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

Christina Li
University of Waterloo
Waterloo, Canada
christina.li1@uwaterloo.ca

1 What is the Problem Being Solved?

This paper investigates whether including or modifying TODO comments (a frequent type of self-admitted technical debt) helps or hinders GitHub Copilot in generating high-quality code. Specifically, it looks at whether unaltered TODOs cause the tool to reproduce their issues, or if prompt engineering—such as removing "TODO" and clarifying the instructions—improves Copilot's output and helps repay technical debt.

2 What is the New Idea They Are Proposing?

The authors propose leveraging TODO comments as potential prompts for Copilot but highlight that unmodified TODO comments may lead to code that repeats the same poor practices developers wanted to fix. They systematically compare three prompts (just the docstring, docstring + TODO, and docstring + modified TODO) to see which leads to code that best addresses the issues described in the original comment. They also provide evidence that simply removing the word "TODO" and presenting the rest of the comment as a natural-language requirement can yield more helpful code generations.

- TODO Comments sometimes Help, sometimes hurt.
- Modified TODOs could lead to more technical debt repayment.
- Copilot can repay technical debt even without TODOs.
- Not all TODOs leads to useful codes.

3 Positive Points

- **Dataset and Sample Quality:** The authors build a large, high-quality dataset of Python TODO comments from active GitHub repositories and carefully filter them, ensuring meaningful, context-rich samples. This thorough approach bolsters the reliability of their findings.
- **Actionable Insight:** Their observation that removing "TODO" from a comment can help Copilot produce more relevant solutions is straightforward and beneficial for both researchers and practitioners. This insight could be easily integrated into existing workflows.
- **Explores a Critical Issue in AI-Assisted Development:** Many AI-generated solutions contain hidden technical debt. This research provides first insights into how AI handles TODO-driven development.

4 Negative Points

- **Single Programming Language Constraint:** Focusing only on Python code with docstrings may limit how far these findings extend to other languages or settings where documentation is less standard.
- **Dataset Bias and Representativeness:** The study only considered well-documented code from relatively popular repositories ($\geq$ 24 stars). The exclusion of trivial or ambiguous TODO notes (e.g. comments that just say "TODO" with no details) also sidesteps what happens when the prompt lacks clear guidance.
- **Narrow Interpretation of Prompt Engineering:** Prompt engineering typically encompasses a broad range of techniques (rephrasing instructions, providing examples, system messages, etc.) to steer an LLM. However, this paper only focus on TODO comments, which can be only a trivial part of prompting issues.
- **Experimental Design Limitations:** The study's methodology, while systematic, may not reflect real-world developer behaviour. In practice, developers might leave a TODO in place (not necessarily at the top), or iteratively prompt the AI rather than providing a fully formed docstring upfront. The one-shot prompt per function might overlook how developers interact with Copilot in multiple passes or with partial code.
- **Labelling Subjectivity and Bias:** The manual labelling process, though rigorous in achieving high inter-rater agreement, still carries subjectivity.

5 Future Work

AI-Driven Refactoring for Technical Debt. One possible direction is to create tools powered by artificial intelligence that can automatically uncover and remediate instances of technical debt. Such tools would analyze code-bases for structural or design weaknesses, then either suggest or apply refactorings that prevent these shortcomings from accumulating over time.

Expanding Copilot's Reach Across Languages. Another area to investigate involves studying how the insights from a Python-centric context generalize to other programming languages with different syntax, paradigms, and documentation standards. This broader view would help determine whether the same strategies for prompt engineering are effective on a wide range of development ecosystems.

Assessing Long-Term Maintenance Impact. While short-term code fixes are valuable, the real test lies in evaluating Copilot's influence on long-term maintenance. Future research could involve tracking how Copilot-generated solutions evolve alongside a codebase over multiple releases or iterations, shedding light on the tool's sustained impact on software quality and maintainability.

AI-Assisted Bug Fixing Beyond TODOs. Finally, it would be worthwhile to examine how Copilot and similar AI tools can assist with a broader range of bug fixes that are not explicitly flagged by TODO comments. This includes scenarios in which defects or improvement opportunities arise from testing, bug reports, or external issues rather than from self-admitted technical debt.

6 Rating

3/5: Despite the paper's strong dataset quality, practical insight on removing the word "TODO," and its focus on an important issue in AI-driven development, the narrow scope of evaluating only Python repositories with docstrings and the somewhat constrained method of prompt engineering limit broader applicability. Additional factors, such as potential dataset bias and the inherently subjective nature of manual labeling, further reduce confidence in the findings.

7 Discussion Points

- How do we balance writing for the AI vs. writing for fellow humans?
- Should there be guidelines to ensure prompt engineering techniques don't inadvertently encourage bad documentation or design habits?
- Are we approaching a future where commenting becomes a form of coding (prompt engineering) to steer AI, and is that a positive trend for productivity and code quality?

8 In-Class Discussion

During the discussion of this paper in class, several students expressed skepticism about Copilot's reliance on pattern matching. One student pointed out that large language models (LLMs) often copy code from examples containing TODO statements, sometimes reproducing the same issues without truly "understanding" them. Others argued that users should not expect Copilot to be entirely consistent or relevant for every situation, as the tool can merely mirror what it has seen in its training data. Another person noted that to use Copilot effectively, developers should employ a conversational or supervised approach—consistently reviewing and verifying generated code rather than blindly trusting it. In that vein, one participant added they would not feed an entire repository to Copilot and ask it to find bugs, as the process would be prone to oversight. There was also a remark that prompt engineering sometimes feels unscientific, with uncertain guidelines.

Questions also arose about the paper's methodology. Some students wondered why the authors did not go further in modifying or rephrasing the TODO comments, suggesting that simple word removals might not be enough to improve Copilot's responses. They pointed out that re-engineering prompts with more details, or systematically rewording the TODO statements, could potentially yield better outcomes. On the topic of code quality, people highlighted that the paper never clearly defined what "high-quality" code looks like—there are many metrics for quality, and it's unclear which ones best measure whether Copilot helped fix or avoid technical debt.

Another point of debate was the paper's level of transparency. Some noted that the prompts were not found in the authors' provided package, leading to confusion about how the experiments were conducted. This fuelled doubts over how the work achieved acceptance in a top conference, with a few students indicating they were unimpressed by the reported results. Lastly, there was a conversation about the labelling process, where only two people labeled the data. One participant thought that was too few for such a subjective task, though the professor replied that more labellers typically increase disagreements, and a smaller, more consistent team can sometimes be more practical.