Enhancing Debugging Skills of LLMs with Prompt Engineering

Keyu He*, Max Li*, Joseph Liu*
University of Southern California
{frankhe, qianqili, jliu7350}@usc.edu

Abstract

This paper presents a comprehensive study on improving the debugging capabilities of Large Language Models (LLMs) like GPT-3.5, focusing on the application of prompt engineering techniques. We explore the efficacy of few-shot learning, chain-of-thought prompting, and a baseline zero-shot model in enhancing LLMs' ability to debug code. Utilizing static and dynamic evaluation metrics, the study rigorously assesses the debugging proficiency of these models. By introducing different types of bugs, including procedural and language model-generated errors, and applying varied prompting strategies, we provide a deeper understanding of LLMs' debugging capabilities. The results provide insights into the limitation of debugging capabilities of GPT-3.5 Turbo, even with the assistance of various prompting techniques. Source code of our evaluation method and bug generation techniques are in GitHub repository: https://github.com/FrankHe2002/CSCI 499FinalProject

1 Introduction and Motivation

Large language models (LLMs), such as the GPT (Brown et al., 2020) and CodeLlama (Rozière et al., 2023) families, have shown promise in automating coding tasks. However, their debugging skills remain limited and relatively untested. Simultaneously, debugging has become an important aspect of development: Poor quality software in the US in 2018 costed approximately $2.8 trillion (Krasner, 2021). Additionally, the popularity of code completion services powered by LLMs, such as GitHub Copilot, is rising. These services are not perfect, and their code may need to be debugged as well. Our research aims to evaluate and enhance the debugging capabilities of LLMs through few-shot and chain-of-thought approaches. We will establish a baseline zero-shot prompt with no chain of thought for comparative evaluation. By utilizing both static and dynamic evaluation metrics, we aim to quantitatively assess the debugging proficiency of these LLMs and pave the way for their effective use in debugging tasks.

2 Hypothesis

In this research, we posit that employing chain-of-thought (Wei et al., 2023) prompt engineering techniques or using few-shots (Fei-Fei et al., 2006) will substantially elevate the debugging performance of large language models. We anticipate that this tailored prompting approach will yield improvements in multiple evaluation metrics, serving as a quantifiable measure of debugging effectiveness. These expected outcomes were rigorously compared against the baseline.

3 Methodology

3.1 Overview

Our base experiment design is as follows: First, we gathered a dataset of pairs of buggy code and their correct versions. Then we ask our model to debug the code using one of four prompt variations: zero-shot without chain-of-thought (CoT), zero-shot with CoT, few-shot without CoT, and few-shot with CoT. We take the outputted code and analyze it using both static metrics, such as CodeBLEU (Ren et al., 2020) and our novel CodeROUGE/CodeF1, and dynamic metrics, in the form of test cases.

3.2 Dataset Collection and Bug Generation

To create our dataset, we had two primary requirements. One, each data pair needed to be relatively independent. While the real-world distribution of bugs include many points where significant amounts of context are needed, this is very difficult to prompt. Therefore, we need a dataset where the information necessary to find the bug can be found in a relatively small number of tokens to be passed into the model. Second, each code sample needed

to be runnable. In order to do dynamic analysis, we need to be able to automate testing of each sample on real test cases. An existing dataset that satisfies both of these constraints could not be found. For example, Bugs2Fix, while similar, is not runnable and thus wouldn't be able to be dynamically analyzed (Tufano et al., 2019).

With all this in mind, Java solutions to Leetcode problems were chosen as our primary data source. Java is a popular language, meaning the model has some exposure to its syntax. It is also more syntactically complex than a language like Python, which allows us to test on common usage errors. The solutions, and therefore bug-free versions of code, were easily acquired from GitHub repositories; we specifically used `AnasImloul/Leetcode-Solutions` (Imloul, 2023). The second step was the creation of the buggy versions of each pair. To accomplish this, we introduce a procedural bug generator and investigate LLM bug generation as an alternative.

3.2.1 Procedural Bug Generation

Our bug generator generates bugs based on common programming mistakes. In all cases, the generator locates some token and replaces or removes it. These include, among others: Replacing array access indices, removing syntactically important characters, and replacing a boolean comparison operator. In total, we have six cases, out of which 4 are logical errors as the code compiles, 1 is a syntax error, and the last is the negative sample (that is, no bugs). We include negative sampling to simulate the possibility of a user asking the LLM to debug code that has no bug. A variable number of bugs are added to a given piece of code to generate its buggy version; one example can be found in Figure 1, and detailed information can be found in appendix A.1.

3.2.2 Language Model Bug Generation

An alternative to a procedural approach to bug generation is using a language model. As large language models are trained on extensive real-world datasets, it can represent a wide spectrum of programming errors that are commonly made by human programmers. This ensures that our testcases more closely match the real-world distribution of bugs. An example can be found in Figure 2, and prompt design for bug generation can be found in Appendix A.2. Note that the LLM bugs are more complex, with certain keywords missing (`break`) or function calls replaced (`add`, `remove`).

```
while (l < r && nums[l] == nums[l + 1]) {
    l++;
}
while (l < r && nums[r] == nums[r - 2]) {
    r--;
}
l++;
r--;
```
Correct solution

```
while (l < r && nums[l] == nums[l + 1]) {
    l++;
}
while (l < r && nums[r] == nums[r - 1]) {
    r--;
}
l++;
r--;
```
Generated bug

Partial Code Snipped

Figure 1: Example of procedural bug, where an array access index has been modified.

```
return (int) res;
```
Correct solution

```
return (int) (res % mod + 1);
```
Generated bug

```
Integer next = zeros.ceiling(map.get(rains[i]));
if (next == null) return new int[0];
res[next] = rains[i];
zeros.add(next);
```
Correct solution

```
Integer next = zeros.ceiling(map.get(rains[i]));
if (next == null) return new int[0];
res[next] = rains[i];
zeros.remove(next);
```
Generated bug

```
if (count[m] > 0) {
    ans = i + 1;
    break;
}
```
Correct solution

```
if (count[m] > 0) {
    ans = i + 1;

}
```
Generated bug

Partial Code Snipped

Figure 2: LLM-generated bugs, demonstrating complex errors

3.2.3 Data Filtering and Cleaning

To clean the data, the following steps were taken:

- **Formatting**: The LeetCode code snippets were formatted using IntelliJ IDEA to fit a standard format. This ensures that CodeBLEU tokenization, which is based on whitespace characters, is consistent. LLM output (that is, the debugged code) also seems to follow a standard format that the model has learned, and we try to make sure that the inputs match that also.

- **Description Filter**: All datapoints without descriptions are removed from consideration. Leetcode solution code snippets are short and contain little information about the problem, typically using variable names such as `i`, `j`, `k`. As such, descriptions are important for the model to determine what exactly constitutes a bug.

- **Length Filter**: We remove 1 sample whose solution is long; it runs the risk of generation over the model's context limit, which would degrade performance.

3.3 Model Selection

We selected GPT-3.5 Turbo as the primary model for debugging code. The model excels in understanding and generating human-like text, and can thus be enhanced by prompting techniques like chain of thought and few-shot learning. These techniques have been shown to improve its performance on a range of other tasks. It additionally offers a good balance between reasoning ability and pure memorization, which is significant risk for its larger counterpart GPT-4. Both models have seen Leetcode problems before, and the higher the parameter count, the more likely it is to remember the answer from training data instead of debugging it.

3.4 Baseline and Prompt Design

The baseline for our study is established using the zero-shot, non-chain-of-thought debugging performance of GPT-3.5 Turbo. In this baseline scenario, the model is provided solely with the buggy code and its corresponding problem description. Comparative analysis is conducted between this baseline and three alternative prompts: zero-shot with chain of thought, few-shot without chain-of-thought, and few-shot with chain-of-thought. Details of our prompt designs are in appendix A.3. Note that for few-shot, we provide 5 examples.

3.5 Evaluation Metrics

Code debugged by the LLM was evaluated on two types of metrics:

- Static evaluation, where it is evaluated based on its similarity to the ground truth (correct code)

- Dynamic evaluation, where it is run on test cases to evaluate output correctness.

We discuss both in more detail.

3.5.1 Static Evaluation

The first metric we use is CodeBLEU:

$$\begin{aligned}\text{CodeBLEU} = \alpha \cdot \text{BLEU} \\ + \beta \cdot \text{BLEU}_{\text{weight}} \\ + \gamma \cdot \text{Match}_{\text{ast}} + \delta \cdot \text{Match}_{\text{df}}\end{aligned} \tag{1}$$

CodeBLEU evaluates the similarity between two snippets of code using four factors: The original BLEU score, weighted n-gram match, syntactic Abstract Syntax Tree match, and semantic data-flow match. We chose CodeBLEU as it more accurately represents unique properties of code. First, code has a strict format and unique instructions which have no ambiguities. This is unlike natural language, and thus any metric designed for language, including BLEU and ROUGE, would be flawed. Second, code has a limited vocabulary. While variable names can change, the most common tokens would be keywords such as `int`, `while`, or `public`. The weighted n-gram match is similar to BLEU, but modified to give more weight to these keywords. This more accurately represents the layout and instructions of the code, rather than the exact variable names used. Lastly, code can be represented as a syntax tree, as opposed to the sequential structure of natural language. The syntactic AST match checks for code structure without taking into account variable names or values at all, and the semantic data-flow match evaluates the inputs and outputs (Ren et al., 2020).

To compute the score for some pair of buggy code and debugged code, we label the ground truth C_0, the buggy code C_b, and debugged code C_d. We can compute two scores

$$S_b = \text{CodeBLEU}(C_0, C_b) \tag{2}$$

$$S_d = \text{CodeBLEU}(C_0, C_d) \tag{3}$$

where S_b represents the score of the buggy code compared to ground truth and S_d is the score of the

debugged code compared to ground truth. Then, the net improvement of the debugging is the change in scores:

$$\Delta S = S_d - S_b \tag{4}$$

With this value, we can then say that $\Delta S > 0$ means the model was able to debug effectively; $\Delta S = 0$ represents that debugged code is approximately the same as bugged code, and $\Delta S < 0$ means the debugged code performed worse. When analyzing results, we will be averaging ΔS over all tested samples.

This method has two main limitations: First, due to the nature of our bugs, S_b, S_d will both be quite large: The majority of the code will still be identical between the buggy and debugged versions. In instances where the model successfully identifies and corrects the errors, ΔS may be quite small. Conversely, swapping two lines would result in a large dip in ΔS. We therefore use dynamic evaluation as a second metric (discussed in 3.5.2), where the exact ordering of instructions does not matter.

Second, CodeBLEU is precision based. It evaluates how much of the debugged code is relevant or correct in relation to the reference code. It follows that CodeBLEU penalizes generations not in ground truth, even if it's possibly useful. An example is the generation of helper functions. To address this, we introduce CodeROUGE, a recall-based technique that instead penalizes code found in ground truth that is missing in the debugged code. This distinction is crucial as it focuses on the extent to which the model's output captures all the relevant aspects of the reference solution. Similar to ROUGE (Recall-Oriented Understudy for Gisting Evaluation, Lin (2004)), which has been effectively used in evaluating text summarization, CodeROUGE offers an alternative perspective on a model's debugging capabilities. This is especially important in cases where the bug is a single missing character. As such, CodeROUGE serves as a complementary metric to CodeBLEU.

Computing CodeROUGE is quite simple: We swap out the various matches in CodeBLEU for a recall-based match, as opposed to precision. Consider the simplified case of BLEU, where we define $C(s, y)$ to be the number of times s appears at a substring of y, y as the reference (the ground truth) and $\hat{y}$ as the candidate (the buggy code). Then the modified n-gram precision can be written as:

$$p(\hat{y}, y) = \frac{\Sigma_s \text{min}(C(s, \hat{y}), C(s, y))}{\Sigma_s C(s, \hat{y})} \tag{5}$$

and the modified n-gram recall, which we use for CodeROUGE, can be written as:

$$r(\hat{y}, y) = \frac{\Sigma_s \text{min}(C(s, \hat{y}), C(s, y))}{\Sigma_s C(s, y)} \tag{6}$$

Note that the only difference is that we are counting over the reference in the denominator. We perform similar replacements in the remaining three factors to compute CodeROUGE.

As we have precision and recall, we can compute an F1 score also. This score, which we call CodeF1, offers a balanced number that does not penalize extra code as heavily, but also doesn't ignore syntactically important missing characters in generated code. We define CodeF1 also as the sum of four factors, where each factor is the harmonic mean between its recall and precision based variants. For the modified n-gram match, for example, we define the corresponding component:

$$\text{F1}(\hat{y}, y) = \frac{2 \cdot \overline{p}'(\hat{y}, y) \cdot \overline{r}'(\hat{y}, y)}{\overline{p}'(\hat{y}, y) + \overline{r}'(\hat{y}, y)} \tag{7}$$

where $\overline{p}'$ and $\overline{r}'$ are BLEU scores, including both the geometric mean over multiple n-grams and the brevity penalty. We then perform a weighted sum over the four components like CodeBLEU.

3.5.2 Dynamic Evaluation

Beyond the static analysis provided by CodeBLEU, CodeROUGE, and CodeF1, we incorporate dynamic analysis, specifically focusing on the percentage of test cases passed by actually running the code. This aspect of our methodology is particularly important, as it addresses a key limitation in static code evaluations. In instances where GPT has made substantial modifications to the original code, these changes often result in lower scores from static evaluation metrics, despite potentially better runtime performance or correctness. By executing the code and measuring its performance against a set of test cases, we can more accurately assess its functional correctness. This dynamic analysis, therefore, serves as a crucial counterbalance to static metrics, ensuring that our evaluation of the model's debugging effectiveness is not only based on syntactic and semantic correctness but also on the practical, real-world functionality of the debugged code.

Status	Score
Accepted	1
Compilation Error	0
Other	Proportion of Cases Passed (e.g. 0.5 for 30/60 passed)

Table 1: Dynamic Evaluation Scores

To evaluate performance at runtime, an automated script was developed that takes each sample and tests its debugged and buggy version on Leetcode's website. The resulting status is scraped and converted to a score based on Table 1. These scores have a minimum of 0 and a maximum of 1. Similarly to CodeBLEU, the exact value does not matter; instead, we wish to investigate whether the debugged code did better on average. Therefore, we compute the difference:

$$\Delta S_{LC} = S_{LC,d} - S_{LC,b} \tag{8}$$

where $S_{LC,d}, S_{LC,b}$ represent the Leetcode scores from Table 1 for debugged and buggy scores, respectively. As with BLEU, a positive number represents improvement.

4 Results & Discussion

4.1 Static Evaluation Results

We evaluated each of the four prompts using CodeBLEU, CodeROUGE, and CodeF1 over a dataset of approximately 1700 code samples, and the results are located in Table 2. For each prompt type, we inserted the relevant buggy code and problem description into the template listed in appendix A.3. After trimming the results and verifying that the output is valid, it is evaluated using the three metrics. These scores are then averaged over all samples; recall that positive numbers signify improvements, while negative numbers signify worse outputs.

	Zero-Shot, no CoT	**Zero-Shot, with CoT**
ΔS_{CB}	-0.1072	-0.2712
ΔS_{CR}	-0.1143	-0.2828
ΔS_{CF1}	-0.1123	-0.2801
	Few-Shot, no CoT	**Few-Shot, with CoT**
ΔS_{CB}	-0.1291	-0.2353
ΔS_{CR}	-0.1374	-0.2475
ΔS_{CF1}	-0.1352	-0.2445

Table 2: Static Evaluation Scores

Our first observation is that generally speaking, outputs are more dissimilar than inputs; that is, the model is generally making the code worse. Additionally, the change is relatively large, with scores differing by around 0.2 on average. We plot the distributions of our baseline and the few-shot with chain-of-thought CodeF1 scores in black (Figure 3). We notice a large and heavy tail which seems to be pulling down the average. Further analysis reveals that the vast majority of these were outputs that had a different number of lines of actual code than the input. That is, after removing blank lines, the debugged code had been extensively modified by the model. Removing these "mismatched" points, which account for around 35% of our output, shrinks the tail significantly and results in far better results (Table 3); the filtered results are shown in green. This behavior occurs with different prompts as well. As a result, we make the observation that GPT-3.5 Turbo is a very opinionated model - it likes to write code a certain way, and even when asked to only debug existing code, will often rewrite portions to fit its understanding. As CodeBLEU, CodeROUGE, and CodeF1 do not take this into account, it results in lower scores. Using an alternative model, such as GPT-4, which has been trained on a more extensive dataset, is a potential solution to addressing this problem. Alternatively, a more targeted prompt may be able to limit GPT's formatting opinions, although the extent to which this can help is unknown.

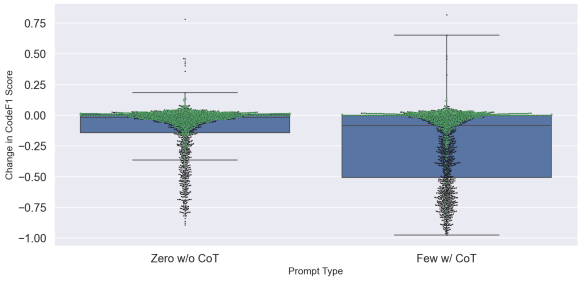

Figure 3: CodeF1 Score Distribution

	Zero-Shot, no CoT	**Zero-Shot, CoT**
ΔS_{CF1}	-0.1123	-0.2801
$\Delta S_{CF1,Filtered}$	-0.0207	-0.0291
	Few-Shot, no CoT	**Few-Shot, CoT**
ΔS_{CF1}	-0.2445	-0.1352
$\Delta S_{CF1,Filtered}$	-0.0219	-0.0299

Table 3: ΔS_{CF1} before/after mismatch removal

Secondly, we compare results with and without chain-of-thought, and with and without few-shot learning. Surprisingly, chain-of-thought seems to

generally perform worse when added in all metrics, with a lower ΔS score than their non-CoT counterparts, whether with or without few-shot. We hypothesize that this is primarily a result of formatting issues. After removing the mismatched points, the scores between CoT and without CoT are much closer. GPT-3.5 also reformatted more samples than were filtered; only samples with mismatched line numbers were removed, while many samples had reorganized contents in the same number of lines. However, it is difficult to filter out even more: The line between debugging and reformatting is blurry in many cases, and even a human would have a hard time distinguishing.

Another possible explanation is that chain-of-thought "distracts" the model from the output format it is supposed to copy. It is reasonable to believe that in the training distribution, explanations of code are typically followed by code formatted in a standard way. The model then has a higher probability of just following the format of its training distribution rather than what it has been given.

Our second comparison is between few-shot and zero-shot. The results seem to be mixed: When the model receives a chain-of-thought prompt, few-shot seems to improve results. However, without CoT, zero-shot performs better. We could not find any numerical explanation as to why this is the case. We hypothesize that this may be a combination of two factors: First, with CoT, the model learns what kind of errors to be looking for. The few-shot serves its purpose of showing the model what it should be focusing on, and therefore improves results. In zero-shot, on the other hand, the model is asked to explain without examples. It does so out of its training distribution instead, which contains much more complex bugs as shown in 3.2.2 and Figure 2. It tries to find complex explanations for a simple bug, and ends up attempting to "fix" valid code.

Lastly, we analyze the difference between CodeBLEU and CodeROUGE. For the most part, the differences seem to be quite similar, with ΔS for CodeROUGE being marginally lower than its CodeBLEU counterpart. This roughly means that the model's output is shorter and more conservative. Investigating samples, however, does not seem to yield significant observations; a regression between the CodeBLEU and CodeROUGE scores also yields an $R^2 \approx 0.9975$, showing that the two are highly correlated. We will primarily use CodeF1 as it is able to represent both metrics well without significant information loss.

4.2 Dynamic Evaluation Results

We now move on to dynamic evaluation. Due to limitations of the Leetcode website, evaluations take significantly longer. The scores (Table 4) are therefore averaged over a random selection of around 70 problems. Note that all 4 prompt variations use the same selection of problems.

	Zero-Shot, no CoT	**Zero-Shot, with CoT**
ΔS_{LC}	-0.0109	0.0167
	Few-Shot, no CoT	**Few-Shot, with CoT**
ΔS_{LC}	-0.0170	0.0137

Table 4: Dynamic Eval Scores

The results for the Leetcode evaluation are much closer to the hypothesis; when CoT is included, we achieve positive scores, showing there is some improvement in model performance. That is, even while CodeF1 scores were worse than their non-CoT counterparts, the CoT prompts are able to generate code that fixes bugs more effectively. This discrepancy can be explained by formatting errors, as mentioned in the previous section; formatting affects CodeF1 scores, but not code execution flow. However, we do note the smaller sample size of the Leetcode evaluation, so this number may need to be refined.

Secondly, we note that few-shot seems to decrease performance across the board. Investigating the samples, few-shot seems to be worse at locating syntax errors than its zero-shot counterpart. However, as there are too few samples to work with, this may be a result of chance rather than a meaningful correlation.

4.3 LLM Generated Bugs

Lastly, we run a fast analysis of LLM generated bugs to investigate their feasibility. The bugs are generated according to section 3.2.2. We then ask it to debug the code it generated using the same prompts as before. We notice a score improvement in all 4 categories of around 0.07 points over the procedural bugs. Plotting the scores additionally resulted in a much lighter tail than before. We believe that this has two primary causes: First, the bugs that the procedural algorithms produces are more likely to be out-of-distribution, as GPT-generated bugs are, by definition, in-distribution. Second, the lighter tail could be caused by formatting: GPT is unlikely to reformat code that it generated itself,

resulting in higher scores across the board. It is evident that in order to evaluate this performance correctly, we will need to use a different language model to minimize the effects of a favorable generated bug distribution.

4.4 Evaluation Metrics

Overall, the CodeROUGE/CodeBLEU/CodeF1 metrics seem to be too sensitive to formatting to accurately represent code performance. Leetcode, on the other hand, only works on smaller datasets due to time constraints. Additionally, we noticed that Leetcode is a very binary metric, with the majority of samples receiving a 1 or a 0. Code that seems to compile, for example, often receive 0's due to array access errors, which leaves very little room for partial correctness. It may be useful to investigate other potential avenues for evaluation metrics. For example, approaching this problem from a background of formal verification, which is exclusively focused on correctness of code, may yield insights in this direction.

4.5 Related Work

4.5.1 Chain-of-Thought Assists LLM Reasoning

Liu et al. (2023b) and Wei et al. (2023) investigate the potential of chain-of-thought prompts in making complex reasoning more accessible to LLMs. They share a common goal of enhancing the model's ability to perform tasks that necessitate advanced logical and sequential thinking. This research aligns with our work by emphasizing the importance of step-by-step reasoning and logical problem solving, a key aspect we explore in debugging tasks with large language models. Their approach of using chain-of-thought instructions to facilitate complex reasoning tasks offers insights into potential methods for enhancing debugging skills in language models.

In addition, the paper "Improving ChatGPT Prompt for Code Generation" explored various prompt design strategies for enhancing code generation (Liu et al., 2023a). In particular, they recursively improve prompts by repeatedly modifying and combining their best-performing prompts. This is a potential path for further exploration.

4.5.2 Pretrained Models for Coding Tasks

The paper "CodeBERT: A Pre-Trained Model for Programming and Natural Languages" (Feng et al., 2020) and "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation" (Wang et al., 2021) demonstrate two LLMs trained on coding tasks, enhancing their understanding of code structures. Code T5 uses a unified pre-training approach, combining denoising sequence-to-sequence training with identifier-aware tasks. CodeBERT's training approach enables it to comprehend code in a way that mirrors human developers. Both of these models are promising targets for further evaluation.

4.5.3 Code Completion Fails with Bugs

Dinh et al. (2023) reveals a significant challenge for Large Language Models (LLMs) like CodeGen and InCoder in code completion tasks. Specifically, LLMs' performance sharply declines in scenarios where the context contains buggy code. These insights suggests that the difficulty observed in GPT-3.5 for debugging tasks may be linked to their inherent struggles with handling the flawed code segments that they are asked to debug, a problem that persists even in models specifically fine-tuned for coding tasks.

5 Conclusion

This research provides insights into the limitations of Large Language Models (LLMs), particularly GPT-3.5, in performing debugging tasks. Despite employing various prompt engineering techniques, including few-shot learning and chain-of-thought prompting, our results indicate that these models struggle to effectively debug code. The study highlights the challenges faced by LLMs, such as their tendency to reformat code, which impede their debugging efficiency. The use of metrics like CodeF1 and Leetcode evaluation quantify these limitations and demonstrate potential avenues for improvement. These findings underscore the need for a more nuanced approach to leveraging LLMs in software development, particularly in tasks requiring precise and logical code correction. Moving forward, there is a clear opportunity to explore alternative models, refine prompt engineering methods, and integrate more sophisticated evaluation metrics to enhance the debugging capabilities of LLMs. We hope this work serves as a foundation for further research, guiding future efforts towards developing LLMs that can more effectively assist in complex programming tasks like debugging.

6 Acknowledgements

We extend our gratitude to Dr. Swabha Swayamdipta for offering crucial guidance and insights that shaped our methodology. Our appreciation also goes to Avi Thawani, whose recommendations on evaluation metrics beyond CodeBLEU were invaluable. We are grateful to Mozhdeh Gheini for steering the direction of our project.

References

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners.

Tuan Dinh, Jinman Zhao, Samson Tan, Renato Negrinho, Leonard Lausen, Sheng Zha, and George Karypis. 2023. Large language models of code fail at completing code with potential bugs.

Li Fei-Fei, R. Fergus, and P. Perona. 2006. One-shot learning of object categories. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(4):594–611.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages.

Anas Imloul. 2023. Leetcode solutions.

Herb Krasner. 2021. The cost of poor software quality in the us: A 2020 report. Technical report, Consortium for Information & Software Quality (CISQ), USA.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Chao Liu, Xuanlin Bao, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan. 2023a. Improving chatgpt prompt for code generation.

Hanmeng Liu, Zhiyang Teng, Leyang Cui, Chaoli Zhang, Qiji Zhou, and Yue Zhang. 2023b. Logicot: Logical chain-of-thought instruction-tuning data collection with gpt-4.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models.

A Appendix: Dataset Details

Generated bugs fall into one of five categories, or the negative case. Examples of each of the five cases are shown below.

A.1 Procedural Bugs

1. **Replace array index with another value**

```
UnionFind(int size) {
    parent = new int[1];
    rank = new int[size];
    for (int i = 0; i < size; i++) {
        parent[i] = i;
        rank[i] = 1;
    }
}
```
Buggy solution

```
UnionFind(int size) {
    parent = new int[size];
    rank = new int[size];
    for (int i = 0; i < size; i++) {
        parent[i] = i;
        rank[i] = 1;
    }
}
```
Correct solution

Figure 4: Replace array index with another value

2. **Remove a syntactically important character**

```
if (p.y > 0 && visited[p.x][p.y - 1 == false) {
    visited[p.x][p.y - 1] = true;
    q.add(new Pair(p.x, p.y - 1));
}
```
Buggy solution

```
if (p.y > 0 && visited[p.x][p.y - 1] == false) {
    visited[p.x][p.y - 1] = true;
    q.add(new Pair(p.x, p.y - 1));
}
```
Correct solution

Figure 5: Remove a syntactically important character

3. **Replace a math operator with a random other operator**

```
for (int i = 0; i < ans.size() - 1; i++) {
    res = Math.max(res, ans.get(i + 1) / ans.get(i));
}
```
Buggy solution

```
for (int i = 0; i < ans.size() - 1; i++) {
    res = Math.max(res, ans.get(i + 1) - ans.get(i));
}
```
Correct solution

Figure 6: Replace a math operator with a random other operator

4. **Replace or modify a hardcoded int**

```
while (l < r && nums[l] == nums[l + 1]) {
    l++;
}
while (l < r && nums[r] == nums[r - 2]) {
    r--;
}
l++;
r--;
```
Correct solution

```
while (l < r && nums[l] == nums[l + 1]) {
    l++;
}
while (l < r && nums[r] == nums[r - 1]) {
    r--;
}
l++;
r--;
```
Buggy solution

Figure 7: Replace or modify a hardcoded int

5. **Replace a boolean comparison operator**

```
if (nums1[right] < max) {
    nums2[pos] = nums1[right];
    right--;
}
```
Buggy solution

```
if (nums1[right] > max) {
    nums2[pos] = nums1[right];
    right--;
}
```
Correct solution

Figure 8: Replace a boolean comparison operator

A.2 Prompt for GPT Bug Generation

Below is the prompt we used to ask GPT-3.5 Turbo to generate bugs given some code.

> Below is a java code:
> // code //
> Here is the problem that the code is solving.
> // problem //
> Given that this code have the correct implementation of the problem, your job now is to introduce one bug in this code that could cause the compile time error/ runtime error. Make sure not to indicate where you make the bug so that some one else can test their ability of debugging.
> When generating the bug, try to create bugs that are likely made by general programmers.

A.3 Prompt Design

For our prompts, which are provided below, we make a few observations. First, we use the phrase "may be buggy". This reminds the model that the code may not have a bug and to not look for bugs that do not exist, as we have negative samples. Second, we try to prevent the model from rewriting code using the phrases "using minimal changes" and "do not optimize". For CoT, we ask the model to explain reasoning, while telling the model explicitly to not explain when not using CoT. Lastly, we ask it to format the code in markdown for easier parsing.

1. **Zero-shot, without CoT (Baseline)**

> The provided Java code may be buggy. Fix the bug if one exists, using minimal changes. Do not reorganize. Do not optimize. Do not provide explanation or justification. Format your code in markdown.
>
> <Problem Description>
>
> ```java
> <CODE>
> ```

2. **Zero-shot, with CoT**

> The provided Java code may be buggy. Fix the bug if one exists, using minimal changes. Explain the reasoning process, thinking step-by-step, for identifying and fixing the bug. Do not optimize. Do not provide explanation or justification. Format your code in markdown.
>
> <Problem Description>
>
> ```java
> <CODE>
> ```

3. **Few-shot, without CoT**

> The provided Java code may be buggy. Fix the bug if one exists, using minimal changes. Do not optimize. Do not provide explanation or justification. Format your code in markdown.
>
> <Problem Description>
>
> Example #1: <CODE>
>
> Example Fix #1: <CODE>
>
> <Four other examples and fixes>
>
> buggy code:
>
> ```java
> <CODE>
> ```

4. **Fewshot, with CoT**

> The provided Java code may be buggy. Fix the bug if one exists, using minimal changes. Do not optimize. Do not provide explanation or justification. Format your code in markdown.
>
> <Problem Description>
>
> Example #1: <CODE>
>
> Example Fix #1: <CODE>
>
> Explanation for the fix
>
> <Four other examples and fixes>
>
> buggy code:
>
> ```java
> <CODE>
> ```

For `<Problem Description>`, both description of the problem context in natural language and constrains that specify input and output of the expected behaviour of the program is provided. Below is an example that demonstrates this:

> ...
> Code Description:
> The function repeatChar takes a character c and an integer times, and returns a string consisting of the character c repeated times times.
> Constraints:
> times >= 0
> c is a valid character
> ...

For `<Example>` and `<Example Fix>`, we provide a buggy code and how the bug is fixed in a correct version. One example is on the next page.

...
Code:
```java
class Solution {
  public int findMax(int[] nums) {
    int max = nums[0];
    for (int i = 1;
    i <= nums.length; i++) {
      if (nums[i] > max) {
      max = nums[i];
    }
  }
  return max;
  }
}
```
Fix:
```java
class Solution {
  public int findMax(int[] nums) {
    int max = nums[0];
    for (int i = 1;
    i < nums.length; i++) {
      if (nums[i] > max) {
      max = nums[i];
    }
  }
  return max;
  }
}
```
...

For few-shot with chain of thought, we also provide explanations for where the bugs is and why it should be fixed according to the sample fix. Specifically, our explanations answer four key questions:

- What is the bug?
- Where is it?
- Why does our code lead to a bug?
- How do we fix it?

Below is explanation for the previous example:

...
Example #1: <CODE>
Example Fix #1: <CODE>
Explanation:
The original code causes an 'ArrayIndexOutOfBoundsException' due to the loop condition 'i <= nums.length', which attempts to access an index out of the array's bounds. In Java, array indices range from 0 to 'length - 1'. The fix is changing the loop condition to 'i < nums.length', ensuring the loop iterates only within the array's valid range.
...