise’s flow, prompt templates, etc., will be available at this link: **[the link is hidden for review process]**

VII. CONCLUSION

In this study, we propose an LLM-centric approach to managing technical debt at the PR level and time, utilising GitHub, SonarCloud, and Flowise for its implementation. By implementing an LLM-as-a-judge framework for our evaluation, we found that LLM proves to be very effective in generating prescriptive recommendations. LLM can also provide contextual explanations related to the project, but only to a limited degree. Hallucinations still prove to be a challenging aspect in generating contextually relevant explanations for TD management. This research contributes to providing a prescriptive and contextual solution in TD management, albeit with some limitations in contextually relevant explanations.

There are still aspects of our research that can be expanded in the future. First, expanding the evaluation to include a larger set of open-source projects and multiple programming languages would enhance the generalizability of the LLM-based TD analysis. Secondly, integrating human expert validation alongside the LLM-as-a-Judge framework could provide a more robust benchmark for prescriptive and contextual scoring, improving the reliability of automated recommendations. Third, exploring real-time integration within CI/CD pipelines could enable the automated prioritisation and remediation of TD, thereby linking LLM recommendations directly to SonarQube quality gates or automated refactoring scripts. Furthermore, improving the RAG pipeline and fine tuning its indexing parameters may provide a more contextually relevant solutions for TD management. Finally, addressing limitations such as LLM hallucination, prompt sensitivity, and token efficiency through advanced prompt engineering or human-in-the-loop approaches with LLMs could further improve the precision and applicability of prescriptive guidance in software maintenance.

REFERENCES

- [1] Z. Codabux, B. J. Williams, G. L. Bradshaw, and M. Cantor, “An empirical assessment of technical debt practices in industry,” *Journal of software: Evolution and Process*, vol. 29, no. 10, p. e1894, 2017.
- [2] H. Kleinwaks, A. Batchelor, and T. H. Bradley, “Predicting the dynamics of earned value creation in the presence of technical debt,” *IEEE Access*, vol. 11, pp. 125381–125395, 2023.
- [3] R. Banker, Y. Liang, and N. Ramasubbu, “Technical debt and firm performance,” *Management Science*, vol. 67, no. 5, pp. 3174–3194, 2021.
- [4] J. A. Ruiz, R. A. Aguilar, J. F. Garcilazo, and A. A. Aguilera, “A tertiary study on technical debt management over the last lustrum,” *IJCOPI*, vol. 15, no. 5, p. 181, 2024.
- [5] A. Martini, T. Besker, and J. Bosch, “Technical debt tracking: Current state of practice: A survey and multiple case study in 15 large organizations,” *Science of Computer Programming*, vol. 163, pp. 42–61, 2018.
- [6] D. Tsoukalas, N. Mittas, E.-M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, and D. Kechagias, “Local and global explainability for technical debt identification,” *IEEE Transactions on Software Engineering*, 2024.
- [7] P. Avgeriou, I. Ozkaya, A. Chatzigeorgiou, M. Ciolkowski, N. A. Ernst, R. J. Koontz, E. Poort, and F. Shull, “Technical debt management: The road ahead for successful software delivery,” in *2023 IEEE/ACM ICSE-FoSE*, pp. 15–30, IEEE, 2023.
- [8] Z. Codabux and B. J. Williams, “Technical debt prioritization using predictive analytics,” in *Proceedings of the 38th ICSE-Companion*, pp. 704–706, 2016.
- [9] W. N. Behutiye, P. Rodríguez, M. Oivo, and A. Tosun, “Analyzing the concept of technical debt in the context of agile software development: A systematic literature review,” *Information and Software Technology*, vol. 82, pp. 139–158, 2017.
- [10] L. Belzner, T. Gabor, and M. Wirsing, “Large language model assisted software engineering: prospects, challenges, and a case study,” in *AISeLA*, pp. 355–374, Springer, 2023.
- [11] A. Zhou, K. Wang, Z. Lu, W. Shi, S. Luo, Z. Qin, S. Lu, A. Jia, L. Song, M. Zhan, *et al.*, “Solving challenging math word problems using gpt-4 code interpreter with code-based self-verification,” *arXiv preprint arXiv:2308.07921*, 2023.
- [12] M. C. O. Silva, M. T. Valente, and R. Terra, “Does technical debt lead to the rejection of pull requests?,” *arXiv preprint arXiv:1604.01450*, 2016.
- [13] S. Karmakar, Z. Codabux, and M. Vidoni, “An experience report on technical debt in pull requests: Challenges and lessons learned,” in *Proceedings of the 16th ACM/IEEE ESEM*, pp. 295–300, 2022.
- [14] F. E. d. O. Calixto, E. C. Araújo, and E. L. Alves, “How does technical debt evolve within pull requests? an empirical study with apache projects,” in *Simpósio Brasileiro de Engenharia de Software (SBES)*, pp. 212–223, SBC, 2024.
- [15] A. Martini, “Anacondebt: A tool to assess and track technical debt,” in *Proceedings of the 2018 International Conference on Technical Debt*, pp. 55–56, 2018.
- [16] T. S. Mendes, D. A. Almeida, N. S. Alves, R. O. Spínola, R. Novais, and M. Mendonça, “Visminertd: an open source tool to support the monitoring of the technical debt evolution using software visualization,” in *ICEIS*, 2015.
- [17] B. Dornauer, M. Felderer, J. Weinzerl, M.-C. Racasan, and M. Hess, “Sohist: A tool for managing technical debt through retro perspective code analysis,” in *Proceedings of the 27th EASE*, pp. 184–187, 2023.
- [18] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt, “Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design,” in *Generative ai for effective software development*, pp. 71–108, Springer, 2024.
- [19] V. Lenarduzzi, N. Saarimäki, and D. Taibi, “The technical debt dataset,” in *15th Conference on PROMISE*, January 2019.
- [20] N. Shirvani-Mahdavi and C. Li, “Rule2text: A framework for generating and evaluating natural language explanations of knowledge graph rules,” *arXiv preprint arXiv:2508.10971*, 2025.
- [21] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing, *et al.*, “Judging llm-as-a-judge with mt-bench and chatbot arena,” *Advances in neural information processing systems*, vol. 36, pp. 46595–46623, 2023.
- [22] A. Szymanski, N. Ziems, H. A. Eicher-Miller, T. J.-J. Li, M. Jiang, and R. A. Metoyer, “Limitations of the llm-as-a-judge approach for evaluating llm outputs in expert knowledge tasks,” in *Proceedings of the 30th IUI*, pp. 952–966, 2025.