ARE PROMPT ENGINEERING AND TODO COMMENTS FRIENDS OR FOES? AN EVALUATION ON GITHUB COPILOT

3/20/25

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

UNIVERSITY OF WATERLOO | FACULTY OF MATHEMATICS

Presenter: Christina Li

Primary Focus

- 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.
  - ✅ Determine the extent to which the symptoms associated with the TODO comments.
  - ❌ Technical Accuracy of the generated code

Research Questions

- RQ1: Does the presence of TODO comments impact the quality of GitHub Copilot's generated code?
- RQ2: Can GitHub Copilot generations repay developer-written TODO comments?

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

New Ideas Proposed

(1) **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.

Methodology - Dataset & Prompt Engineering

Dataset:

- Extracted 36,381 TODO comments from 102,424 Python repositories.
- Iterative method **filtered to 380 relevant samples** using Boa infrastructure.

Prompt Engineering Strategies:

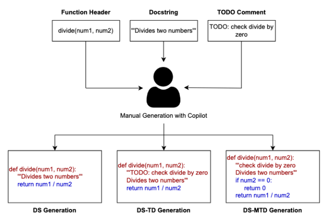
- **DS:** Uses DocString only.
- **DS-TD:** Uses DocString + TODO comment.
- **DS-MTD:** Uses DocString + modified TODO (removes "TODO" for clearer instruction).

Generated Code Analysis:

- **1,140 function bodies manually labeled** (380 per prompt type).

Table 2: Overview of our TODO comment preprocessing

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

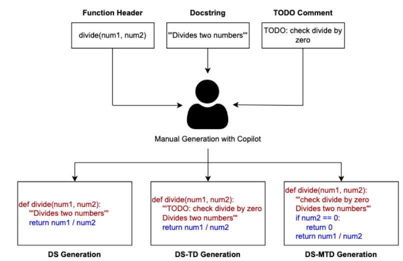


Methodology - Dataset & Prompt Engineering

(1) Does the DS generation fix the symptoms of the concerned TODO comment?

(2) Does the DS-TD generations contain the symptoms of the concerned TODO comment?

(3) Does the DS-MTD comment fix the symptoms of the concerned TODO comment?



Finding 1 - TODO Comments Sometimes Help, Sometimes Hurt

- Including TODO comments in the prompt (DS-TD) leads to **35.36% of generated code reproducing TD**.

- Many cases **repeat** poor coding practices instead of resolving them.

- **More structured TODOs** help Copilot generate better solutions.

```python
# 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
```

```python
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 Top **and DS-TD** Bottom **Copilot generations where DS-TD includes "TODO: Document" in the prompt.**

Finding 2 - Modified TODOs Lead to More Technical Debt Repayment

- DS-MTD prompts (modified TODOs) improved TD repayment by 10.53%.

- Removing the word "TODO" and **rephrasing the instruction clearly** resulted in better AI-generated solutions.

- Highlighting potential for preprocessing for TODO technical debt repayment.

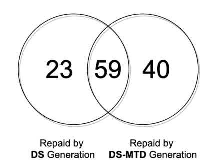


Figure 6: Comparison of TODO comments repaid by DS and DS-MTD generations.

Finding 2 - Modified TODOs Lead to More Technical Debt Repayment

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%

Finding 3 - Copilot Can Repay TD Even Without TODOs

- 21.57% of generated code repaid TD even in DS prompts (without TODOs).
- Copilot learns patterns from training data and sometimes suggests better code even without being asked.
- AI-based code generation is improving, but still unreliable.

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

Finding 4 - Not All TODOs Lead to Useful Code

- Harmful TODOs include:
  - Ambiguous statements ("Fix later")
  - Referencing unknown context ("Optimize this")
  - **Unclear questions** ("Does this work?")

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%

Positive Points

1. **Large-Scale Study with Strong Data Analysis**
   - The study uses **a large dataset from real-world repositories**.
   - Manual evaluation of **1,140 function bodies** adds reliability.
2. **Clear Practical Implications**
   - Highlights **how AI models interact with software maintenance issues**.
   - Findings can guide **prompt engineering** and **AI-assisted code review tools**.
3. **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**.

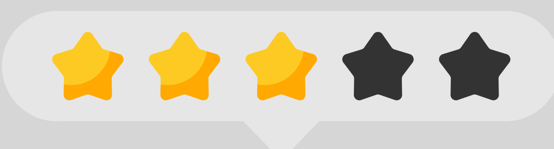
Negative Points

1. **Limited to Python, Github Repo and TODO Comments**
   - Study only evaluates **Python repositories**, so findings **may not apply to other languages and other platforms.**
   - Other **types of AI-generated code quality issues** were not considered.
2. **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.

Negative Points Cont'd

- **3. Dataset Bias & Representativeness:**
  - the study only considered well-documented code from relatively popular repositories (≥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.
- **4. 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.
- **5. Labeling Subjectivity and Bias:**
  - The manual labeling process, though rigorous in achieving high inter-rater agreement, still carries subjectivity.

RATINGS



Future Work

- Develop AI-driven refactoring tools to detect and fix TD automatically.
- Analyze Copilot's effectiveness across multiple programming languages.
- Evaluate Copilot's performance on long-term software maintenance tasks.
- Explore AI-assisted bug fixing beyond TODO comments.

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?