Are Prompt Engineering and TODO Comments Friends or Foes? An Evaluation on GitHub Copilot

David OBrien
Dept. of Computer Science
Iowa State University
Ames, IA, USA
davidob@iastate.edu

Sumon Biswas
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
sumonb@cs.cmu.edu

Sayem Mohammad Imtiaz
Dept. of Computer Science
Iowa State University
Ames, IA, USA
sayem@iastate.edu

Rabe Abdalkareem
Dept. of Computer Science
Omar Al-Mukhtar University
Elbyda, JK, Libya
rabe.abdalkareem@omu.edu.ly

Emad Shihab
Concordia University
Montreal, QC, Canada
emad.shihab@concordia.ca

Hridesh Rajan
Dept. of Computer Science
Iowa State University
Ames, IA, USA
hridesh@iastate.edu

ABSTRACT

Code intelligence tools such as GitHub Copilot have begun to bridge the gap between natural language and programming language. A frequent software development task is the management of technical debts, which are suboptimal solutions or unaddressed issues which hinder future software development. Developers have been found to "self-admit" technical debts (SATD) in software artifacts such as source code comments. Thus, is it possible that the information present in these comments can enhance code generative prompts to repay the described SATD? Or, does the inclusion of such comments instead cause code generative tools to reproduce the harmful symptoms of described technical debt? Does the modification of SATD impact this reaction? Despite the heavy maintenance costs caused by technical debt and the recent improvements of code intelligence tools, no prior works have sought to incorporate SATD towards prompt engineering. Inspired by this, this paper contributes and analyzes a dataset consisting of 36,381 TODO comments in the latest available revisions of their respective 102,424 repositories, from which we sample and manually generate 1,140 code bodies using GitHub Copilot. Our experiments show that GitHub Copilot can generate code with the symptoms of SATD, both prompted and unprompted. Moreover, we demonstrate the tool's ability to automatically repay SATD under different circumstances and qualitatively investigate the characteristics of successful and unsuccessful comments. Finally, we discuss gaps in which GitHub Copilot's successors and future researchers can improve upon code intelligence tasks to facilitate AI-assisted software maintenance.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; • **Software and its engineering** → **Software creation and management**.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639176

KEYWORDS

technical debt, LLM, GitHub Copilot, code generation

ACM Reference Format:
David OBrien, Sumon Biswas, Sayem Mohammad Imtiaz, Rabe Abdalkareem, Emad Shihab, and Hridesh Rajan. 2024. Are Prompt Engineering and TODO Comments Friends or Foes? An Evaluation on GitHub Copilot. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639176

1 INTRODUCTION

Artificial intelligence (AI) has begun to aid software developers by offering a range of development tasks [8, 17, 32, 33, 36, 49], including code generation [16], comment maintenance [32, 33, 36, 57], defect resolution [33, 36, 57], and automated code review [17, 49, 57]. These intelligent coding solutions leverage AI advancements and a vast corpus of open-source software data. Microsoft's GitHub Copilot [16] is a notable example, trained on billions of lines of code from GitHub repositories using OpenAI's Codex model [7]. We select GitHub Copilot to evaluate upon in this study due to recent research's involvement [34, 37, 41] and its capabilities, as Copilot was found to achieve a 61-91% success rate in providing immediate solutions or a potential useful starting point for developers [37].

"Technical debt" (TD) is a widely recognized phenomenon in the software development industry, which was first introduced by Ward Cunningham [9]. TD refers to the long-term impact of insufficient solutions on the development process [5, 13, 26, 46, 48]. TD can include unfinished implementations, hacky workarounds, poor code quality, outdated documentation, and many other poor software quality symptoms. While TD is often described as a financial metaphor [1], it can also have a significant impact on other expenses beyond monetary means, such as time and effort required to maintain and improve the system [5, 13, 26, 46, 48].

Potdar and Shihab [42] explored the concept of "self-admitted technical debts" (SATD) which are locations where developers acknowledge the existence of a TD. The most commonly researched SATDs are those left as source code comments (e.g., TODO comments) [3, 14, 18, 28–31, 39, 42, 43, 55], although SATD has been found in other software artifacts such as issue trackers [24, 25, 52], build systems [53], code review [21], and Docker files [2]. SATD has

also been leveraged in practice [50], and SATD in practice and open source share similarities [54]. Many distinct symptoms of TD have been reported as SATD such as Requirement Debts, Code Debts, Defect Debts, Test Debts, Design Debts, and Documentation Debts [3, 42]. Prior works have been proposed to assist in technical debt management by using AI techniques to detect and classify SATD comments [18, 27–29, 31, 43, 44], classify a SATD's removal [56], and remove obsolete (previously fixed) TODO comments [15], and evaluate transformers and LLMs' applicability towards SATD repayment [35]. Specifically, this study investigates TODO comments as a subset of SATD, and thus we refer to our data as TODO comments. We also refer to the SATD broadly when discussing prior works and future works which our study can motivate.

Since TD can affect a plethora of software maintenance activities, a technique to assist in repaying the accrued technical debts would be valuable. Because TD can be disclosed in natural language through SATD [42], and GitHub Copilot has been shown to perform well upon natural language prompts [37], is it possible that SATD comments could serve as instruction in these prompts to produce less technically indebted code? In doing so, are the symptoms of the SATD reproduced or repaid in the proceeding generations? Additionally, is it possible that not all TODO comments are well fit to be included in code generation prompts? For instance, the comment `TODO: this is hacky` is very vague and describes poor qualities of pre-existing code. Meanwhile, `TODO: add divide by zero checks` is another TODO comment which describes a specific future action to be implemented. In these two instances, is it possible that there are characteristics of TODO comments such as implying an action or text clarity which better equip a SATD comment to become a prompt for code generation? The examination of code generations from prompts including TODO comments can provide insights into prompt engineering best practices to generate technical debt by analyzing successfully and unsuccessfully repaid TODO comments. However, to the best of our knowledge, *no prior works have investigated the extent to which the symptoms associated within TODO comments can be resolved by supplementing code generation prompts with the information available in these comments.*

Inspired by these questions, we curated a dataset consisting of 36,381 TODO comments found in the most recent revisions of Python repositories as of January 2022 from a dataset utilized by a prior research [11]. From this dataset, we manually examined 1,013 comments to extract a statistically significant sample of 380 fitting TODO comments with a confidence interval of 95% and a margin of error of 5%. From the 380 TODO comments, we generate 1,140 function bodies using different prompt-constructing procedures to gain initial insights into how TODO comment-including prompts affects Copilot's output. This dataset is publicly available, making it a usable resource for future research in AI-assisted software maintenance. A thorough analysis of the dataset enables the following research questions to be explored:

- **RQ1: Does the presence of TODO comments impact the quality of GitHub Copilot's generated code?** Since code generative tools such as Copilot are trained on open-source software [16] where SATD is prevalent [42], it is crucial to understand if its generated code reproduces the symptoms of TODO comments from training data, and if so, explore ways to mitigate this issue.
- **RQ2: Can GitHub Copilot generations repay developer-written TODO comments?** Given that GitHub Copilot is trained on a vast amount of software data [16], we investigate if its generated code can serve as alternative solutions which do not contain the TD symptoms of developer-written code.
- **RQ3: Can TODO comments be modified to enhance prompts which lead to generated code that repays the symptoms?** Prior work speculates whether a TODO comment can be modified to document software following the TODO comment's resolution [15]. Could this be utilized to engineer prompts which produce TODO comment-repaying solutions?

We have made the following contributions in this paper:

(1) To the best of our knowledge, the first study evaluating the applicability of prompt engineering via TODO comment inclusion/modification to assist in automatic technical debt repayment.
(2) Recommendable best practices for prompt engineering to produce code which avoids the symptoms of SATD being reproduced by code generative tools.
(3) Insights on the limitations of code generative tools and inspirations for future research on code intelligence techniques applied towards SATD repayment.
(4) A publicly available dataset consisting of 1,140 GitHub Copilot generations which future work can evaluate against to facilitate AI-assisted software maintenance.

The rest of this paper is organized as such: Section 2 discusses the methodology, Section 3 overviews the results of the specified RQs, Section 4 discusses the implications of our findings, Section 5 outlines related work, and Section 6 concludes this paper.

2 METHODOLOGY

This section describes our approach for utilizing open-source repositories to create prompts for GitHub Copilot generations.

2.1 Dataset

In order to address the research questions outlined, we have elected to utilize the Boa Language and Infrastructure [12]. Boa has been demonstrated to be a capable and efficient solution for large-scale mining of software repositories. Relevantly, OBrien et al. [39] demonstrated its capabilities of extracting self-admitted technical debt (SATD) comments from Python machine learning repositories. Therefore, we decide to also study Python repositories due to its increasing popularity in both machine learning and recent research [4, 10, 11, 39, 40]. Moreover, in this study, we have chosen to expand the scope beyond solely machine learning repositories, opting instead to utilize a dataset of diverse Python repositories provided by the Boa Language and Infrastructure as of February 2022. This dataset has been employed in prior research for the analysis of Python programming paradigms [11].

This dataset consists of repositories whose primary language is Python, was gathered from GitHub in February 2022, and consists of repositories created from 2008-2021. The repositories in this dataset were cloned in descending order of star count, resulting

in each contained repository having a minimum of 24 stars. We present the statistics for the overall dataset in Table 1. Despite the dataset comprising over 293 million file snapshots, in order to focus only on analyzing *currently existing* TODO comments (comments beginning with the word "todo", case insensitive) using GitHub Copilot, we have mined only the most recent snapshots, which encompasses a total of 23,848,176 files.

Table 1: The dataset statistics

Granularity	Amount
Projects	102,424
Revisions	32,231,939
Unique Files	63,681,580
File Snapshots	293,231,664
AST Nodes	105,512,426,611

2.2 Creating TODO Comment Dataset

In this section, we describe our approach for gathering, filtering, and sampling a dataset of Python TODO comments and docstrings. Data from this section is provided in Table 2.

In order to analyze the potential of code generative tools to reproduce and resolve issues admitted in TODO comments, we have undertaken a thorough extraction and filtering process of the TODO comments present in the Boa dataset. Through querying the most recent snapshots of Python files, we have identified a total of 36,381 TODO comments. Adhering to established protocols for discerning SATD comments as outlined in previous literature [39], we have eliminated any comments with duplicate signatures (exact same comment in the same file) and those that are not native to their respective repository (instead in a local copy of a library or package) by removing those with substrings such as "site-packages". This methodical approach ensures that all comments analyzed were created by their respective repository developers and eliminates the possibility of duplicate comments being present within a repository. This results in a dataset of 29,672 TODO comments.

We reuse a sampling method previously employed in prior studies on SATD [14, 39]. Specifically, in later sections, we continually sample from the 29,672 TODO comments until we identify 380 fitting TODO comments. This sample size is chosen with a confidence level of 95% and a margin of error of 5%. This statistically significant sample will be the focus of our analysis in the subsequent sections. In our analysis of the sampled source code functions, we found that the smallest function consisted of 0 statements (unimplemented functions), the largest comprised 197 statements, and the median function size was 8.5 statements, reflecting a wide variety in code sizes within the dataset.

2.3 GitHub Copilot Generations

In this section, we will describe the process followed to create a dataset of GitHub Copilot generations from TODO comments, documentation strings (docstrings), and function headers found in open-source repositories.

In order to evaluate differing prompts' effectiveness in resolving TODO comments, we created a local copy of Python files containing TODO comments. By generating code within a copy of the original file, we aim to simulate how developers would utilize Copilot and provide additional context to Copilot [16].

Table 2: Overview of our TODO comment preprocessing

Preprocessing Step	TODOs Remaining
Total Extracted	36,381
Unique + Native	29,672
Sampled	380

We adopt a strategy that leverages the use of docstrings to guide Copilot, as previously employed in related work [34]. In doing so, we leverage existing software data to perform prompt engineering. Docstrings, which are blocks of natural language describing the functionality, inputs, outputs, or examples of a Python function, serve as valuable information for Copilot to generate code. Therefore, in our analysis, we preserve the docstrings found in the original functions. Additionally, we also retain the function headers, which include the function name, input names, and type hints, if provided. By providing both docstrings and function headers as input, we establish a context for Copilot to generate within.

Because some samples are unfit for inclusion in this study, we continually sample from our population of 29,672 TODO comments until we achieve 380 fitting samples. A sample can be deemed unfit for many reasons such as the TODO comment appearing outside of a function body, the respective function does not contain a docstring (lacking providable context), the TODO comment is very ambiguous (e.g., the comment containing only the word `TODO`), and the TODO comment or the entire repository having been removed since the Boa dataset was created. Therefore, each sampled instance is manually labeled as being fit or unfit for this study.

Human labeling is prone to bias and mistakes. Therefore, we follow a iterative procedure [45] used in a variety of studies in software engineering to mitigate this effect [6, 20, 39]. At each iteration, two labelers would independently label 38 function bodies as to whether it is appropriate for our study or not. The comments' location and available context are all assessed at this stage. Following each iteration, the Cohen's Kappa would be measured between the labelers. Cohen's Kappa is a statistical measure to evaluate classification agreement which discards the possibility that the labelers agree randomly. A Cohen's Kappa above 0.8 is considered as "excellent" agreement. Following each iteration, the labelers would meet to discuss their disagreements, with a third author prepared to settle any disagreements left unsettled.

Through this process, the labelers achieved consistent "excellent" agreement, with Kappa values of 0.9408, 0.8177, and 0.9444 in each iteration. With consistent "excellent" agreement, the labelers independently labeled until 380 fitting samples were achieved. In total, 1,013 comments were manually investigated before 380 fitting instances were found to serve as the basis for later generations.

After gathering 380 relevant samples, we used the Visual Studio Code IDE to manually generate Copilot generations, as there was no available API for Copilot at the time of experimentation [16]. This method is consistent with previous evaluations of Copilot [37, 41]. For each triplet of information (function header, docstring, TODO comment), we generated three function bodies using 3 unique prompts with GitHub Copilot for a total of 1,140 (380 * 3) generations. We use Copilot's first provided generation following

prior work, which speculated a partial solution 61-91% of the time [37]. Although the prompt-constructing strategies used may introduce bias to our results, this study aims to provide *initial* insights into the alignment between TODO comments and effective code generation prompts. Therefore, if the prompts in this study are found to be successful, it could inspire future work to experiment with alternative processing of TODO comments for prompt construction. An example of our prompt-constructing procedure is illustrated in Figure 1. The three prompts per triplet are as follows:

(1) **DS (Docstring):** We pass only the docstring and original function header from the open-source repositories as a Copilot prompt.
(2) **DS-TD (Docstring-TODO)**: We append the TODO comment to the beginning of the docstring to construct a Copilot prompt. The decision to append at the start of the docstring was decided so that the context of the TODO comment is the first task consumed as input by Copilot.
(3) **DS-MTD (Docstring-Modified TODO)**: Prior works note that the removal of the word "TODO" from a TODO comment can suffice as the documentation after the specified change is completed [15]. Therefore, this generation preprocesses the TODO comment by discarding the word "TODO" before appending the modified TODO comment ahead of the docstring.

Evaluating or validating the generated code is a challenging task, and technical debt repayment is no different. In this paper, we follow work which evaluates human repayment of SATD and evaluate Copilot's generations with rigorous manual evaluation [55] and we involve two labelers to reduce bias. Previously, Mastropaolo et al. [34] utilized software tests from the repositories and similarity scores to evaluate their generations. However, Mastropaolo et al. [34] found that re-using these tests may not evaluate desired behaviors, and leveraging similarity metrics may produce misleading results (e.g., a correctly-behaving generation that wildly differs from ground truth causes a correct solution to have a lower score). In SATD repayment specifically, resolving SATD may involve untested aspects such as code legibility and documentation. The multitude of solutions available can introduce variability in similarity scores, making such metrics less reliable. Thus, this study leverages manual evaluation of SATD repayment.

It is important to note that the primary objective of our assessment is not to assess the technical accuracy or functional equivalence of the generated code, as this is performed by prior work [37]. Instead, the evaluation aims to determine the extent to which the symptoms associated with the TODO comments, as identified in the original code, are addressed in the generated code through varying prompts including, excluding, or modifying TODO comments. Manual labeling has been used previously to assess the repayment of a SATD [55]. Even if the generated code does not behave correctly, its generation could still guide the user to a quicker resolution of the technical debt Nguyen and Nadi [37]. However, assessing whether AI-generated solutions expedite or impede a technical debt's resolution is out of the scope of this study. Instead, our primary focus is whether the inclusion of a TODO comment influences the output of code generative tools, and whether this influence can resolve the TODO comment's symptoms. To do so, we consider the different

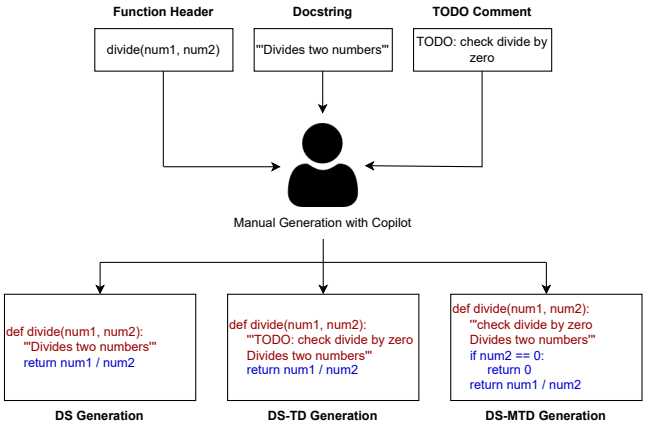

Figure 1: The process for creating three generations from open-source repository. Red text represents the input to GitHub Copilot; blue text represents the generated code.

prompt inputs to be our independent variable, and Copilot's output from varying prompts is our dependent variable. Because our experimental setup maintains the same file for the varying prompts to be placed within, any changes to Copilot's output are due to the different prompts leveraged in the respective generations.

The authors have labeled 3 questions regarding the 3 generations for each triplet of information. For edge-case samples, multiple labelers were consulted to assign an appropriate label. These questions assess whether or not the Copilot generations fix or contain the concerned TODO comment symptoms. For example, the **DS** and **DS-TD** generations in Figure 1 do not implement error checks as admitted by the original TODO comment, and therefore, they do not solve the symptoms of the TODO comment. Conversely, the **DS-MTD** generation implements error checking, and the labelers assigned this generation as repaying the TODO comment. Specifically, the three questions examined are:

(1) *Does the **DS** generation fix the symptoms of the concerned TODO comment?* In Figure 1, the **DS** generation does not implement error checks.
(2) *Does the **DS-TD** generations contain the symptoms of the concerned TODO comment?* Notably, the **DS-TD** generation in Figure 1 does not implement error checks. The cause for these patterns in explored in later sections.
(3) *Does the **DS-MTD** comment fix the symptoms of the concerned TODO comment?* In Figure 1, when the TODO comment is modified, Copilot outputs code with the specified error checks. This suggests that proper prompt modification can effectively guide Copilot towards resolving TODO comments, thus assisting in software maintenance.

This study employed a thorough labeling process to enable accurate and consistent results. A training session was held by the authors where the characteristics of TODO comments and Copilot's potential effects were discussed. Following this, utilizing the same method previously used for identifying TODO-docstring-header triplets, the authors went through four rounds of labeling, achieving improving levels of agreement as measured by Cohen's

Kappa scores of 0.508, 0.7419, 0.8488, and 0.8819. All discrepancies are discussed and resolved with a moderator prepared to settle unresolved disagreements. The third and fourth iterations which received 0.8488 and 0.8819 indicate consistent excellent agreement among the authors, which further confirms the validity and reliability of the results. To ensure transparency and reproducibility, all authors' independent labels, as well as the final settled labels, are made available in our replication package.

3 RESULTS

In this section, we delve into the findings of our examination of 1,140 GitHub Copilot generations, whose generation procedures are described in the preceding section. Our study aims to uncover the impact of TODO comments on AI code generations and assess the feasibility of using Copilot to address the symptoms of TODO comments found in open-source repositories via three different Copilot generations, i.e., input containing only the docstring (**DS**), input containing docstring and TODO (**DS-TD**), and input containing docstring and modified TODO (**DS-MTD**).

3.1 Impact of TODO Comments on Generations

RQ1: Does the presence of TODO comments impact the quality of GitHub Copilot's generated code?

RQ1 aims to investigate whether code generative tools such as GitHub Copilot can generate code containing the symptoms described in TODO comments by injecting them with the unmodified TODO comment text. Since GitHub Copilot was trained on open-source repositories, where self-admitted technical debts have been found [42], is it possible the tool may reproduce these technical debts if prompted with SATD such as TODO comments? Or, have tools such as GitHub Copilot mitigated the generation of these symptoms? In the former case, the generation of technical debt symptoms could be problematic as it would impede developer productivity, rather than expedite it. This concern is drawn from prior research [7] which comments that large language models on code may not produce high-quality code because their predictive goals are focused on reproducing the distribution of their training data rather than producing code of measured quality. Thus, the labelers assess both the **DS** and **DS-TD** generations produced by Copilot and their relation to the original developer-written code. By reading relevant information (docstrings, TODO comments, used or similar nearby code), the labelers determine whether the TODO comment symptoms are present in each generation. Additionally, the original code is also reviewed to provide perspective or additional context to the TODO comment's symptoms.

In order to understand if GitHub Copilot is capable of reproducing TODO comment symptoms, we conducted a comparison between the **DS** and **DS-TD** generations. These generations were defined in the previous section, the only difference between the two inputs is the presence of the unchanged TODO comment in **DS-TD**. Therefore, any changes in the generated code (harmful or helpful) would be due to the varying prompts. Our analysis aimed to determine if the presence of the TODO comment influenced the **DS-TD** generation to produce the technical debts described.

To provide an example of the effect in question, we illustrate an instance from our analysis in Figure 2. The TODO comment in question is `TODO: Document`, which indicates a lack of documentation in the original code where the TODO comment was found. Missing, incomplete, or out-of-date documentation is a recurring technical debt found in open-source repositories [3, 42]. In the **DS** generation which does not contain the TODO comment, Copilot generates code with source code comments throughout, despite not being explicitly instructed to prioritize documentation in the inputted docstring.

However, when the aforementioned TODO comment is included in the prompt for the **DS-TD** generation, the generated code is absent of source code comments. Ideally, the inclusion of a TODO comment would act as additional instruction for Copilot to specifically address the tasks disclosed. However, the opposite effect was observed in some cases such as Figure 2; the TODO comment injection causes Copilot to instead *reproduce* the admitted technical debt rather than repay. The cause is likely due to the nature of next-token predictors such as GitHub Copilot [7] where instead of solving the TODO comment, the code which was found near similar comments in its training data (i.e., code near comments similar to `TODO: Document` lack documentation) influences the output to instead omit the source code comments. This instance exemplifies that the presence of a TODO comment can have a negative impact on the quality of code produced.

Table 3: Confusion matrix of DS and DS-TD results.

	Does **DS** Repay?		
Does **DS-TD** Reproduce?	No	Yes	Total
No	13	53	66
Yes	285	29	314
Total	298	82	380

We present the results of our evaluation in Table 3. RQ1 is concerned with instances where the presence of a TODO comment caused a negative effect which is not implemented without the TODO comment in the **DS** Generation, such as the previously described example in Figure 2. Therefore, RQ1 focuses on cases where the **DS** generation repays the described TODO symptoms, and the **DS-TD** generation reproduces the admitted technical debt.

Figure 3 shows that the TODO comment's symptoms are resolved by the **DS** generation in 82 cases. Of the 82 cases, the symptoms are reintroduced in 29 of the respective **DS-TD** generations. Therefore, we witness 35.36% of our investigated TODO comments causing a previously repaid technical debt to be instead be reproduced in the Copilot generations when involved in the prompt. Thus, the inclusion of the TODO comment has a negative effect on the generated code quality.

Although developers in practice may not place TODO comments in their prompts, this result indicates 1) the majority of the studied TODO comments may require additional preprocessing to be repaid through code generative prompt engineering, 2) although the characteristics of Copilot's training data are not disclosed [16], Copilot's training data likely contains TODO comments since it was shown to reproduce TODO comment symptoms when prompted with these comments, and 3) TODO comment symptom-containing outputs are not currently mitigated. These findings can motivate future work, which is discussed further in Section 4.1.

```
# Load the rule image
rule_img = cv2.imread(rule_image_name, cv2.
    IMREAD_GRAYSCALE)
# Load the density image
density_image = cv2.imread(density_image_name, cv2.
    IMREAD_GRAYSCALE)
# Save the density image in temp folder
cv2.imwrite('temp/density_image.png', density_img)
return rule_img, density_img
```

```
rule_img = cv2.imread(rule_image_name, cv2.
    IMREAD_GRAYSCALE)
density_image = cv2.imread(density_image_name, cv2.
    IMREAD_GRAYSCALE)
cv2.imwrite('temp/density_image.png', density_img)
return rule_img, density_img
```

Figure 2: DS (left) and DS-TD (right) Copilot generations where DS-TD includes "TODO: Document" in the prompt.

```
elif isinstance(index, slice):
    #TODO-Should we copy the alphabet and gap character?
    new_align = Alignment(self._alphabet)
    new_align._records = self._records[index]
    return new_align
```

Figure 3: A piece of a DS-MTD generation involving the comment `TODO - See Bug 2554 for changing the __init__method` which produced an unrelated TODO comment

Finding 1: Because code generative tools can reproduce their training data, 35.36% of the 380 TODO comments studied had their symptoms reproduced when directly included in prompts.

Furthermore, other notable cases occurred when generations were being conducted in this study. Figure 3 shows the snippet of a generation which was a **DS-MTD** generation which concerned the TODO comment `TODO - See Bug 2554 for changing the __init__method` which could be considered a Defect Debt by prior works [3, 42] due to referencing a bug report. In the resulting generation shown in Figure 3, another unrelated TODO comment was generated by Copilot: `TODO - Should we copy the alphabet and gap character?` which questions the absence of functionality, thus suggesting it is a functional Requirement Debt [3].

Currently, RQ1 discussed the effects of TODO comments that were prompted (i.e., the unmodified TODO comment is included in the prompt). The TODO comment displayed in Figure 3 was generated *unprompted* (i.e., there was no information in the prompt involving a TODO comment or the generated Requirement Debt). Yet, Copilot's generation still included this unrelated TODO comment, likely caused by TODO comments in the training data related to the prompted task. In other words, because similar code in the training data contained TODO comments, these unrelated symptoms were included in the generated solution. This further emphasizes the need for mitigating SATD such as TODO comments from code intelligence tools, since even in situations where TODO comments are not explicitly mentioned in the prompt, the harmful symptoms and TODO comments are still producible.

Finding 2: Code generative tools can reproduce TODO comments and their symptoms unprompted (without explicit instruction).

3.2 Repaying TODO Comments through Copilot

RQ2: Can GitHub Copilot generations repay developer-written TODO comments?

RQ2 delves into the potential impact of GitHub Copilot on self-admitted technical debts in software development. Despite the best efforts of developers, the creation and maintenance of software can be challenging, with factors such as lack of experience, unawareness, and time constraints leading to the accumulation of technical debt [5, 9, 13, 26, 46, 48]. RQ2 investigates whether GitHub Copilot can help mitigate these challenges by producing code that repays the technical debts admitted with existing TODO comments in open-source software. Our intuition is that in providing developers with fast, potentially high-quality code generated from a vast corpus of software data, code-generative tools can help repay the TODO comment symptoms accumulated by developers. To answer RQ2, we use the same labeling results presented in RQ1, but do not compare with the results of the **DS-TD** generation, we only investigate successful **DS** generations.

Our methodology for RQ2 involves analyzing the **DS** generations produced by GitHub Copilot against the TODO comments found in the original code. By comparing the generated code to the original code, we can analyze whether the generated code addresses and potentially resolves the admitted symptoms as is done by prior work [55]. If the generated code is found to address the symptoms, it may serve as an alternative to human-written code or may accelerate the repayment of these specific technical debts.

Figure 4 provides such an illustration of the potential impact of GitHub Copilot's code generation on a TODO comment. In the original developer-written code, a TODO comment is present, admitting a lack of error checking on the `filehash` parameter, thus considered to be a Defect Debt [3, 42]. However, the **DS** generation which re-uses the same developer-written docstring has ensured to implement such a check without explicit specification to do so in the **DS** prompt. This suggests that Copilot generates code which addresses technical debts not prioritized by developers. Furthermore, this illustrates the potential for Copilot to automatically resolve technical debts without additional prompt engineering.

The results presented in Table 3 offer an indication into the potential of Copilot to address TODO comment symptoms. 82 (21.57%) instances were identified where the **DS** generations produced by Copilot effectively address the symptoms of the TODO comment, indicating that the code generated by Copilot is not burdened by the same technical debt as developer-written code. This suggests that Copilot has the potential to be a valuable tool for software maintenance, offering an alternative solution for replacing or assisting in the repayment of technically indebted code.

Finding 3: In our dataset, 21.57% of TODO comment symptoms were labeled as resolved in the **DS** generation, relying solely on the information provided within the developer-written docstring.

```
#TODO error checking on filehash
#build the query string
query = "download/v1/" + filehash
return query
```

```
#validate parameters
if len(filehash) != 64:
    raise Exception("Invalid file hash")
query = "download/v1/" + filehash
return query
```

Figure 4: A comparison between developer-written code (left) and the DS Copilot generation (right).

We further investigate the 82 **DS** generations that were identified to at least address the TODO comments that developers implemented in the original code. In doing so, we aim to identify themes of technical debts that developers have encountered that Copilot generations repay in its **DS** generations. These themes are thus technical debts which Copilot-written code may assist in repaying. We find three recurring themes in our studied 82 repayments: defect-handling, additional support, and better assumptions.

Previous work on SATD types [3, 42] has classified a taxonomy. One such instance is the *Defect Debt*, characterized as known defects' resolutions deferred to a later time due to competing priorities. In a similar vein, we find that TODO comments involving the handling of defects is a task which **DS** Copilot generations can perform without any specific instruction. Defect-handling examples of comments resolved by **DS** generations include `TODO: add std OSError attributes or pick more approp. exception` which motivates more fitting exception throwing and `TODO: Warning of unhandled characters` which self-admits a currently unimplemented warning. Both of these comments are repaid without including the respective TODO comments in the **DS** prompts.

In our dataset, we find a reoccurring theme involving supporting new features as a task that **DS** Copilot generations were able to provide. Such examples include `TODO: Support URLS that don't start with 'static'` and the comment `TODO: this has to be improved now that we also support other datasets that may not have list.txt`, our studied **DS** generations for these examples do not depend on these hardcoded values or restrictions.

Finally, we find that developers' TODO comments indicate poor assumptions which results in questionable decisions later in its lifetime. For example, the comment `TODO err_crit is never used?` self-admits a technical debt where an input parameter is never used. However, the **DS** generation does not ignore `err_crit` and instead involves it in the computation.

Finding 4: Defect-handling, additional support, and better assumptions are identifiable themes in which code generative tools can assist in mitigating the TODO comment symptoms.

```
elif FLAGS.platform == 'nccl2':
    exe = fluid.ParallelExecutor(
        use_cuda=True,
        loss_name=self.debug_keys[0],
        main_program=self.get_main_program(FLAGS),
        exec_strategy=fluid.ExecutionStrategy())
```

Figure 5: DS-MTD generation whose prompt includes the modified TODO comment "parallel executor".

3.3 Repaying TODO Comments with Prompt Engineering

RQ3: Can TODO comments be modified to enhance prompts which lead to generated code that repays the symptoms?

RQ3 builds on RQ2 by exploring how Copilot can be used to address TODO comments. Since RQ2 finds that Copilot generations can resolve symptoms of TODO comments via generating from docstrings alone, RQ3 investigates whether prompt engineering can enable TODO comment repayment. Building on prior work's speculation [15], we experiment with removing the word "TODO" from TODO comments to include them as documentation-like instruction as part of Copilot's docstring input. This approach is referred to as **DS-MTD** generations. For RQ3, we manually examine the output of the **DS-MTD** generations and compare it with the original developer-written code as done in prior generations. However, we also refer back to the previous **DS** and **DS-TD** generations to see if the modifications of the prompts caused the generated code to differ. If the produced code is identical, then the labels should reflect this as well. However, in cases where the code differs, the authors investigated the difference and determine whether the **DS-MTD** generation resolves the TODO comment symptoms.

We exemplify this procedure and motivate its results with the Copilot generation shown in Figure 5. Note that for presentation's sake, the resulting generation contains more code than illustrated, but only the code relevant to this finding is shown. However, all generations in their entirety can be found in our replication package. In Figure 5, the concerning TODO comment is `TODO: parallel executor` which self-admits the symptoms of a lack of a parallel executor functionality, thus this can be considered a functional Requirement Debt [3, 42]. In the **DS** and **DS-TD** generations, no parallel executor is implemented either from a lack of specification or misalignment involving the word "TODO" as RQ1 previously speculated. When the word "TODO" is removed from this comment, the text becomes `parallel executor`. The elimination of the term "TODO" alters the comment's semantics, and might improve the generated output. Rather than producing code which is similar to code near comments like `TODO: parallel executor` in its training data, which likely lacks the desired parallel executor, the comment `parallel executor` is now expected to be describing code that meets the desired specification in Copilot's training data. Thus, the **DS-MTD** generation contains the illustrated relevant code in Figure 5 whereas no parallel executors were generated in the **DS** nor **DS-TD** generations. Thus, even a minor adjustment to a TODO comment can positively impact its generation's quality when included in a prompt.

Figure 6 depicts the results of our manual assessment of **DS** and **DS-MTD** generations. Notably, our generations involving leveraging and processing the TODO comments to provide modified context and instruction resulted in 40 (10.53%) additional TODO

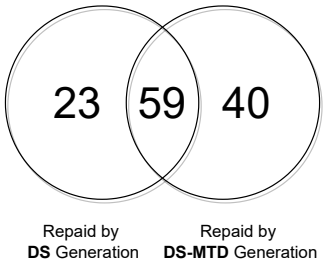

Figure 6: Comparison of TODO comments repaid by DS and DS-MTD generations.

comment repayments beyond the 82 (21.58%) **DS** generations' repayments. In total, 122 (32.11%) of the 380 TODO comments were labeled as addressed by Copilot in our study. Therefore, with just minor modifications (removing the word "TODO"), code generation was able to repay more technical debts. This is because of the additional effective information available in TODO comments for guiding software maintenance. Yet, the presence of the word "TODO" in unmodified TODO comments can reverse the semantics of the comment, thus prompting code generative tools for code *without* the desired effects. Therefore, the following observation is made: *although directly injecting TODO comments into prompts can reproduce the symptoms, preprocessing these comments before prompt engineering can enable next-token code generation techniques to repay the technical debts.* Even though the modification to produce a **DS-MTD** is simple, this initial study can motivate future works which can perform heavier preprocessing to extract effective instruction and specification from TODO comments to generate symptom-avoiding code. This observation sets our work apart from previous work investigating the robustness of Copilot given similar prompts [34], since our study finds that modified SATD within prompts can enable code generations involving less technical debt.

Finding 5: Omitting "TODO" in comments enables code generative tools to address 10.53% additional comments, highlighting potential for preprocessing for TODO technical debt repayment.

Motivated by these results, we sought to understand the characteristics of the comments which allowed Copilot to be successful or unsuccessful in repaying the described symptoms. Two of the authors reviewed the comments' texts separated by having been repaid by a **DS** or **DS-MTD** generation in this study (122 total) and those not repaid by either (258 total). In doing so, the authors identified recurring characteristics about the information present in these comments. In the end, all 380 comments were reviewed and discussed by both authors to assign labels, and the resulting datasets are available in our replication package. These characteristics are described, exemplified, and explained how the comments are fitting or unfitting for inclusion in prompt engineering in Table 4 and 5. It is important to note that a comment can receive multiple, or none of the characteristics in this labeling process.

Characteristics found to likely contribute to effective prompt engineering can assist in creating large-scale datasets of SATDs which are likely repayable by code generation for future studies and can provide prompt engineering best practices. Characteristics found to likely harm effective prompt engineering can inspire future work towards heavier TODO comment preprocessing to make these TODO comments prompt-ready or alternative strategies for automatic technical debt repayment. Additionally, future researchers can build classifiers to automatically separate TODO comments to aid the repayment by AI tools.

Specifically, we observe that TODO comments that describe concrete actions, rather than identifying poor symptoms of code resulted in many repaid instances. This is because the task of code generation is to reproduce code from its training data which the prompts describe. If a TODO comment describes an actionable instruction, then this previous disclosure of an absence of functionality can become additional requirement for code generation. Meanwhile, TODO comments which only describe symptoms of code instead cause code generative tools to reproduce these poor qualities rather than generate a solution to mitigate these qualities. Fucci et al. [14] provides a classification of SATD comments' content, including classes such as poor implementation choices and misalignment, which we speculate to be the symptoms rather than solution-providing comments, e.g., the comments shown in Table 5. Additionally, we find that supplying extra contextual information, rationales, or future considerations can also assist in technical debts in being repaid by Copilot with our various prompt constructing techniques. This is due to the additional details present in these comments that can guide Copilot to generate the intended quality or functionality. However, Copilot has additional difficulties in cases such as TODO comments questioning software or TODO comments which have relationships to the original body of code. These observations can motivate currently unexplored future work on additional code intelligence tasks applied to SATD such as code editing and code Q&A. We discuss this further in Section 4.2.

4 DISCUSSION

This paper studies TODO comments specifically, since TODO comments were found to be prevalent in all projects studied in Huang et al. [18] and consist of over half of the SATD studied. Additionally, Gao et al. [15] provides speculations on TODO comments specifically which we form our prompt engineering strategies around. Although our work is on TODO comments specifically, our results can motivate SATD mitigation and AI-assisted technical debt repayment broadly.

4.1 Mitigation of SATD Generation

Our experiments demonstrate that code-generative tools can generate TODO comment symptoms prompted or unprompted. While code generation tools aim to enhance developer productivity, the presence of SATD can impede development and add complexity to software systems [42]. To address this issue, we propose preprocessing and postprocessing techniques to mitigate this reaction.

We propose a preprocessing technique for cleaning the training data of these tools. This technique involves removing any code that contains instances of SATD from the training data. Previous research on source code comment completion used this approach to prevent SATD comments from being completed by the downstream

Table 4: Helpful characteristics found in the 122 repayable SATD comments

Quality Name	Description	How it can Affect Prompts	Example(s)	Percentage
Concrete Action	SATD comments may describe desired implementation or action.	Since code generation produces code according to instructions, these SATD comments are well-aligned with the goals of code generation.	`todo: support sparse matrix!!`, `todo: check if the name does not contain forbidden characters:`	65.57%
Contextual Info	SATD comments may include contextual details such as where or when a change is to occur.	SATDs which provide adequate context can guide code generative tools to produce relevant code.	`todo: fix code to fail with clear exception when filesize cannot be obtained`	32.79%
Rationale	SATD comments may provide reason for the described repayment.	SATDs which detail the rationale of its repayment can provide non-functional requirements for its generation.	`todo: would it be more efficient using a dict or hash values instead`	6.56%
Future Consideration	SATD comments may imply considerations of changes.	The DS generations may make these future considerations without being specified to.	`todo: need tau possibly here`	27.87%

Table 5: Harmful characteristics found in the 258 non-repaid SATD comments

Quality Name	Description	How it can Affect Prompts	Example(s)	Percentage
Symptom	SATD comments disclose poor quality code instead of concrete actions.	Inclusion of these comments in prompts leads to generative tools producing poor-quality code instead of solutions.	`todo: untested for glms?`, `todo: too much slop permitted here impossible, todo# too long?`	14.34%
Proximity	SATD comments refer to code nearby in the original body.	Without access to the original code, the relationship between these comments and specific code segments is lost, hindering code generation's performance.	`todo: fix next line`, `todo: clean this up`, `todo: complete this documentation`	32.95%
Question	SATD comments question poor qualities of code.	When injected into prompts, they result in code with these questionable qualities instead of solutions.	`todo: remove redundant attributes and fix the code that uses them?`, `todo: how to accommodate regression?`	15.12%

model [32]. We recommend applying this approach to code generation as well, to ensure that SATD comments are not produced by the generative model. Additionally, removing SATD from the training data may also prevent the model from reproducing SATD-affected code in later generations. This modification could lead to the generation of code that fully resolves the symptoms of injected unmodified TODO comments, unlike observations made in RQ1.

A second approach is a postprocessing mechanism that involves incorporating SATD identification into Copilot's ranking procedures. SATD detection has been approached by many prior works [18, 27–29, 31, 43, 44]. Currently, Copilot generates multiple candidate generations for a given prompt, then uses a ranking procedure to determine the order in which they are presented to the user [16]. By removing candidates that are identified to contain SATD from consideration or adjusting the ranking procedure to prefer candidates without SATD, the likelihood of SATD being output can be reduced. This modification has the advantage of not requiring any pre-existing code generative models to be retrained.

4.2 AI-Assisted TD Maintenance & Repayment

As shown in our results for RQ2 and RQ3, there are plenty of TODO comments that Copilot was unable to repay. We find that although GitHub Copilot and SATD exist at the bridge of natural language and programming language, there are characteristics of a TODO comments that do not align well with a generative prompt. In these instances, we find that AI agents which generate code may not be the only tool which can assist in alleviating the symptoms of a self-admitted technical debt.

Previously, we have speculated that SATD which expresses a question may cause difficulties for code generation to repay. In these instances, a code Q&A agent trained upon varying circumstances may best assist in resolving these comments as opposed to code generation. In our dataset, we find that developers leave TODO comments whose questions involve API applicability, express unknown developer knowledge, disclose doubts about current implementations, and ask questions about their specific software system. An example of a question about API applicability is `TODO: refactor. Can I use disag_upsample()?`, which can be assisted by a code Q&A trained on API documentation to guide developers on its intended usage and prevent misusage. TODO comments involving unknown developer knowledge include `TODO: figure out swap!` and `TODO: how to properly limit max number of function calls?`, which can be assisted by code Q&A trained on development tutorials or Stack Overflow posts to educate inexperienced developers on best practices. TODO comments disclosing doubts on current implementations include `TODO(zhiting): is it okay to have stand-alone random generator?` and `TODO: is m.ClassDeclaration enough?` Meanwhile, an example of a TODO comment questioning a specific software system is `TODO: unused?` Both types of comments doubting current implementations and questioning aspects of their specific system can be assisted

by code Q&A which learns facts about a software system to act as a senior developer to answer the rationale behind design decisions.

Previously it was discussed that comments such as `TODO - should these be in if clause?` and `TODO - This potentially needs to be expanded` refer to code in proximity to the concerning comment. Because our experiments discard the original code to generate potentially technical debt-free code, this relationship is lost, and thus these comments do not translate well to Copilot prompts. However, there are code intelligence techniques involving "code editing" [57] where code is refined according to specific instructions. In these scenarios, the original code is used as an additional channel of input. Thus, our study can motivate future work which could replicate our experiments with code editing techniques to compare to our code generation performances. Speculatively, these techniques may also perform well on comments which only disclose symptoms, since common solutions to these symptoms may be learnable and reapplicable.

Finally, some TODO comments were out of scope of our experiments, such as `TODO make another listener for target-changed`, which indicates a task of creating another function. Our study only generated function bodies which our TODO comments were found in, and thus this comment was out of the scope of our experiments. Future work could investigate system-wide SATD resolution and its relationship with the existing surrounding code using code intelligence techniques. Additionally, future works can expand upon our work by incorporating code quality metrics to evaluate the quality of the generated code. Additionally, our analysis is only performed with the most recent revisions of the repositories and does not consider how TODO comments and their corresponding code evolve over time. Therefore, future work can explore how code intelligence tools can assist in software evolution and improve developers' interaction with Copilot by exploring how often developers approve or modify generated solutions.

5 THREATS TO VALIDITY

Internal validity: To ensure the accuracy of our labeled data, we adopted an iterative process where two authors independently labeled the dataset and resolved disagreements with a third author. Cohen's Kappa measure was employed to verify that the agreement between the authors' labels was not due to chance. Through several iterations, the two authors consistently achieved excellent agreement, following a well-established approach used in prior studies involving manual investigation [6, 20, 39, 45]. Although we systematically generate code, it is possible that users with differing skill levels may use Copilot in different ways, creating a feedback loop between users and Copilot to produce the best solutions.

Although we only study Python repositories, the mined dataset consists of over 100,000 projects that have at least 24 stars on GitHub. Additionally, the function bodies studied contain 0-194 source code statements, reflecting a wide variety of task complexities and potential topics. Additionally, the choice to leverage docstrings in our prompts led our studied instances to include only Python functions with docstrings. Our prompt-construction procedures could introduce bias, considering that we may have overlooked potentially more effective prompts during the study. However, we ensured a systematic approach by applying the procedure to all 380 fitting TODO comments. Furthermore, we have made all generations and labels available in our replication package, promoting transparency and open science.

During the labeling process, the authors labeled whether a GitHub Copilot generation resolved the symptoms of a concerning TODO comment found originally in an open-source repository. Although it is possible that the generated code is functionally different than the original open-source code, the correctness of the generation and other dynamic factors are not evaluated in this study. Previous works have evaluated the correctness of GitHub Copilot generations [37] to find up to 91% of the time, Copilot could provide a partial fix. Instead, we have evaluated whether the generations contain or resolve the symptoms of the concerning TODO comment depending on its presence or modification in the prompt. Regardless of correctness or loss of functionality, the generations were labeled according to their relationship with the concerning TODO comment as done for human SATD repayment in prior works [55]. Prior work also found that harmful code can be produced from Copilot [41], however, we do not evaluate this alongside TODO comment resolution. Additionally, it is possible that the TODO comment could be misleading, obsolete, or inaccurate. Although this could lead to misinterpretation, multiple labelers were consulted to decide on a difficult TODO comment label. Finally, since the labelers were aware of the prompting techniques used, this knowledge may have unconsciously added bias to their labels.

External validity: We selected GitHub Copilot for its prominence in previous works [34, 37, 41] and its extensive training by OpenAI and GitHub, despite some implementation details being unavailable due to its proprietary nature. Nevertheless, the techniques discussed in Section 4 for improving preprocessing and post-processing can be adapted to any code-generative tool to achieve SATD-free code generations. Moreover, our findings can likely apply to other code generative tools since current state-of-the-art approaches, like Copilot, utilize next-token predictors for code generation [7]. Hence, we argue that our claims regarding reproducing and repaying SATD can be generalized to other code-generative tools.

6 RELATED WORKS

6.1 Management of Technical Debt

"Technical debt" was coined by Ward Cunningham, comparing software development to debt, emphasizing the importance of repayment [9]. Numerous studies investigate its impact on software products and developers [5, 13, 26, 46, 48]. Tom et al. [48] conducted a survey revealing TD's influence on developers' morale, productivity, quality, and project risk. For technical debt management [5], Li et al. [26] surveyed various aspects, including identification, measurement, prioritization, prevention, monitoring, repayment, documentation, and communication. Prior works proposed TD measurement techniques like SQALE [22, 23]. This paper utilizes GitHub Copilot to aid in technical debt repayment, specifically addressing technical debts admitted through TODO comments in different stages of ML pipeline [4].

Potdar and Shihab [42] introduced "self-admitted technical debt" (SATD), revealing intentional disclosures by developers in software artifacts. SATD is prevalent in issue trackers [24, 25, 52], code review

[21], build systems [53], and Docker files [2]. Most SATD research, including this work, focuses on exploring SATD in source code comments (e.g., TODO comments) [3, 14, 18, 28–31, 39, 42, 43, 55].

Our study leverages AI techniques to aid TODO comment management. Prior research explores similar AI-based approaches for managing self-admitted technical debt (SATD) through DL and text mining [18, 27–29, 31, 43, 44]. Zampetti et al. [56] introduces a DL-based method to classify SATD into 6 removal strategies [55] for developer assistance. Addressing the complexities of SATD removal status [30, 55] and outdated comments [47, 51], Gao et al. [15] proposes an approach that tracks "obsolete" TODO comments using comment text, source code, and commit messages. Mastropaolo et al. [35] is the most similar work to ours, as they examine how LLMs (pretrained, finetuned, and chatbot) can be used to repay SATD. However, our work differentiates itself from Mastropaolo et al. [35] by proposing prompt engineering for GitHub Copilot and evaluating how often the symptoms of TODO comments are reproduced in the downstream code generations.

6.2 GitHub Copilot Evaluations

Code intelligence tasks, boosted by deep learning [8, 17, 32, 33, 36, 49], include code generation [16], comment maintenance [32, 33, 36, 57], defect resolution [33, 36, 57], and automated code review [17, 49, 57]. In this study, we assess GitHub Copilot's performance in the automatic repayment of TODO comments through prompt engineering. However, other aspects of Copilot have been evaluated by prior works [19, 34, 37, 41].

Mastropaolo et al. [34] finds that semantic-preserving changes in prompts can affect the output of Copilot, thus questioning the robustness of Copilot. In contrast, our work examines how well modifying prompts with TODO comments can lead to code generation that resolves specified technical debt symptoms. Pearce et al. [41] utilized GitHub Copilot to generate code for high-risk CWE scenarios, revealing 40% vulnerability in insecure prompts. Nguyen and Nadi [37] evaluated Copilot's generation on LeetCode and found initial correct answers in 27%-57% of cases, with 61%-91% of time providing a potentially useful starting point for developers. Imai [19] examines how well Copilot serves as a pair programmer.

7 CONCLUSION

The title of this paper questions whether prompt engineering and TODO comments are friends or foes. To provide an answer, this paper has manually gathered 380 TODO comments from open-source repositories to generate 1,140 GitHub Copilot generations. First, we find that unmodified TODO comments can cause Copilot to produce technically indebted code. However, we also find that TODO comments possess valuable information that can serve as actionable instructions or additional specifications in these prompts. Depending on the characteristics of these comments, our study exemplifies how prompt engineering can be leveraged to address more instances of TODO comments' symptoms. The examination of successfully and unsuccessfully repaid TODO comments enabled us to provide prompt engineering best practices for producing less technically indebted code. Ultimately, we demonstrate how code generation can perform towards automatic technical debt repayment and motivate future work for the improvement of code generative tools and the application of other code intelligence tools towards automatic technical debt repayment.

DATA AVAILABILITY

The scripts and resulting TODO comment data is publically available on Zenodo [38].

ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation under Grants CCF-15-18897, CNS-15-13263, CNS-21-20448, CCF-19-34884, and CCF-22-23812. All opinions are those of the authors and do not reflect the views of sponsors. Generative AI was used to revise sections of this paper's writing.

REFERENCES

[1] Areti Ampatzoglou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Paris Avgeriou. 2015. The financial aspect of managing technical debt: A systematic literature review. *Information and Software Technology* 64 (2015), 52–73. https://doi.org/10.1016/j.infsof.2015.04.001

[2] Hideaki Azuma, Shinsuke Matsumoto, and Yasutaka Kamei. 2022. An empirical study on self-admitted technical debt in Dockerfiles. *Empirical Software Engineering* 27 (2022). https://doi.org/10.1007/s10664-021-10081-7

[3] Gabriele Bavota and Barbara Russo. 2016. A large-scale empirical study on self-admitted technical debt. In *International Conference on Mining Software Repositories*. ACM, 315–326.

[4] Sumon Biswas, Mohammad Wardat, and Hridesh Rajan. 2022. The Art and Practice of Data Science Pipelines: A Comprehensive Study of Data Science Pipelines In Theory, In-The-Small, and In-The-Large. In *ICSE'2022: The 44th International Conference on Software Engineering* (Pittsburgh, PA, USA).

[5] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, Raghvinder Sangwan, Carolyn Seaman, Kevin Sullivan, and Nico Zazworka. 2010. Managing Technical Debt in Software-Reliant Systems. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research* (Santa Fe, New Mexico, USA) *(FoSER '10)*. Association for Computing Machinery, New York, NY, USA, 47–52. https://doi.org/10.1145/1882362.1882373

[6] Junming Cao, Bihuan Chen, Chao Sun, Longjie Hu, Shuaihong Wu, and Xin Peng. 2022. Understanding Performance Problems in Deep Learning Systems. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 357–369. https://doi.org/10.1145/3540250.3549123

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[8] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2021. An Empirical Study on the Usage of Transformer Models for Code Completion. *IEEE Transactions on Software Engineering* PP (11 2021), 1–1. https://doi.org/10.1109/TSE.2021.3128234

[9] Ward Cunningham. 1992. The WyCash Portfolio Management System. *SIGPLAN OOPS Mess.* 4, 2 (dec 1992), 29–30. https://doi.org/10.1145/157710.157715

[10] Luca Di Grazia and Michael Pradel. 2022. The Evolution of Type Annotations in Python: An Empirical Study. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 209–220. https://doi.org/10.1145/3540250.3549114

[11] Robert Dyer and Jigyasa Chauhan. 2022. An Exploratory Study on the Predominant Programming Paradigms in Python Code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*.

Association for Computing Machinery, New York, NY, USA, 684–695. https://doi.org/10.1145/3540250.3549158

[12] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *2013 35th International Conference on Software Engineering (ICSE)*. 422–431. https://doi.org/10.1109/ICSE.2013.6606588

[13] Neil A. Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L. Nord, and Ian Gorton. 2015. Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. 50–60.

[14] Gianmarco Fucci, Nathan Cassee, Fiorella Zampetti, Nicole Novielli, Alexander Serebrenik, and Massimiliano Di Penta. 2021. Waiting around or job half-done? Sentiment in self-admitted technical debt. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 403–414. https://doi.org/10.1109/MSR52588.2021.00052

[15] Zhipeng Gao, Xin Xia, David Lo, John Grundy, and Thomas Zimmermann. 2021. Automating the Removal of Obsolete TODO Comments. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 218–229. https://doi.org/10.1145/3468264.3468553

[16] GitHub. 2022. *GitHub Copilot Your AI pair programmer.* Retrieved August 8, 2022 from https://github.com/features/copilot

[17] Vincent J. Hellendoorn, Jason Tsay, Manisha Mukherjee, and Martin Hirzel. 2021. Towards Automating Code Review at Scale. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1479–1482. https://doi.org/10.1145/3468264.3473134

[18] 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 Softw. Engg.* 23, 1 (feb 2018), 418–451. https://doi.org/10.1007/s10664-017-9522-4

[19] Saki Imai. 2022. Is GitHub Copilot a Substitute for Human Pair-Programming? An Empirical Study. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 319–321. https://doi.org/10.1145/3510454.3522684

[20] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 510–520. https://doi.org/10.1145/3338906.3338955

[21] Yutaro Kashiwa, Ryoma Nishikawa, Yasutaka Kamei, Masanari Kondo, Emad Shihab, Ryosuke Sato, and Naoyasu Ubayashi. 2022. An empirical study on self-admitted technical debt in modern code review. *Information and Software Technology* (2022), 106855.

[22] Jean-Louis Letouzey. 2012. The SQALE method for evaluating Technical Debt. In *2012 Third International Workshop on Managing Technical Debt (MTD)*. 31–36. https://doi.org/10.1109/MTD.2012.6225997

[23] Jean-Louis Letouzey and Michel Ilkiewicz. 2012. Managing Technical Debt with the SQALE Method. *IEEE Software* 29, 6 (2012), 44–51. https://doi.org/10.1109/MS.2012.129

[24] Yikun Li, Mohamed Soliman, and Paris Avgeriou. 2020. Identification and Remediation of Self-Admitted Technical Debt in Issue Trackers. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 495–503. https://doi.org/10.1109/SEAA51224.2020.00083

[25] Yikun Li, Mohamed Soliman, and Paris Avgeriou. 2022. Identifying self-admitted technical debt in issue tracking systems using machine learning. *Empirical Software Engineering* (2022), 495–503. https://doi.org/10.1007/s10664-022-10128-3

[26] Zengyang Li, Paris Avgeriou, and Peng Liang. 2015. A systematic mapping study on technical debt and its management. *Journal of Systems and Software* 101 (2015), 193–220. https://doi.org/10.1016/j.jss.2014.12.027

[27] Zhongxin Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. 2018. SATD Detector: A Text-Mining-Based Self-Admitted Technical Debt Detection Tool. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 9–12. https://doi.org/10.1145/3183440.3183478

[28] Rungroj Maipradit, Bin Lin, Csaba Nagy, Gabriele Bavota, Michele Lanza, Hideaki Hata, and Kenichi Matsumoto. 2020. Automated Identification of On-hold Self-admitted Technical Debt. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 54–64. https://doi.org/10.1109/SCAM51674.2020.00011

[29] Everton Maldonado, Emad Shihab, and Nikolaos Tsantalis. 2017. Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt. *IEEE Transactions on Software Engineering* (2017), to appear.

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

[31] Everton da S. Maldonado and Emad Shihab. 2015. Detecting and quantifying different types of self-admitted technical Debt. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*. 9–15. https://doi.org/10.1109/MTD.2015.7332619

[32] Antonio Mastropaolo, Emad Aghajani, Luca Pascarella, and Gabriele Bavota. 2021. An Empirical Study on Code Comment Completion. *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2021), 159–170.

[33] Antonio Mastropaolo, Nathan Cooper, David Nader Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2022. Using Transfer Learning for Code-Related Tasks. *IEEE Transactions on Software Engineering* (2022), 1–20. https://doi.org/10.1109/TSE.2022.3183297

[34] Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliverto, and Gabrielle Bavota. 2023. On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Australia) *(ICSE '23)*. Association for Computing Machinery, New York, NY, USA.

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

[36] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks. 336–347. https://doi.org/10.1109/ICSE43902.2021.00041

[37] Nhan Nguyen and Sarah Nadi. 2022. An Empirical Evaluation of GitHub Copilot's Code Suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/3524842.3528470

[38] David OBrien. 2023. Replication package for "Are Prompt Engineering and TODO Comments Friends or Foes? An Evaluation on GitHub Copilot". https://zenodo.org/records/10460738

[39] David OBrien, Sumon Biswas, Sayem Mohammad 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* (Singapore, Singapore).

[40] David OBrien, Robert Dyer, Tien Nguyen, and Hridesh Rajan. 2024. Data-Driven Evidence-Based Syntactic Sugar Design. In *2024 IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal). https://doi.org/10.1145/3597503.3639580

[41] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. 754–768. https://doi.org/10.1109/SP46214.2022.9833571

[42] Aniket Potdar and Emad Shihab. 2014. An Exploratory Study on Self-Admitted Technical Debt. In *International Conference on Software Maintenance and Evolution*. IEEE Computer Society, 91–100.

[43] Xiaoxue Ren, Zhenchang Xing, Xin Xia, David Lo, Xinyu Wang, and John Grundy. 2019. Neural Network-Based Detection of Self-Admitted Technical Debt: From Performance to Explainability. *ACM Trans. Softw. Eng. Methodol.* 28, 3, Article 15 (2019), 45 pages.

[44] Barbara Russo, Matteo Camilli, and Moritz Mock. 2022. WeakSATD: Detecting Weak Self-Admitted Technical Debt. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 448–453. https://doi.org/10.1145/3524842.3528469

[45] C.B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering* 25, 4 (1999), 557–572. https://doi.org/10.1109/32.799955

[46] C. Seaman and Y. Guo. 2011. Measuring and Monitoring Technical Debt. *Advances in Computers* 82 (2011), 25–46.

[47] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /*icomment: Bugs or Bad Comments?*/. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 145–158. https://doi.org/10.1145/1294261.1294276

[48] Edith Tom, Aybüke Aurum, and Richard Vidgen. 2013. An exploration of technical debt. *Journal of Systems and Software* 86, 6 (2013), 1498–1516. https://doi.org/10.1016/j.jss.2012.12.052

[49] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using Pre-Trained Models to Boost Code Review Automation. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2291–2302. https://doi.org/10.1145/3510003.3510621

[50] Carmine Vassallo, Fiorella Zampetti, Daniele Romano, Moritz Beller, Annibale Panichella, Massimiliano Di Penta, and Andy Zaidman. 2016. Continuous Delivery Practices in a Large Financial Organization. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 519–528. https://doi.org/10.1109/ICSME.2016.72

[51] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. A Large-Scale Empirical Study on Code-Comment Inconsistencies. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 53–64. https://doi.org/10.1109/ICPC.2019.00019

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

[53] Tao Xiao, Dong Wang, Shane McIntosh, Hideaki Hata, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. 2022. Characterizing and Mitigating Self-Admitted Technical Debt in Build Systems. *IEEE Transactions on Software Engineering* 48, 10 (2022), 4214–4228. https://doi.org/10.1109/TSE.2021.3115772

[54] Fiorella Zampetti, Gianmarco Fucci, Alexander Serebrenik, and Massimiliano Di Penta. 2021. Self-admitted technical debt practices: a comparison between industry and open-source. *Empirical Software Engineering* 26 (11 2021). https://doi.org/10.1007/s10664-021-10031-3

[55] Fiorella Zampetti, Alexander Serebrenik, and Massimiliano Di Penta. 2018. Was Self-Admitted Technical Debt Removal a Real Removal? An In-Depth Perspective. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 526–536.

[56] Fiorella Zampetti, Alexander Serebrenik, and Massimiliano Di Penta. 2020. Automatically Learning Patterns for Self-Admitted Technical Debt Removal. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 355–366. https://doi.org/10.1109/SANER48275.2020.9054868

[57] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2023. CoditT5: Pretraining for Source Code and Natural Language Editing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 22, 12 pages. https://doi.org/10.1145/3551349.3556955