PromptDebt: A Comprehensive Study of Technical Debt Across LLM Projects

Ahmed Aljohani ahmedaljohani@unt.edu University of North Texas, TX, USA Hyunsook Do hyunsook.do@unt.edu University of North Texas, TX, USA

Abstract

Large Language Models (LLMs) are increasingly embedded in software via APIs like OpenAI, offering powerful AI features without heavy infrastructure. Yet these integrations bring their own form of self-admitted technical debt (SATD). In this paper, we present the first large-scale empirical study of LLM-specific SATD: its origins, prevalence, and mitigation strategies. By analyzing 93,142 Python files across major LLM APIs, we found that 54.49% of SATD instances stem from OpenAI integrations and 12.35% from LangChain use. Prompt design emerged as the primary source of LLM-specific SATD, with 6.61% of debt related to prompt configuration and optimization issues, followed by hyperparameter tuning and LLM-framework integration. We further explored which prompt techniques attract the most debt, revealing that instructionbased prompts (38.60%) and few-shot prompts (18.13%) are particularly vulnerable due to their dependence on instruction clarity and example quality. Finally, we release a comprehensive SATD dataset to support reproducibility and offer practical guidance for managing technical debt in LLM-powered systems.

Keywords

Self-admitted technical debt (SATD), Large language models (LLMs), Prompt Engineering

1 Introduction

Large Language Models (LLMs) have rapidly become integral to modern software development [20], enabling developers to perform a wide range of downstream tasks, including text summarization, conversational agents, and advanced automation [11, 17]. To cope with the substantial cost and complexity of training and maintaining in-house LLMs, developers increasingly rely on APIs from companies such as OpenAI, Anthropic, and Cohere for seamless integration and rapid development [6, 12, 34].

The integration of LLMs into applications introduces new codelevel artifacts [16, 46], including prompts[15], hyperparameter settings [7] (e.g., temperature and max tokens), and LLMs frameworks like LangChain[21]. Among these, *prompts* play a pivotal role as the primary mechanism for guiding LLM behavior, allowing developers to tailor outputs to specific needs through a process known as prompt engineering [43]. In addition to these code-level artifacts, LLM-based applications encompass additional considerations, including operational costs, such as token usage[35], and advanced capabilities such as model fine-tuning offered by LLM APIs [5].

Although LLM APIs provide convenient access to powerful models, they pose significant long-term maintenance challenges. These challenges include the need for continuous adjustments to well-engineered prompts in response to model updates [27], as well as

the frequent modification of prompts—such as adding new instructions, rephrasing details, or restructuring templates—to align with evolving project requirements and the introduction of new features [46]. Furthermore, developers often resort to short-term fixes, such as prompt tweaks or token adjustments, to address the immediate challenges posed by LLMs rather than implementing scalable, long-term solutions [32]. Over time, these continual adjustments risk accumulating *Technical Debt (TD)*, increasing the maintenance burden required to keep LLM-based applications functional and aligned with user needs.

The technical debt (TD) concept describes the trade-offs developers make to accelerate software delivery at the expense of long-term maintainability [13]. Such trade-offs yield an *interest* that must be paid in the form of future rework [39, 50]. When these debts remain unaddressed, they become more expensive to resolve and can hinder the agility of the development team [50]. Within TD, *Self-Admitted Technical Debt (SATD)* refers to cases in which developers explicitly acknowledge design or implementation deficiencies through the comments, often annotated with TODO or FIXME [39]. SATD offers valuable insights into technical debt's human-driven decisions, revealing where and why debt accumulates. Identifying SATD comments allows developers to prioritize areas in the codebase requiring immediate attention [8] and assist developers in controlling long-term maintenance costs[28].

Numerous studies have explored SATD in traditional software systems, identifying SATD types, developing automated detection tools, and analyzing their lifecycle [8, 28, 50]. However, as software development continues to evolve, new application domains and emerging technologies introduce novel forms of SATD and associated challenges. In particular, Machine Learning (ML) and Deep Learning (DL) systems introduce unique complexities that go beyond those encountered in traditional software systems [44]. For example, O'Brien et al.[33] proposed a taxonomy of 23 ML-specific SATD types such as data dependency debt, model debt, and evaluation debt, while Bhatia et al.[9] pinpointed highly debt-prone stages in ML pipelines like data preprocessing and model training. Moreover, Pepe et al. [37] identified SATD categories specific to DL, highlighting infrastructure dependencies and inference-related SATD.

LLM-based applications introduce a fundamentally different development environment. Instead of managing large-scale training pipelines with dedicated hardware, developers leveraging LLM APIs primarily focus on *prompt engineering*, *hyperparameter tuning*, *LLM framework integration*, and *cost management* [11]. Yet we lack data on how these practices contribute to SATD in real-world LLM projects. Specifically, developers of LLM-based applications:

 Prioritize prompt management, which is a critical aspect that previous studies have largely overlooked but is integral

to shaping model behavior, defining user roles, aligning task contexts, and ensuring appropriate output formats. For instance, developers must decide whether to #break new prompts into configurations or use frameworks like LangChain to streamline prompt management (e.g., #TODO: use langchain?).

- (2) Utilize prompt learning[10] techniques (e.g., in-context learning, such as *n-shot* examples) as a more flexible alternative to data-intensive fine-tuning. A common challenge here includes SATD comments like #TODO: consider few-shot examples.
- (3) Iteratively configure **LLM hyperparameters** (e.g., temperature, top-p, and max tokens), which are crucial for optimizing LLM outputs [42] yet these configuration decisions are rarely captured in SATD studies [37]. For instance, developers document issues like #TODO handle top_p, top_k, etc. as unresolved configuration decisions.
- (4) Adopt **LLM-specific frameworks** to orchestrate interactions between applications and LLMs. These frameworks also introduce their own SATD when decisions on implementation or customization are left suboptimal or incomplete.
- (5) Manage operational costs, including token usage and model pricing, which are unique to the LLM domain and add a distinct dimension to technical debt.

By identifying these distinct sources of SATD in LLM-based applications, we aim to enhance existing ML/DL taxonomies [9, 24, 33, 37], systematically identify poor prompt engineering practices such as "prompt smells" [42] and "prompt requirement smells" [48], which can compromise model performance, and motivate the development of automated tools for effective prompt refactoring. Our findings contribute to establishing best practices for the long-term maintainability and reliability of LLM-based applications. To achieve these goals, we conduct an empirical study to address the following research questions:

RQ1: What is the extent and distribution of SATD in LLM projects? *Motivation:* As LLMs become integral to advanced applications, understanding the prevalence and patterns of SATD is crucial for identifying risks, improving maintainability, and developing targeted strategies to manage and mitigate technical debt in LLM-based projects effectively.

RQ2: Which parts of LLM-based applications are prone to SATD? *Motivation:* LLM systems introduce complexities distinct from traditional ML/DL pipelines. Understanding which components most frequently incur SATD can help refine and extend existing technical debt taxonomies.

RQ3: Which prompt techniques are more prone to SATD? *Motivation:* Building on insights from RQ2, this question further investigates prompt-level debts to identify which specific prompt techniques[38] might introduce unique technical debt challenges.

This paper advances the understanding of SATD in LLM-based applications by:

(1) Conducting the first empirical study to identify and classify five unique types of LLM-related SATD, analyzing their prevalence and impact.

- (2) Providing actionable guidelines for effective mitigation strategies to reduce LLM-related SATD.
- (3) Providing a comprehensive replication dataset to support reproducibility and facilitate further research.¹

The paper is structured as follows: Section 3 presents the approach used in this study, and Section 4 explains the results and findings of the study. Section 5 presents the implications and additional discussion, and Section 6 discusses the limitations of our study. Section 2 presents related work of self-admitted technical debt and their effect on traditional and ML software projects. Finally, in Section 7, we provide conclusions and discuss future work.

2 Related Work

In this section, we discuss existing work relevant to self-admitted technical debt (SATD) research conducted on traditional and ML software projects.

2.1 SATD in traditional software

SATD has been widely studied in traditional software development. Potdar et al. [39] found that 2.4% to 31% of codebases in open-source projects contain SATD, often introduced by experienced developers. Maldonado et al. [28] showed that SATD is typically used to mark areas needing future improvements or bug fixes, remaining in the code for an average of 18 to 172 days, with design debt being the most common type [29]. Further research by Bavota et al. [8] and Wehaibi et al. [50] explored the relationship between SATD and software quality, showing that SATD tends to accumulate over time and has no direct correlation with code complexity.

2.2 SATD in ML projects

SATD Prevalence and Categorization in ML Systems: One of the key empirical studies on SATD in ML systems, Obrien et al.[33] introduced a comprehensive taxonomy of 23 distinct types of SATD specifically tailored to ML systems. This work provides a deeper understanding of the types of debt that commonly arise in ML projects, including data preprocessing debt, algorithm-specific debt, and configuration debt. These types of debt are especially prevalent in the early stages of the ML pipeline, such as data preprocessing and model training, where developers must balance speed and accuracy.

Building on this foundation, Bhatia et al. [9] highlights the high prevalence of SATD in these projects, reporting that ML systems accumulate 2.1 times more SATD than non-ML software. This increased prevalence is largely attributed to the inherent complexities of data dependencies, configuration management, and the need for frequent experimentation in ML pipelines. As developers iterate quickly to refine models and manage large datasets, they often introduce SATD in areas like model performance, code maintainability, and data handling.

SATD in **Deep Learning Frameworks**: Another critical aspect of SATD in DL projects arises from the frameworks used to develop these models. Liu et al.[24] investigated the prevalence of technical debt in deep learning frameworks such as TensorFlow and PyTorch.

¹https://doi.org/10.5281/zenodo.15292881

The study revealed that design debt and data dependency debt were particularly common in these frameworks, driven by the need for compatibility with evolving libraries and hardware infrastructure. The accumulation of SATD in DL projects often necessitates continuous refactoring to maintain code quality. Tang et al.[47] explored the relationship between refactorings and SATD in ML/DL systems, demonstrating that refactoring efforts are frequently concentrated in areas where SATD is most prevalent, such as data preprocessing and model training.

Similar to Obrien et al.[33], Pepe et al.[37] introduced a comprehensive taxonomy of SATD specific to Deep Learning (DL) systems, categorizing 41 unique SATD types into infrastructurerelated (e.g, such as GPUs and TPUs) and DL life-cycle-related debt. The DL life-cycle category focuses on SATD arising from data preparation, model design (e.g., attention masks and pooling layers), training logic (e.g., loss functions), and inference (e.g., postprocessing). While previous efforts [37] highlight SATD across the DL pipeline – including prompts in inference-related SATD, their focus remains on DL system development rather than the unique challenges faced by developers building applications on top of LLM. In many DL systems, inference revolves around passing input data to a trained model, coupled with occasional adjustments to hyperparameters such as beam size or repetition penalties. By contrast, prompts in LLM-based applications serve as elaborate programming instructions[23, 41] that specify user roles, task contexts, and expected output formats. Refining these prompts often requires manual trial-and-error, multiple iterations, and can incur significant time and financial costs [16]. This expanded function demands ongoing, fine-grained maintenance-for example, configuring nshot examples or setting system/user prompts. Developers may defer these refinements (e.g., # TODO Figure out how to create system prompt), and thus introducing technical debt that requires dedicated management. This distinction positions our work to explore SATD in LLM-based applications that remain unexplored in the existing literature.

3 Research Method

Figure 1 illustrates the overall methodology used in our study. The approach involves multiple key steps: first, the collection of relevant Python files related to Large Language Models (LLMs) (the top layer in the figure); the second step involves extracting source code comments, conducting a thorough cleaning process, and detecting Self-Admitted Technical Debt (SATD) comments as shown in the second layer. In the third layer, we illustrate our manual classification of LLM-related SATD using our sample. Finally, we present the collection of developer-written prompts and their corresponding SATD instances.

3.1 LLM Files Collection

We utilized the PromptSet dataset ² [38]. This dataset is curated to capture developer-written prompts involving major LLM libraries, such as OpenAI, Anthropic, and Cohere. It also includes references to frameworks like LangChain, which, while not an LLM itself, developers typically interact with the LLMs via such a framework rather than calling the LLMs APIs directly[3]. By including

LLM	Count (%)
OpenAI	41,403 (44.45%)
LangChain	22,348 (24.00%)
LangChain, OpenAI	20,803 (22.33%)
Cohere	3,517 (3.78%)
Unknown	1,765 (1.89%)
Anthropic	505 (0.54%)
Cohere, OpenAI	531 (0.57%)
Anthropic, OpenAI	446 (0.48%)
Cohere, LangChain, OpenAI	443 (0.48%)
Anthropic, LangChain	415 (0.45%)
Cohere, LangChain	331 (0.36%)
Anthropic, LangChain, OpenAI	307 (0.33%)
Anthropic, Cohere, OpenAI	154 (0.17%)
Anthropic, Cohere, LangChain, OpenAI	140 (0.15%)
Anthropic, Cohere, LangChain	24 (0.03%)
Anthropic, Cohere	10 (0.01%)
Total	93,142 (100.00%)

Table 1: Distribution of Files by LLM/Framework

LangChain in our SATD study, we aim to capture a more comprehensive view of how technical debt arises from the LLMs and the frameworks that manage LLM's input and output.

The dataset, compiled in January 2024, was sourced by mining 37,944 GitHub repositories and focused on Python files that explicitly reference these LLM APIs or frameworks. It consists of 93,142 Python files, each containing source code and metadata, including developer-written prompts. To better understand the distribution of SATD within the dataset, we grouped the files according to the specific LLM libraries or frameworks they referenced (e.g., OpenAI, Anthropic, Cohere, and LangChain). This classification was based on unique keywords outlined by Pister et al.[38]. In Table 1, we present the LLM distribution. The complete list of keywords used for this grouping is provided in Appendix A.

Files that did not match the predefined API or framework keywords were classified under the *Unknown* category. A manual examination of 20 sample files from this category revealed that they typically served as utility files that complemented LLM-related code, rather than directly interacting with the specified LLMs or frameworks. Consequently, we excluded these utility files from our analysis, resulting in a refined dataset of **91,377** Python files.

3.2 SATD Comments

In this section, we detail the process of extracting Python comments from LLM-related files and outline the steps taken to clean these comments for effective SATD detection. Additionally, we explain how SATDDetector, complemented by SATD-specific keywords, was utilized to identify SATD comments within the dataset.

Comments Extraction: To begin quantifying SATD comments, we first extracted source code comments from the 91,377 Python files in our dataset. This extraction was performed using Python's tokenize module³, which breaks down Python source code into individual components (tokens), including single-line comments (#). During the extraction process, we discovered that 20,493 out

 $^{^2} https://hugging face.co/datasets/pisterlabs/promptset$

 $^{^3} https://docs.python.org/3/library/tokenize.html\\$

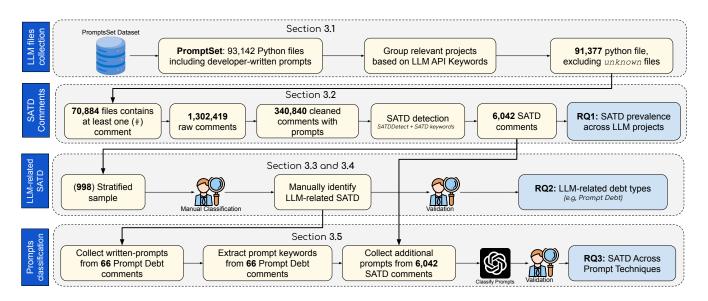


Figure 1: An overview approach.

of the 91,377 Python files contained no single-line comments. As a result, these files were excluded from further analysis, reducing our dataset to 70,884 Python files. The initial comments extraction yielded 1,302,419 raw comments across the remaining files. To ensure the comments were suitable for SATD detection, we applied a cleaning process similar to those described in prior research [14, 33]. This process involved removing non-informative or irrelevant comments, such as duplicate comments, license headers, generated code comments, and non-English comments. After cleaning, we obtained a refined dataset of 340,840 comments, which became the basis for SATD analysis.

Detecting SATD: For SATD detection, we adopted a methodology by O'Brien et al.[33]. We first employed the SATDDetector tool⁴, an automated tool designed to classify SATD comments based on a machine learning model trained on traditional software project data [19]. SATDDetector, however, was primarily trained on comments from traditional software projects and showed some false positives when applied to ML-related SATD, as noted by O'Brien et al. To mitigate this issue, we supplemented SATDDetector's results with an additional keyword-based detection approach, using SATD-related keywords identified in [14]. From the 340,840 cleaned comments, we detected a total of **6,042** SATD comments, representing approximately 1.77% of the entire comment dataset. Further analysis showed that 3,630 (5.12%) of the 70,884 Python files contained at least one SATD comment.

3.3 SATD Sampling

To address our RQ2, we utilized a statistically significant, stratified sample of 998 SATD comments (with a 95% confidence level and a 5% margin of error) from the 6,042 SATD comments identified in LLM-related files. Our sampling method followed the statistical approach described by [9, 33, 51], where we ensured a proportional

representation of comments across various LLM APIs. The sample distribution is detailed in Table 2.

LLM	Sample Size
OpenAI	382
OpenAI, LangChain	250
LangChain	167
Cohere	85
OpenAI, Cohere	25
OpenAI, Anthropic, LangChain	21
OpenAI, Anthropic	15
OpenAI, Cohere, LangChain	15
Anthropic	10
OpenAI, Anthropic, Cohere	9
Anthropic, LangChain	8
Cohere, LangChain	4
OpenAI, Anthropic, Cohere, LangChain	3
Anthropic, Cohere	1

Table 2: Sample Sizes by LLMs

3.4 Manual Categorization of LLM-Related SATD

We aim to extend existing ML/SATD taxonomies by identifying categories that are unique to LLM-based projects. To this end, we conducted a manual analysis of 998 comments from our dataset to construct a comprehensive classification of LLM-related SATD. Grounding our approach in (i) official documentation for widely used LLMs [1–4] and (ii) LLM application challenges and evolution [11, 18, 37, 46], we focused on practical aspects that frequently introduce debt in LLM-based systems—namely, prompting, finetuning, hyperparameter tuning, token cost, and framework usage (e.g., LangChain).

 $^{^4} https://github.com/Tbabm/SATDDetector-Core\\$

SATD Type Definition Examples Count (%) # TODO: refine the prompt template Prompt Debt Debt resulting from poorly structured, 66 (6.61%) inefficient, or suboptimal prompts. # TODO Figure out how to create system prompt # HACK since we can't specify exact prompt in .yaml # TODO Fix cases where this makes the prompt too long LLM-45 (4.51%) Debt related to improper or suboptimal # TODO handle top_p, top_k, etc. hyperparameters tuning of LLM hyperparameters. # TODO: Make max_tokens configurable Debt # TODO: figure out which temperature is best for evaluation LLM-Integrated Debt arising from the integration and # TODO: remove hardcoded context size once ... via langchain 43 (4.31%) Framework Debt orchestration of LLM frameworks # TODO: this check has been switched to "false" after LlamaIndex deprecated LLMPredictors # TODO: Figure out pipeline prompts to avoid this 21 (2.10%) Cost Debt Debt refers to explicit issues related to # TODO calculate cost based on the whole prompt token consumption and model pricing # TODO Figure out how to use 'gpt-3.5-turbo' instead, since it's selection. 1/10 the price of davinci # TODO Assume a cost of \$0.00006 per token. Debt related to insufficient fine-tuning Learning Debt' # TODO: push a finetuned gpt2 image to Huggingface and .. 6 (0.60%) or prompt learning. # TODO: it might be profitable to fine-tune Result statements to emphasize key skills # TODO: consider few-shot examples

Table 3: LLM-Related SATD per comment, Definition and Examples

For instance, official LLM documentation [1-4] outlines recommended practices for prompt design, hyperparameter settings, and LLM usage, yet our dataset contained numerous developer comments (e.g., #TODO: figure out max input size for cohere, #TODO: consider better prompt) demonstrating a deviation from these guidelines. In addition, research findings [11, 18, 37, 46] highlight common pitfalls such as repeated prompt revisions and token-limitation workarounds. Observing similar challenges in our dataset allowed us to more precisely label and categorize instances of SATD related to LLM usage. Interestingly, while these sources did not explicitly mention LLM framework usage (e.g., LlamaIndex[26]), our classification captured multiple SATD instances related to such frameworks. From these analyses, we identified new categories to reflect these LLM-specific concerns. In Table 3 (the table will be explained in detail in Section 4.2), we show the types we found based on our sample, their definition, and examples.

To address the risk of bias in our manual classification of SATD comments, we adopted a multi-rater approach inspired by prior SATD studies [33]. Two qualified investigators independently labeled each comment, and we computed Cohen's Kappa[31] to quantify inter-rater reliability. We obtained a Cohen's Kappa of 0.74, indicating a substantial level of agreement. Any disagreements were resolved through a joint discussion with a third investigator.

3.5 Prompt Classification

To determine which prompt techniques used by developers are most subjected to technical debt (RQ3), we categorized the prompts (e.g., Zero/Few shots, Chain-of-Thoughts (CoT)) according to Pister et al. [38] prompt techniques classifications.

The findings from RQ2, which focused on LLM-related debts, revealed that Prompt Debt was the most prevalent type of debt in our sample. Specifically, out of 998 instances, 66 were linked to

issues involving prompts. To build on this insight, we began by collecting developer-written prompts associated with these 66 cases of Prompt Debt. Then, we expanded our prompt sample size by developing search strings based on the prompt keywords identified from these 66 instances to gain a deeper understanding. The keywords used were prompt, conversation, message, and query. We applied these keywords across the entire dataset of 6,042 SATDs. This approach enabled us to identify an additional 383 SATD comments that are associated with the prompts crafted by developers, bringing the total to 449 SATD comments related to prompting, along with their developer-written prompts.

However, in our analysis of the prompt content, we observed two distinct methods of prompt presentation. Some files include the full prompt (e.g., *Prompt: ["You are a bot designed to summarize a"]), which aligns with our objective to classify the precise techniques developers employ. In contrast, other prompts adopt a placeholder-based approach, replacing the actual text with placeholders such as ['PLACEHOLDER': 'PLACEHOLDER']. Since our core study centers on analyzing SATD comments, we focused on only those prompts that explicitly displayed their full content, leading us to exclude 294 placeholder-based prompts and retain a final set of 155 prompts for classification.

To classify these 155 prompts, we employed LLM-based approach similar to previous studies [38, 46]. Our prompt contains the definitions of each prompt technique alongside illustrative examples in a few-shot setting, allowing us to effectively handle longer prompts (more than 40 words) that include multiple techniques. We manually validated a random sample of 40 instances ($\approx 26\%$ of the dataset) to assess the reliability of the prompt technique classifications. Among these, 92.5% (37/40) were correctly classified, while three cases (7.5%) were misclassified. The misclassifications primarily involved distinguishing between Instruction-based prompting

^{*}Refers to both data-intensive fine-tuning (continuing training on a task-specific dataset) and prompt learning[10], where temporary learning methods (e.g., In-context learning) used to guide the model's output.

and Chain-of-Thought (CoT) prompting, where all misclassified cases involved instructional prompts being incorrectly labeled as CoT due to the presence of multiple sub-instructions.

Each prompt in our dataset maintains a one-to-many relationship with its associated SATD comments. Nonetheless, it is important to note that while these SATD comments reference *prompting activities*—as identified by prompt-related keywords—they do not necessarily tie back to the specific prompt techniques developers used. We provide an example of this case in the RQ3 results.

4 STUDY RESULTS

The study results present the prevalence of SATD comments across all LLM files, as shown in Table 4. Out of 6,042 SATD comments, we selected 998 representative sample comments proportionally based on the distribution of LLMs in our dataset as described in Section 3-A. From these comments, we identified SATD comments related to LLMs (see Table 3). Each LLM-specific SATD type in the table is supported by at least one comment in our sample. We used a subset of 155 developer prompts to analyze prompt techniques, each accompanied by its associated SATD comment. We then classified these prompts based on the prompt technique. Each technique shown in Table 5 is represented by at least one written-prompt in our sample.

4.1 RQ1: What is the prevalence of SATD in LLMs files?

The prevalence of SATD across LLM APIs offers valuable insight into the scale and significance of these LLMs' challenges, helping to determine whether they deserve greater attention. Table 4 illustrates the distribution of SATD among various LLM APIs, with OpenAI alone accounting for a substantial 54.49% of all identified SATD. Following OpenAI, the combination of OpenAI with LangChain constitutes 20.39% of the SATD.

A closer examination of our sample—250 SATD comments specifically related to both OpenAI and LangChain—provides insights into each API's contribution to the overall technical debt. Within this subset, we noticed that LangChain contributes slightly more to SATD than OpenAI. This is also evident, where LangChain alone accounts for 12.35% of the SATD, which adds a layer of complexity that developers need to address.

On the other hand, Cohere and Anthropic, whether used independently or in combination with different APIs, exhibit lower incidences of SATD, with Cohere contributing 5.81% and Anthropic only 0.61%. This difference may stem from their less frequent usage compared to OpenAI and LangChain, as evidenced by the substantially lower total comment volumes of 23,605 for Cohere and 1,271 for Anthropic, which results in fewer instances of SATD.

Overall, the high occurrence of SATD in OpenAI-based systems and LangChain highlights the need for proactive technical debt management, particularly given the widespread adoption of these tools

Summary of RQ1: OpenAI exhibits the highest occurrence of SATD compared to other LLMs. Additionally, the frequent integration of the LangChain framework contributed to a higher SATD within LLM-related files.

LLM	SATD Occur.	Total # of Comments
OpenAI	3292 (54.49%)	197,380
OpenAI, LangChain	1232 (20.39%)	72,044
LangChain	746 (12.35%)	30,751
Cohere	351 (5.81%)	23,605
OpenAI, Cohere	98 (1.62%)	4,766
OpenAI, Anthropic, LangChain	81 (1.34%)	1,819
OpenAI, Anthropic	58 (0.96%)	2,400
OpenAI, Cohere, LangChain	58 (0.96%)	2,909
Anthropic	37 (0.61%)	1,271
OpenAI, Anthropic, Cohere	34 (0.56%)	761
Anthropic, LangChain	30 (0.50%)	1,300
Cohere, LangChain	12 (0.20%)	995
OpenAI, Anthropic, Cohere, LangChain	11 (0.18%)	756
Anthropic, Cohere	2 (0.03%)	42
Total	6042 (100%)	340,840

Table 4: SATD Occurrence by LLM.

4.2 RQ2: Which parts of the LLM are prone to SATD?

In RQ2, we categorize and analyze LLM-related technical debts to identify which parts of LLM, such as prompting and fine-tuning, are most prone to SATD. This section includes examples illustrating these debts and providing insights into common challenges developers face in LLM-based projects.

Table 3 presents the distribution of LLM-related SATD per comment in our sample. *Prompt Debt* is the most common, with 66 instances (6.55%) out of 998 comments, indicating frequent challenges in designing and optimizing prompts. *LLM- hyperparameters Debt* follows with 45 instances (4.46%), highlighting issues related to crucial LLM tuning parameters like temperature. *LLM-Integrated Framework Debt*, with 43 instances (4.27%), points to difficulties in integrating and managing frameworks such as LangChain for LLM workflows. Lastly, *Cost Debt* (21 instances, 2.10%) and *Learning Debt* (6 instances, 0.60%) appear less frequently, showing that concerns about tokens costs and incomplete fine-tuning are relatively rare but still notable.

Prompt Debt: In our analysis of *Prompt Debt* comments, the most frequent issue involves incomplete *prompt configurations*—that is, the need for prompt structures or templates to be dynamically adjustable in order to accommodate changing requirements and contexts (as indicated by examples in Table 3). Developers frequently struggle to create flexible, reusable prompts that can be universally applied without extensive hardcoding. For instance, the comment #TODO: other conversation template highlights the absence of a centralized, standardized approach to conversation-based task, while #TODO: dynamically set the city here in the prompt shows that certain context-specific (i.e, city) details remain manually embedded.

A second significant challenge developers encounter revolves around *optimizing prompt design*. Our dataset reveals that LLM developers often struggle to craft prompts that achieve desired outcomes, follow the correct format and manage prompt length, which can directly influence the LLM's performance for a given task [30].

For example, a comment such as #TODO write custom prompt and parse it to get best results underscores the need for more precise, task-specific prompts. Similarly, comments referencing overly long prompts (e.g.,.. prompt too long) point to the difficulty of maintaining clarity and focus within the token limits of the model, potentially degrading comprehension and output accuracy [25]. Another design challenge is getting the output in the right format. Developers sometimes desire the model to respond in JSON or another structured layout, but the prompt does not clearly state it. This leads to comments like #TODO: Turn response to JSON, which shows that formatting issues are often postponed.

LLM-hyperparameter Tuning Debt: In LLM-based systems, tuning models and adjusting their hyperparameters is critical to achieving optimal results [7, 36]. However, improperly configured parameters, such as temperature, and top_p, can lead to suboptimal results and technical debt. Table 3 shows several examples of developers facing difficulties tuning parameters to fit their specific use cases. For instance, the comment #TODO: figure out which temperature is best for evaluation demonstrates a common challenge where developers struggle to determine the appropriate temperature setting to control the randomness of the model's output. Similarly, ### TODO: handle top_p, top_k, etc. reflects the complexity in adjusting probability and sampling techniques, which can significantly impact the diversity and creativity of generated LLM responses [7].

LLM-Integrated Framework Debt: frameworks like LangChain [3] and LlamaIndex[26] provide significant advantages for developers working with LLM-based applications. In our analysis, LangChain was the most frequent framework regarding associated technical debt, having the majority of SATD instances compared to other frameworks like LlamaIndex.

LangChain helps developers in two main perspectives: *prompt management* and *LLM output handling*. In prompt management, LangChain allows developers to create, format, and optimize prompts before sending them to an LLM. In LLM output handling, LangChain processes the responses from the LLM such as parsing and formatting the LLM output.

The primary challenges in using LangChain often come from prompt management as shown in Table 3. For instance, ## TODO: Replace with f strings and all PromptTemplate indicates a situation to refactor prompt construction using Python's f-strings and LangChain's PromptTemplate ⁵ for more precise, more maintainable prompt. For LLM output formatting, technical debt is evident in comments like ## TODO: use langchaing output parser. This shows a recognized but postponed effort to incorporate LangChain's output parser, which can lead to inefficient handling of responses, causing potential inconsistencies in downstream applications that rely on formatted and predictable outputs.

Cost Debt: While LLMs are highly accessible and powerful, they come with significant cost considerations. Costs are broken down into input and output costs—both scaling with token length—making extensive use prohibitively expensive for LLM-based applications [49]. We found that Cost Debt is strongly related to finding options

for i) lower-cost models or ii) minimizing token usage. The examples in Table 3 indicate that the developers are strongly aware of LLM usage costs. For instance, ## TODO Figure out how to use 'gpt-3.5-turbo' instead, since it's 1/10 the price of davinci illustrates the consideration of cost-effective model alternatives. Comments like # TODO: Truncate the output to meet the token requirement and save\$\$ highlight developers' efforts to minimize token usage. This indicates that cost management is a significant aspect of SATD in LLM projects, as developers actively look for ways to reduce expenses by minimizing token usage or finding lower-cost models.

Learning Debt: Developers use prompts to guide LLMs in generating desired outputs, often employing learning techniques such as in-context learning, including few-shot prompting[10]. Another effective approach is fine-tuning LLMs on task-specific data. Currently, certain models by OpenAI can be fine-tuned to better align with specific datasets [5]. Fine-tuning typically allows developers to achieve more consistent and high-quality results compared to using prompts alone. We consider two learning approaches. The first approach is fine-tuning, which involves training a pre-trained model on a new dataset tailored to a desired task [40]. The second approach is prompt-learning [10]. In this method, the model is provided with n-examples at inference time, guiding it to perform the task based on the context provided by these examples.

Our analysis of Learning Debt indicates that insufficient or incomplete fine-tuning is a prevalent source of technical debt in LLM-based projects. For example, the comment # TODO: it might be profitable to fine-tune ... instead of letting the base model handle it suggests that developers acknowledge the potential performance improvements achievable through fine-tuning rather than relying solely on the base model. This observation also reflects a decreasing dependence on the prompt learning approach, as incorporating examples or context into a general-purpose model may yield the desired outcomes more effectively.

While prompt-learning appeared slightly less frequently in our dataset compared to fine-tuning, we observed often impact on Prompt Debt. For example, the comment ## TODO: consider few-shot examples. This case demonstrates a missing learning approach and restructuring the prompt design to include the example.

Summary of RQ2: Prompt design and configuration, hyperparameter tuning, and the integration of LLM frameworks are the primary sources of LLM-specific SATD.

4.3 RQ3: Which prompt techniques are more prone to SATD?

For RQ3, we collected 155 developer-written prompts and categorized them based on the prompting techniques employed to identify which techniques are most prone to accumulating SATD. We analyzed the associated SATD comments for each technique to determine if they were directly related to the specific prompting approach used. Table 5 presents the results. In total, we analyzed 155 prompts, excluding 24 prompts (13.71%) from classification due to a lack of meaningful content. For example, prompts like ['Hi, this

⁵LangChain PromptTemplate Documentation: https://python.langchain.com/api_reference/core/prompts/langchain_core.prompts.prompt.PromptTemplate.html

is a test'] were categorized as unknown, as they did not provide sufficient context to be identified as any specific technique.

In examining the SATD comments associated with these prompt techniques, we found that a significant number of comments were specific to the techniques used. However, it is essential to note that not all SATD comments were directly related to the technique itself. For instance, SATD comments associated with Zero-shot prompts were more general and did not directly relate to the unique aspects of Zero-shot prompting. Our analysis highlights SATD comments directly referencing challenges inherent to the techniques.

Instruction Block prompts, where developers provide explicit commands to LLMs, were identified as particularly vulnerable to SATD, comprising 37.7% of our dataset. This technique often results in issues related to Prompt Debt, specifically suboptimal instruction clarity and excessive instruction length. Such issues can lead to comments like **p** #If prompt is too long ... and .. Could be missing instructions, where developers acknowledge the need to adjust the prompt length and enhance the clarity of provided instructions. Instruction length is particularly problematic, as the model's ability to retrieve and interpret instructions from the middle of the context diminishes as prompts become longer[25]. This can result in incomplete or less accurate outputs when essential details are buried within lengthy prompts. Unclear or suboptimal instructions, coupled with overly long prompt instructions, can severely impact the effectiveness of LLM responses by leading to misinterpretation or incomplete task execution.

Few-shot Few-shot prompts accounted for 17.7% of our dataset, underscoring their common use in providing contextual examples to

guide LLM behavior. The SATD comments linked to this technique were mainly associated with Learning Debt, as discussed in RQ2. These comments revealed key challenges, especially around improving the quality and relevance of the examples used. Comment like make this few-shot on real examples .. suggests that developers often rely on placeholder examples that do not reflect real tasks. Similarly, so .. positive/negative examples to determine final classification. Without these carefully selected examples, classification task and model reliability may decline [9, 10].

The **Documentation** technique (12.28%) involves incorporating

documentation directly within prompts to provide context or background information for LLMs. SATD in this area often arises from the challenges of managing and processing large document inputs. Developers frequently encounter issues with the efficient handling of these substantial text blocks, as illustrated by comments like . . adds all documents in the prompt. Test more smart approaches and Optional chunking for documents that are too large. These comments reflect the struggle to ensure that large documents are integrated in a way that maintains performance without overwhelming the model's capacity. To optimize prompt performance and reduce memory load, developers often identify the need for advanced document handling methods. For instance, comments like . . . combine with LangChain.DocumentLoader? to integrating tools specifically designed for document management, which could improve the workflow.

Table 5: Distribution of Prompt Techniques, Associated SATD Examples, and Prompt Examples

Technique	Prompt Example	Associated SATD	Count (%)
Instruction	You create summaries that keep all the information from the original text. You must keep all numbers and statistics from the original text. You will provide the summary in succint bullet points. For longer inputs, summarise the text into more bullet points	# If the prompt is too long, warn and give up. # TODO: Make it follow the prompt? Could be missing instructions # I need the instructions very specific, but maybe I can break it up into multiple system prompts, and # , you should revise the Instructions so that AI Assistant would quickly and correctly respond in the future.	66 (38.60%)
Few-Shot	Here are examples of different locations seperated by newlines	# TODO: make this few-shot on real examples instead of dummy ones # TODO: run few-shot and positive/negative examples to determine final classification	31 (18.13%)
Doc	Summarize this document to answer this question: Document: <i>PLACE-HOLDER</i> If the document isn't relevant, answer: Not Relevant.	# todo: debug. "stuff" just adds all documents in the prompt. Test more smart approaches. # TODO combine with LangChain.DocumentLoader? # TODO: Optional chunking for documents that are too large	21 (12.28%)
Chain-of-Thought	Let's work this out in a step by step way to be sure we have the right report use the goal	# let's get a chain of thought (COT) approach to understanding the data set. # TODO: after refactor: sample randomly instead, otherwise might e.g. only evaluate on CoT realized examples	12 (7.02%)
CodeBlock	Review this diff code change and suggest possible improvements and issues, provide fix example [PLACE-HOLDER]	# TODO: Consider adding project description for context in prompt # TODO: make this less hardcoded with if-else statements	11 (6.43%)
Zero-Shot	What is a good title for this chat that is 20 characters or less?	# FIXME: Doesn't seem to have same max_tokens == -1 for prompts==1	10 (5.85%)

The SATD examples for **Chain-of-Thought (CoT)** and **Code-Block** prompt techniques were fewer than those for the aforementioned techniques, yet they reveal the challenges developers face when integrating these approaches. In CoT, issues often arise from a lack of refining and refactoring the step-by-step structure to improve response quality. In CodeBlock, challenges include providing insufficient code-related context within prompts and addressing hard-coded structures.

Summary of RQ3: Instruction Block prompts are the most prone to SATD, often due to prompt length and instruction clarity issues. Few-Shot prompts accumulate debt when placeholder examples are not replaced with relevant, task-specific ones. Documentation prompts face challenges with managing large text inputs.

5 Additional Discussion and Implications

This section discusses the implications of our study's results in relation to each research question and provides practical guidance to mitigate technical debts arising from LLMs.

Additional Discussions: In RQ1, we showed that LLM-based applications exhibit challenges that make developers prone to introducing technical debts. We found that OpenAI-based applications account for most of SATD in the LLM projects we investigated. This is likely attributable to both the widespread use of OpenAI's API by developers [11] and the extensive number of associated comments (i.e., 197,380 out of 340,840 total comments). The combination of OpenAI with LangChain ranks second. This highlights that the interplay between LLM model functionalities and their orchestration is a critical factor in the accumulation of technical debt. This is also supported by our results in RQ2, LLM-Framework debts usually related to prompting design.

In RQ2, the most prevalent LLM-related debt was identified as Prompt Debt. Prompt engineering is an ongoing field that is continuously evolving both in academia and industry [15]. This rapid development creates challenges for developers, who are compelled to revisit and enhance their prompts as the design directly influences the desired output. Our observations suggest that some of the Prompt Debt issues, such as length, not only influence the quality of the LLM output but also contribute to Cost Debt. Longer prompts contain more tokens, which, in turn, increase the cost of using LLM services. This is also similar to Hyperparameter Tuning Debt. We frequently encounter hyperparameters like max_tokens, which set limits on the output length of LLMs. Although max_tokens does not necessarily affect the diversity and creativity of the responses of the LLM, we believe that it could also influence Cost Debt, as controlling the token limit affects the token pricing (i.e., a higher number of tokens increases the cost).

In RQ3, we found that Instruction Block, Few-shot, and Documentation prompts were the most prone to accumulating SATD, showing that certain prompting techniques naturally introduce specific risks. Instruction Block prompts often lead to issues with instruction clarity and length, Few-shot prompts with poor example selection, and Documentation prompts with large unstructured inputs that overwhelm context windows. However, some of this

debt can be reduced by adopting alternative strategies; for instance, using Retrieval-Augmented Generation (RAG) [22] can help manage Documentation prompts more effectively by retrieving only relevant information instead of embedding entire documents.

Implications:

Takeaway 1: Early Detection for Prompt Smells and Requirement Smells: Our findings on Prompt Debt closely relate to the ideas of Prompt Smells[42] and Prompt Requirement Smells[48]. Prompt Smells describe semantic and syntactic issues in prompts that reduce output desirability, explainability, or traceability, which can lead to lower output quality or harder-to-explain results. Prompt Requirement Smells, on the other hand, refer to issues like ambiguity or missing information when requirements are used as prompts, which can confuse the model and lower performance.

In our study, the issues we found under Prompt Debt—such as missing prompt templates, hardcoded values, long and unclear instructions, and poorly managed document inputs—align closely with Prompt Smells. These problems show that many prompts are fragile, difficult to maintain, and challenging for models to interpret reliably, directly affecting the quality of the outputs.

At the same time, we also found cases that align more with Prompt Requirement Smells. For example, some comments showed developers struggling with selecting relevant few-shot examples or forgetting to include important project context in prompts (e.g., #TODO: Consider adding project description for context in prompt). These issues mirror how vague or incomplete requirements can lead to requirement smells.

By identifying and addressing Prompt Debt early, developers can improve the clarity, structure, and completeness of prompts, which ultimately supports achieving the key aspects of generative AI output desirability.

Takeaway 2: Towards Deterministic Hyperparameter Tuning: The primary challenge lies in determining optimal threshold values for the hyperparameters of LLMs. Hyperparameters such as temperature, top_p, and top_k are critical because they significantly influence the diversity and creativity of the responses generated by the model [7]. These settings adjust how the model predicts and varies its output. Developers should experiment with parameter values to identify the most suitable or deterministic thresholds for their specific tasks. Once these optimal thresholds are established, they can be consistently applied to similar tasks to maintain reliable performance.

Takeaway 3: Managing LLM Framework Efficiently: Frameworks like LangChain simplify prompt management and LLM output handling, but improper use can introduce technical debt. Our findings indicate that issues such as unstructured prompt construction and delayed adoption of output parsers contribute to inefficiencies. Developers should consistently use prompt templates to maintain structured and reusable prompts while integrating LangChain's output parsers early to enforce predictable response formatting output.

Takeaway 4: Cost-Aware LLM Selection: Costs vary significantly among LLM providers. For instance, Wang et al.[49] show that using OpenAI's GPT-4 can cost \$30 for 1 million tokens, whereas

alternatives like Mistral 7B may cost as little as \$0.20. This difference emphasizes the importance for developers to carefully select models that align with their budget and performance requirements. Our findings indicate that Cost Debt is a significant concern, with developers actively seeking ways to minimize costs. This highlights the importance of dynamically selecting from a range of LLMs, utilizing lower-cost for general tasks, and switching to more powerful, higher-cost models only when necessary.

Takeaway 5: Improving Prompting Techniques: Instruction Block and few-shot prompting techniques were the most prone to SATD. Excessive instruction length and poorly selected examples can reduce prompt clarity and output quality. Developers should focus on concise, well-structured instructions and use example selection methods (e.g., embedding-based similarity [45]) to select more relevant few-shot examples, thereby improving model performance.

Threats To Validity

Internal Validity: One potential threat to our internal validity lies in the accuracy of the tool (SATDDetector [33]) used to detect SATD in LLM-focused repositories. Although SATDDetector has been effective in prior studies, it was originally designed for more general software settings. To mitigate this risk, we supplemented the tool's results with a keyword-based detection approach[14]. Another concern involves human bias in the manual classification of LLM-specific debts (RQ2). By following a process well established in prior work[33], we used a coding process for classification, achieving a Cohen's Kappa of 0.74 (indicating substantial agreement). For our GPT-based prompt classification, we additionally validated a random sample of 40 instances (26% of the dataset), finding a 92.5% correct classification rate. This approach reduces subjective errors and increases confidence in our classification outcomes. We chose not to use an LLM-based classifier to identify LLM-related debt because this domain is still emerging and frequently requires human judgment. For instance, automated methods might fail to recognize certain framework-specific debts (Section 3.4) if the classification categories are not yet well established. In contrast, manual classification allows for more precise detection of these emerging debt

External Validity: Our findings may be limited in scope because they focus on LLMs based primarily on PromptSet projects[38], which capture a specific subset of LLM-based projects and LLM frameworks. Consequently, the types of SATD identified might not apply equally to highly customized transformer models, where deeper configurations—such as learning rates, optimizer choices, batch sizes, and fine-tuning epochs-are more prominent. In contrast, general-purpose LLMs emphasize prompt engineering and API-level customization. Similarly, our findings on LLM-Integrated Framework Debt primarily focus on LangChain due to its widespread use. Still, similar challenges may arise in other LLM orchestration frameworks like LlamaIndex, which also introduce complexities in prompt management, response parsing, and dependency maintenance. Future studies should expand to a wider range of

models and LLM frameworks to ensure that the forms of SATD uncovered generalize across diverse LLM-based applications.

Construct validity: This concerns the extent to which our study accurately captures LLM-related SATD. We refined our classification by integrating insights from official LLM documentation [1-4] and empirical research on LLM-related development challenges [11, 18, 46]. Additionally, we considered inference-related debt [37], which briefly addresses prompt formulation and hyperparameter tuning within a deep learning (DL) applications. Moreover, to improve construct clarity, we distinguished between closely related debt types. For example, while few-shot examples are embedded in prompts and often overlaps with prompt formulation, we categorized comments as Learning Debt when they involved reasoning about example selection or task-specific adaptation. This distinction allowed us to separate issues related to high-level learning approach from those related to Prompt Debt.

7 **Conclusion And Future Work**

This paper presents the first empirical investigation into self-admitted technical debt (SATD) specific to large language models (LLMs). Analyzing 93,142 Python files, the study finds that OpenAI accounts for 54.49% and the LangChain framework 12.35% of SATD instances. Prompt configuration is identified as the primary source of LLMspecific SATD, with instruction-based (37.7%) and few-shot prompts (17.7%) being the most prone to prompting techniques due to the lack of instruction clarity and example quality. The study provides a reproducible dataset and offers insights on LLM-related technical debt for more maintainable applications. In the future, we plan to expand our exploration of technical debt associated with LLMs by encompassing a wider range of debt types. Additionally, we aim to investigate the specific technical debts linked to various prompt techniques.

References

- [1] 2024. Anthropic API Documentation. https://www.anthropic.com/api.
- 2024. Cohere Documentation, Cohere Platform. https://docs.cohere.com/.
- [2] [3] 2024 LangChain Documentation: Providers OpenAI. https://python.langchain.com/docs/integrations/providers/openai/.
- 2024. OpenAI Platform Documentation. https://platform.openai.com/docs/concepts.
- 2024. OpenAI Platform Guide https://platform.openai.com/docs/guides/fine-tuning.
- Anthropic. 2024. Getting Started. https://docs.anthropic.com/en/api/gettingstarted. Accessed: 2024-10-05.
- Chetan Arora, Ahnaf Ibn Sayeed, Sherlock Licorish, Fanyu Wang, and Christoph Treude. 2024. Optimizing Large Language Model Hyperparameters for Code Generation. arXiv preprint arXiv:2408.10577 (2024).
- Gabriele Bavota and Barbara Russo. 2016. A large-scale empirical study on selfadmitted technical debt. In Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on. IEEE, 315-326.
- Aaditya Bhatia, Foutse Khomh, Bram Adams, and Ahmed E Hassan. 2023. An Empirical Study of Self-Admitted Technical Debt in Machine Learning Software. arXiv preprint arXiv:2311.12019 (2023)
- [10] Tom B Brown. 2020. Language models are few-shot learners. arXiv preprint arXiv:2005.14165 (2020).
- [11] Xiang Chen, Chaoyang Gao, Chunyang Chen, Guangbei Zhang, and Yong Liu. 2024. An Empirical Study on Challenges for OpenAI Developers. arXiv preprint arXiv:2408.05002 (2024).
- Cohere. 2024. API Reference. https://docs.cohere.com/reference/about. Accessed:
- Ward Cunningham. 1992. The WyCash Portfolio Management System. SIGPLAN OOPS Mess. 4, 2 (Dec. 1992), 29-30. doi:10.1145/157710.157715
- Everton da Silva Maldonado, Emad Shihab, and Nikolaos Tsantalis. 2017. Using natural language processing to automatically detect self-admitted technical debt.

- IEEE Transactions on Software Engineering 43, 11 (2017), 1044–1062.
- [15] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. In 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE). IEEE, 31–53.
- [16] Ahmed E Hassan, Dayi Lin, Gopi Krishnan Rajbahadur, Keheliya Gallaba, Filipe Roseiro Cogo, Boyuan Chen, Haoxiang Zhang, Kishanthan Thangarajah, Gustavo Oliva, Jiahuei Lin, et al. 2024. Rethinking software engineering in the era of foundation models: A curated catalogue of challenges in the development of trustworthy finware. In Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering. 294–305.
- [17] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. ACM Transactions on Software Engineering and Methodology 33, 8 (2024), 1–79.
- [18] Xinyi Hou, Yanjie Zhao, and Haoyu Wang. 2024. Voices from the Frontier: A Comprehensive Analysis of the OpenAI Developer Forum. arXiv preprint arXiv:2408.01687 (2024).
- [19] Qiao Huang, Emad Shihab, Xin Xia, David Lo, and Shanping Li. 2018. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering* 23, 1 (2018), 418–451.
- [20] Jean Kaddour, Joshua Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. 2023. Challenges and applications of large language models. arXiv preprint arXiv:2307.10169 (2023).
- [21] LangChain. 2024. API Reference. https://python.langchain.com/v0.2/api_reference/
- [22] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. Advances in neural information processing systems 33 (2020), 9459–9474.
- [23] Jenny T Liang, Melissa Lin, Nikitha Rao, and Brad A Myers. 2024. Prompts are programs too! understanding how developers build software containing prompts. arXiv preprint arXiv:2409.12447 (2024).
- [24] Jiakun Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. 2020. Is using deep learning frameworks free? characterizing technical debt in deep learning frameworks. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Society. 1–10.
- [25] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. Transactions of the Association for Computational Linguistics 12 (2024), 157–173.
- [26] LlamaIndex. 2024. LlamaIndex: Unlocking the Power of LLMs with Data-Driven Applications. https://www.llamaindex.ai/. Accessed: 2024-10-05.
- [27] Wanqin Ma, Chenyang Yang, and Christian Kästner. 2024. (Why) Is My Prompt Getting Worse? Rethinking Regression Testing for Evolving LLM APIs. In Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering-Software Engineering for AI. 166–171.
- [28] Everton da S Maldonado, Rabe Abdalkareem, Emad Shihab, and Alexander Serebrenik. 2017. An empirical study on the removal of self-admitted technical debt. In Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on. IEEE, 238–248.
- [29] Everton da S Maldonado and Emad Shihab. 2015. Detecting and quantifying different types of self-admitted technical debt. In 7th International Workshop on Managing Technical Debt (MTD). IEEE, 9–15.
- [30] Ggaliwango Marvin, Nakayiza Hellen, Daudi Jjingo, and Joyce Nakatumba-Nabende. 2023. Prompt engineering in large language models. In *International* conference on data intelligence and cognitive informatics. Springer, 387–402.
- [31] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. Biochemia medica 22, 3 (2012), 276–282.
- [32] Nadia Nahar, Christian Kästner, Jenna Butler, Chris Parnin, Thomas Zimmermann, and Christian Bird. 2024. Beyond the Comfort Zone: Emerging Solutions to Overcome Challenges in Integrating LLMs into Software Products. arXiv preprint arXiv:2410.12071 (2024).
- [33] David OBrien, Sumon Biswas, Sayem Imtiaz, Rabe Abdalkareem, Emad Shihab, and Hridesh Rajan. 2022. 23 shades of self-admitted technical debt: An empirical study on machine learning software. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 734–746.
- [34] OpenAI. 2024. Overview. https://platform.openai.com/docs/overview. Accessed: 2024-10-05.
- [35] OpenAI. 2024. Overview. https://openai.com/api/pricing/. Accessed: 2024-10-05.
- [36] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. 2024. An Empirical Study of the Non-determinism of ChatGPT in Code Generation. ACM Transactions on Software Engineering and Methodology (2024).
- [37] Federica Pepe, Fiorella Zampetti, Antonio Mastropaolo, Gabriele Bavota, and Massimiliano Di Penta. 2024. A Taxonomy of Self-Admitted Technical Debt in Deep Learning Systems. arXiv preprint arXiv:2409.11826 (2024).

- [38] Kaiser Pister, Dhruba Jyoti Paul, Ishan Joshi, and Patrick Brophy. 2024. Prompt-Set: A Programmer's Prompting Dataset. In Proceedings of the 1st International Workshop on Large Language Models for Code. 62–69.
- [39] Aniket Potdar and Emad Shihab. 2014. An exploratory study on self-admitted technical debt. In Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on. IEEE, 91–100.
- [40] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.
- [41] Laria Reynolds and Kyle McDonell. 2021. Prompt programming for large language models: Beyond the few-shot paradigm. In Extended abstracts of the 2021 CHI conference on human factors in computing systems. 1–7.
- [42] Krishna Ronanki, Beatriz Cabrero-Daniel, and Christian Berger. 2024. Prompt Smells: An Omen for Undesirable Generative AI Outputs. In Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering-Software Engineering for AI. 286–287.
- [43] 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).
- [44] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. Advances in neural information processing systems 28 (2015).
- [45] Benjamin Steenhoek, Md Mahbubur Rahman, Monoshi Kumar Roy, Mirza Sanjida Alam, Earl T Barr, and Wei Le. 2024. A Comprehensive Study of the Capabilities of Large Language Models for Vulnerability Detection. arXiv preprint arXiv:2403.17218 (2024).
- [46] Mahan Tafreshipour, Aaron Imani, Eric Huang, Eduardo Almeida, Thomas Zimmermann, and Iftekhar Ahmed. 2024. Prompting in the Wild: An Empirical Study of Prompt Evolution in Software Repositories. arXiv preprint arXiv:2412.17298 (2024).
- [47] Yiming Tang, Raffi Khatchadourian, Mehdi Bagherzadeh, Rhia Singh, Ajani Stewart, and Anita Raja. 2021. An empirical study of refactorings and technical debt in machine learning systems. In 2021 IEEE/ACM 43rd international conference on software engineering (ICSE). IEEE, 238–250.
- [48] Andreas Vogelsang, Alexander Korn, Giovanna Broccia, Alessio Ferrari, Jannik Fischbach, and Chetan Arora. 2025. On the Impact of Requirements Smells in Prompts: The Case of Automated Traceability. arXiv preprint arXiv:2501.04810 (2025).
- [49] Can Wang, Bolin Zhang, Dianbo Sui, Zhiying Tu, Xiaoyu Liu, and Jiabao Kang. 2024. A survey on effective invocation methods of massive llm services. arXiv preprint arXiv:2402.03408 (2024).
- [50] Sultan Wehaibi, Emad Shihab, and Latifa Guerrouj. 2016. Examining the impact of self-admitted technical debt on software quality. In Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on, Vol. 1. IEEE, 179–188.
- [51] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. 2019. An empirical study of common challenges in developing deep learning applications. In 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE). IEEE. 104–115.

Appendix A

```
# Keywords for Identifying LLMs
OpenAI: [
    "openai",
"OpenAI",
    "openai.Completion.create",
    "openai.ChatCompletion.create",
    "openai.Completion",
    "openai.ChatCompletion"],
Anthropic: [
    "anthropic",
"Claude",
    "anthropic.Completion.create",
    "claude.Completion"],
Cohere: [
    "cohere",
"cohere.Client",
    "cohere.generate",
    "cohere.chat",
    "cohere.summarize"],
LangChain: [
    "langchain",
    "LLMChain",
"PromptTemplate",
    "HumanMessage",
    "AIMessage",
"BaseTool",
    "@tool",
    "langchain.llms"]
```