D-QURE: Dynamic Question based Code Refinement - A Prompt Refinement Approach

Abstract

Large Language Models (LLMs) have revolutionized various fields by significantly advancing tasks such as text generation, translation, and comprehension. Despite these successes, their application in software development tasks like debugging, code summarisation and code correction remains underexplored. While traditional approaches to code debugging and refinement integrate unit tests and program verifiers to iteratively improve the generated programs, they often fall short in handling complex logic flows and intricate data operations. Moreover advance methods often rely on static prompts or extensive model fine-tuning, which are computationally intensive and lack adaptability. In response to these limitations, we introduce **D**ynamic **QU**estion-based Code **RE**finement **D-QURE**, a novel framework leveraging LLMs to iteratively refine code through dynamic question generation. D-QURE stands out by continuously generating and refining prompts to guide the debugging process, ensuring more accurate and reliable code correction. Unlike existing methods, our framework addresses the limitations of performance saturation and declining effectiveness observed in iterative code refinement and natural language feedback-based code refinement approaches. D-QURE improves accuracy by **34.62%** (HumanEval) and **12.98%** (MBPP+) over the baseline.

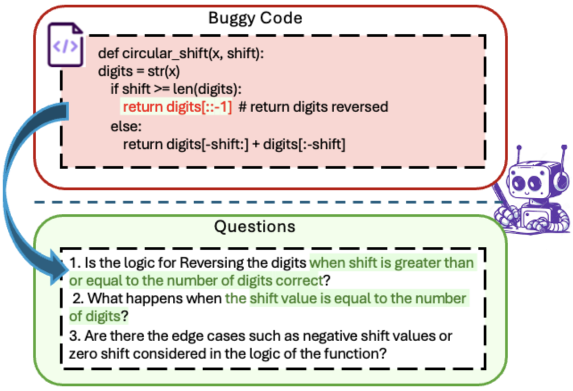

Figure 1: Example of Buggy Code and Corresponding Questions Generated by D-QURE. The generated questions pinpoint to the bug in handling shift values greater than or equal to the number of digits.

1 Introduction

The advent of pre-trained large language models has revolutionized the field of natural language processing (NLP), leading to significant advancements in tasks such as text generation, translation, and comprehension. Models such as GPT-4 (AI, March 14, 2024), Gemini (Google and Deepmind, May 14, 2024), and BERT (Jacob Devlin and Toutanova, 2018) exhibit remarkable capabilities in understanding and generating human-like text. Traditional approaches have relied on extensive model fine-tuning, which involves adjusting the model parameters for each individual task, demanding significant computational resources and storage. More recently, Brown et al. (Tianyu Gao and Chen, 2021) showed that prompt design is effective at modulating a frozen GPT-3 model's behavior through text prompts. While extensive research has been conducted on using LLMs for various NLP tasks, their application in software development tasks such as debugging, code correction, and code summarization has not received equivalent attention. This disparity is noteworthy given the complexity of modern software systems and the crucial role of efficient debugging in the software development lifecycle.

Fine-tuning (Zhang et al., 2024) large language models (LLMs) for specific tasks in software development presents several challenges like computational overhead. This approach limits scalability, making it impractical for managing diverse and dynamic codebases. Furthermore, fine-tuned models are often tailored to specific programming languages, which reduces their versatility across different coding environments. Additionally, in realworld applications, the scarcity of labeled datasets for fine-tuning poses a significant obstacle. There are few datasets with correctly annotated code snippets available for training, making it difficult to achieve robust fine-tuning. In contrast, **Prompt** (Tianyu Gao and Chen, 2021) based methods can be adapted to a variety of tasks by simply altering the input prompts, eliminating the need for multiple fine-tuned models. Prompts can be crafted to work with different programming languages, enhancing the model's versatility. This adaptability makes prompt-based methods particularly suited to the diverse and evolving needs of modern software development.

In software development, **Debugging** (Cheryl Lee and Lyu, 2024), **Code Correction** and **Code Summarisation** (Weisong Sun and Chen, 2024) are critical yet challenging tasks. Traditional methods for debugging often rely heavily on the developer's intuition and experience, which can lead to inconsistent and time-consuming outcomes. Automated debugging tools have made strides in improving efficiency, but they still fall short in handling the complexities and nuances of modern software systems. This is where leveraging the questioning approach can bring significant improvements.

Questions play a crucial role in human problem-solving, helping to clarify, focus, and direct attention to specific aspects of a problem. In the context of debugging, questions can help identify the root cause of an issue by systematically exploring various dimensions of the code. This systematic probing is akin to the way expert developers debug code by forming hypotheses and testing them through targeted inquiries. Unlike traditional automated tools, LLMs can generate contextually relevant and diverse questions that cover multiple aspects of the code, leading to a more thorough and accurate debugging process. On the other hand, relying on bug summaries or test case feedback can be less effective. Bug summaries are often verbose, can provide incorrect information, fail to highlight all mistakes, and may even hallucinate issues that do not exist. Test cases, while useful, are hard to synthesize and the use of incorrect test cases can lead to more errors at a faster pace during code refinement process. In contrast, a question-driven approach leverages the probing nature of questions to uncover deeper insights and more precise code corrections.

1.1 Dynamic Questions based Code Refinement

We introduce a novel, structured, Question-driven approach to code analysis and bug refinement, utilizing the capabilities of LLMs. **D-QURE** involves generating questions about a given piece of code, evaluating the relevance of these questions, and using the relevant ones to guide the debugging process.

Our contributions are as follows:

1. We propose a multi-agent, Question-driven debugging framework that utilizes LLMs to generate, evaluate, and filter relevant questions for code rectification.

2. We demonstrate that our approach outperforms traditional LLM-based code correction methods, yielding more accurate and reliable results.

3. We validate our methodology using established benchmarks and datasets, ensuring its robustness and practical applicability.

Through comprehensive experiments, we show that D-QURE not only enhances debugging accuracy by 34.62% for Human Eval and 12.98% for MBPP+ but also offers valuable insights into the debugging capabilities of LLMs. This work paves the way for more efficient and reliable code correction techniques, harnessing the full potential of LLMs in the field of software development and maintenance. By addressing the current research gap, we aim to foster further innovation and exploration in applying LLMs to code-related tasks.

2 Related Work

In recent years, several advancements have been made in enhancing the capabilities of LLMs through iterative refinement processes and prompt evolution mechanisms. The iterative refinement approach, as exemplified by Self Refine(Aman Madaan and Clark, 2023), focuses on improving initial outputs from LLMs through iterative feedback and refinement. (Claire Cardie and Du, 2017) introduces a data-driven approach to automatic question generation for reading comprehension using attention-based neural networks. Unlike rule-based systems, their model learns end-to-end to generate natural and challenging questions directly from text passages. The system achieves state-of-the-art performance in automatic

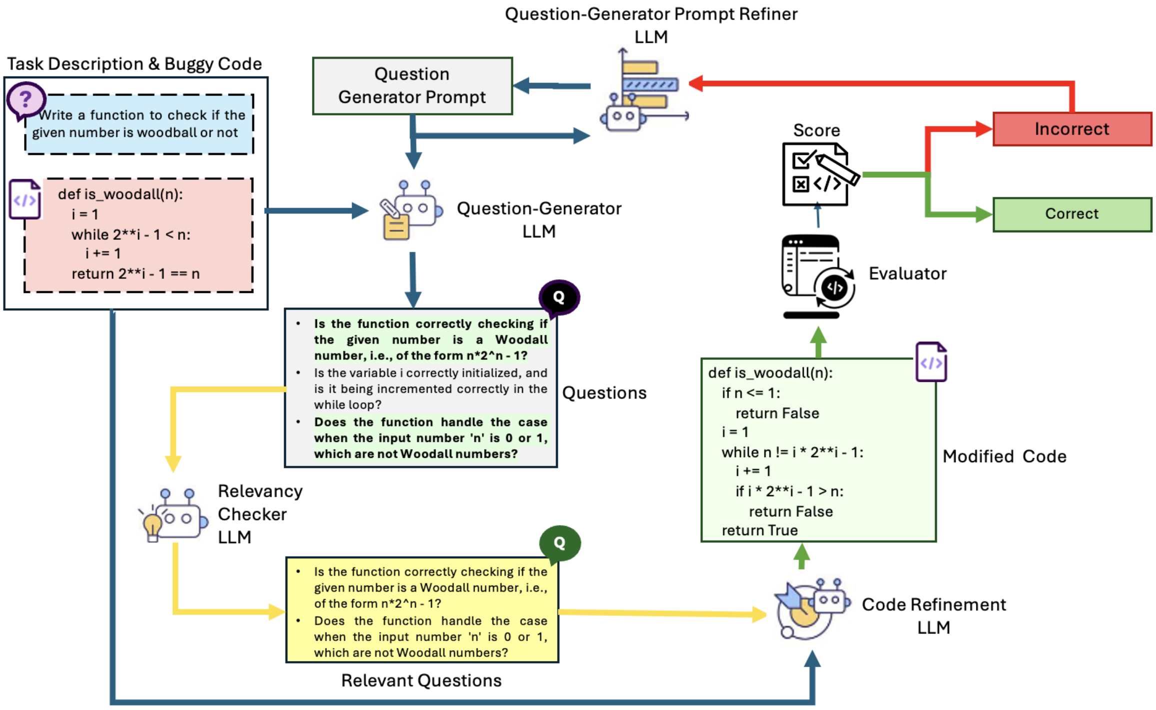

Figure 2: This figure illustrates the D-QURE Framework. The process begins with the Question Generation LLM , which generates questions based on the task description and buggy code. The Relevant Question Extractor filters these questions to identify the most relevant ones. The Rectified Code Generator then uses these relevant questions to refine the code. The Evaluator runs the refined code against the test cases to evaluate its correctness. Finally, the Prompt Refiner iterates on the process by adjusting the prompts based on the results, aiming to improve the debugging performance in subsequent cycles.

evaluations and human assessments, underscoring the importance of generating contextually relevant questions to enhance comprehension tasks.

Prompt Breeder (Fernando et al., 2023) introduces a novel self-improvement mechanism for LLMs through prompt evolution. This method iteratively evolves task-prompts and mutation-prompts using an evolutionary algorithm guided by the LLM itself. By continuously mutating and evaluating prompts, PromptBreeder surpasses traditional prompt strategies in various domains, including arithmetic and commonsense reasoning tasks. This approach highlights the efficacy of self-referential systems in adapting prompts for specific tasks without human intervention. We draw inspiration from these works to develop a novel approach for debugging code framework. Our focus lies in refining prompts iteratively to guide LLMs effectively in generating questions and modifying code. Unlike existing methods, which often rely on static prompts or manual or no feedback, our approach adapts prompts dynamically based on performance metrics and iterative feedback. By integrating these insights, we aim to fill the gap in automated debugging systems by enhancing the precision and efficiency of code correction processes through adaptive prompt refinement.

3 Methodology

3.1 Problem Formulation

In the context of code debugging, our goal is to transform a *buggy code* snippet C into a corrected version that meets the requirements outlined in the task description T.

Formally, we can describe this process as Algorithm 1.

Algorithm 1: D-QURE Code Refinement Process
Input: Task description T, Buggy code C;
Output: Corrected code C^*

1: **Generate Questions:** $Q_g = QG(T, C)$
2: **Filter Relevant Questions:** $R = S(Q_g, T, C)$
3: **Correct Code:** $Q_g^r = \{q \in Q_g \mid R(q, T, C) = 1\}$ $C^* = CR(Q_g^r, C, T)$

---

where Q_g is the set of generated questions, R is

the relevance score, Q_g^r is the subset of questions with relevance score = 1, and CR is the function that refines C based on Q_g^r to produce C^*.
The objective is to enhance the efficiency of code debugging, validated against test cases T_h for robustness.

3.2 Overview

Our debugging workflow 2 begins with identifying the task description and buggy code. A Question-Generator LLM creates code-specific questions, which are filtered by a Relevancy Checker to retain only the most useful ones. These guide the Code Refinement Agent in generating a corrected version of the code. The modified code is then tested against predefined test cases. If it fails, the process is iteratively refined until the code passes or a set iteration limit is reached.

The following sections detail each component of this workflow:

3.2.1 Dynamic Question Generator

The Question Generation block is integral to our debugging framework. The process begins with the identification of a *Task description* and the associated *Buggy Code*. This information is formulated into a Prompt [**??**], which is used to generate detailed and relevant questions about the code. Given the task description T and the corresponding buggy code C, our framework uses the pretrained Language Learning Model (LLM) to generate a set of questions. The goal of these questions is to uncover potential logical errors, incorrect implementations, or edge cases that the code may not handle properly.

Mathematically, this process can be represented as follows:

$$QG(C,T) \to \{Q_1, Q_2, \ldots, Q_n\}$$

where QG is the Dynamic Question Generator and $\{Q_1, Q_2, \ldots, Q_n\}$ is the set of generated questions. Each Q_i is crafted to probe specific aspects of the code's functionality and logic.

For instance, if the task is to *check whether a number is a Woodall number*, the generated questions are:

1. Is the function correctly checking if the given number is a Woodall number, i.e., of the form $n \cdot 2^n - 1$?

2. Does the function handle the case when the input number n is 0 or 1, which are not Woodall numbers?

3. Is the variable i correctly initialized, and is it being incremented correctly within the while loop?

These questions aim to dissect the code logic, identify potential edge cases, and ensure that all variables and control structures are correctly implemented. By generating a diverse set of questions, the framework can effectively pinpoint areas where the code may be failing or where improvements can be made.

In essence, the Question Generation block leverages the capabilities of LLMs to provide a structured and systematic way of interrogating the code, laying the groundwork for subsequent stages.

3.2.2 Question Relevance Evaluation

The Relevant Question Evaluation is critical for ensuring the efficiency and effectiveness of the debugging process. Not all questions generated by the Question-Generator are equally useful. Filtering out irrelevant questions is essential because including unnecessary questions can clutter the context, making it harder for the subsequent stages to rectify the bugs. This filtering helps maintain the clarity and relevance of the information provided to the Code Refinement block.
Mathematically, the process of evaluating and selecting relevant questions can be represented as follows:

$$\{Q_1, Q_2, \ldots, Q_n\} \to \{Q'_1, Q'_2, \ldots, Q'_m\}$$

where $\{Q_1, Q_2, \ldots, Q_n\}$ is the initial set of questions, and $\{Q'_1, Q'_2, \ldots, Q'_m\}$ is the filtered set of relevant questions, with $m \leq n$.
Each question is analysed based on its context, specificity, and relevance to the *task description* and the *buggy code*. The goal is to select questions that are most likely to lead to insights about the underlying issues in the code, thereby streamlining the debugging process and avoiding unnecessary computational overhead.

3.2.3 Code Refinement

The Code Refinement CR stage uses the *relevant questions*, the *buggy code*, and the *task description* to generate a modified version of the code.

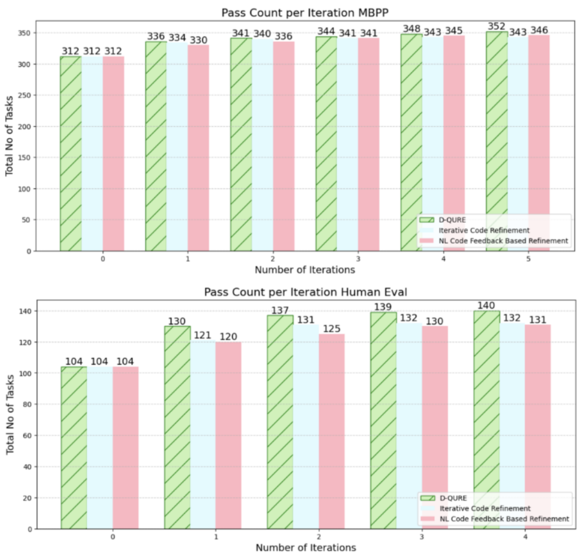

Figure 3: Comparison of Pass Rates Across Iterations for NL Code Feedback, Iterative Refinement, and D-QURE. This bar plot illustrates the number of test cases passed by each model at different iterations, highlighting the effectiveness of D-QURE in improving code correctness over multiple iterations.

The main idea is to leverage the additional context provided by the questions to guide the LLM in understanding where modifications are needed and how to apply them effectively.
Mathematically, the process can be represented as follows:

$$C^* = CR(C, T, \{Q'_1, Q'_2, \ldots, Q'_m\})$$

where:
- C is the buggy code and T is the task description.
- $\{Q'_1, Q'_2, \ldots, Q'_m\}$ are the relevant questions after Relevance Evaluation.
- C^* is the modified code output by the LLM.

3.2.4 Evaluator

The evaluation stage is crucial for validating the effectiveness of the code refinement process. This stage assesses whether the modifications introduced during refinement have successfully addressed the bugs or shortcomings in the original code. We evaluate the refined code on two key metrics for each pass, denoted as pass@k, where k represents the k-th iteration of the refinement process. For example, pass@1 accuracy and pass@1 PI score are calculated for the first iteration. Pass@k indicates the evaluation of the code's performance after the k-th iteration. We use these metrics to assess the cumulative performance of the refined code:

- Accuracy : This metric measures the binary outcome of all the test cases cumulatively.

$$Accuracy = \frac{\#\ Buggy\ code\ fixed}{Total\ Tasks\ count}$$

- **P**artial **I**mprovement (PI) Score: This metric provides a more detailed assessment of the refinement process beyond a simple pass/fail outcome. It considers the severity or importance of each test case.
The scoring mechanism can be mathematically represented as follows:

$$PI\ Score = \frac{\#\ Test\ cases\ passed}{Total\ test\ cases}$$

We leverage the **EvalPlus** (Liu et al., 2023) framework for comprehensive testing and scoring. This framework provides a structured environment for running the refined code against a suite of test cases designed to assess its correctness and reliability. By analyzing these metrics, we can identify any residual issues remaining in the refined code. This information can then be used to guide further refinements and ultimately enhance the overall code quality.

3.2.5 Prompt Refinement

Prompt refinement aims to iteratively enhance the prompts provided to the LLM for generating precise and contextually relevant questions and code. The prompt refinement process consists of the following steps:

1. Initial Prompt Creation: We already begin by generating k distinct prompt templates based on the *task description* and the identified *buggy code*. These templates serve as the initial set of instructions for generating questions.

2. Scoring and Evaluation: Each of the k prompt templates is employed to generate questions, which are subsequently used to guide the code refinement process. The effectiveness of each prompt template is quantified by the PI score of the refined code across all the test cases.

3. Identifying Prompts for Refinement: To determine which prompt templates require refinement, we consider two criteria:

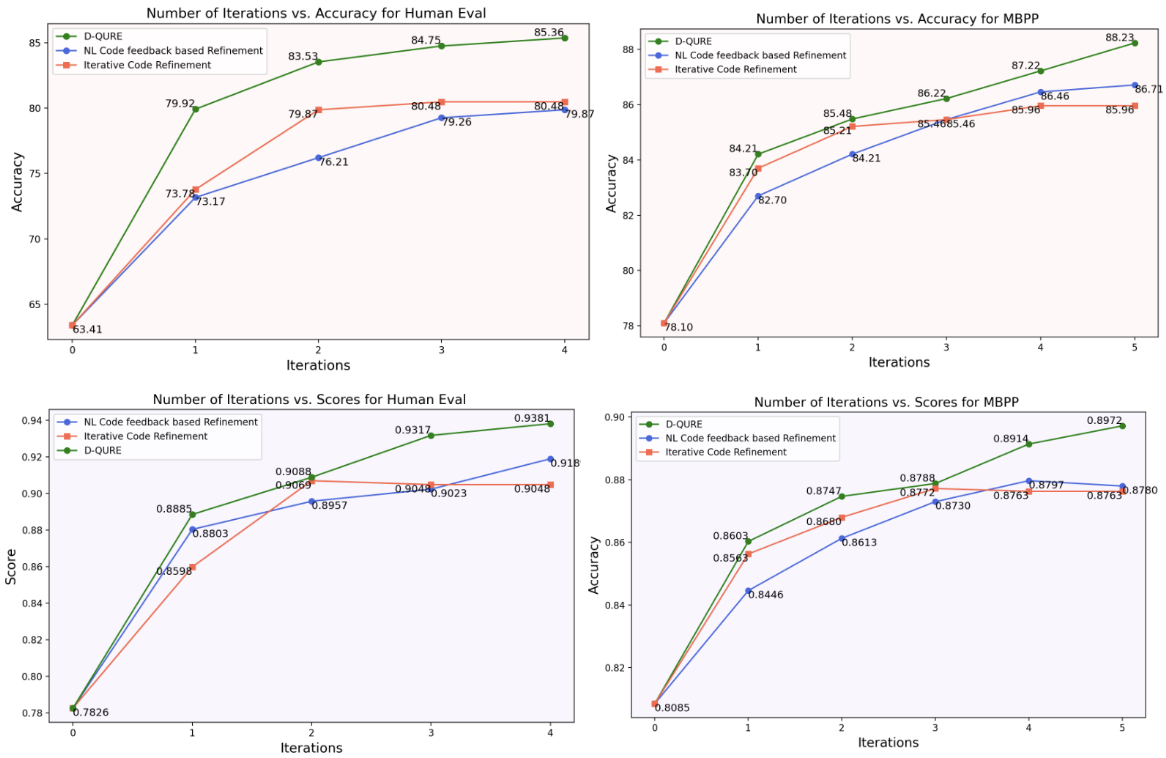

Figure 4: Performance Comparison of Natural Language Code Feedback based Refinement, Iterative Code Refinement, and D-QURE on MBPP+ and Human Eval datasets: (a) Scores plot highlighting the superior performance of D-QURE in generating high-quality, accurate code solutions. (b) Accuracy Comparison demonstrating that D-QURE consistently outperforms the other models in generating precise code corrections over multiple iterations.

(a) Minimum Edit Distance and Maximum Score Difference: We select a pair of prompt templates based on the minimum edit distance between them and the maximum difference in their scores. This approach identifies pairs of prompts that are structurally similar but exhibit significant performance differences.

(b) Max and Min Score Prompts: Select a pair of prompts with the highest and lowest scores. However, empirical evidence indicates that the first approach (minimum edit distance and maximum score difference) yields better results.

4. Dynamic Question Generator Prompt Refinement : Upon identifying the prompt pair, the refinement process begins with the analysis of the high-scoring prompt template to identify elements that contribute to its success. These elements are considered beneficial and are prioritized in the refined prompt and exclusion of low-scoring elements that is elements from the low-scoring prompt template identified as detrimental to performance are excluded from the refined prompt.

Based on this analysis, a new set of refined question generator prompts is generated. These prompts incorporate the advantageous elements from the high-scoring template while excluding the detrimental elements from the low-scoring template. The process of scoring, evaluation, and refinement is repeated iteratively. Each iteration aims to produce more precise and contextually aware questions, leading to more effective correct code generation. This iterative process continues until one of the following conditions is met: A cycle length of 5 iterations is achieved or all test cases are passed by the refined code or there is no increase in the score for that cycle.

4 Experiments and Results

4.1 Experiment Setup

In our experimental setup, we utilized the Human Eval (Mark Chen and Bavari 2021) and MBPP+ (Jacob Austin and Le 2021) datasets to evaluate

Methods	MBPP+(%)	Human Eval(%)
First Pass Code **Accuracy**	78.10	63.41
Iterative Code Refinement	85.96 (↑10.06)	80.48 (↑26.92)
NL feedback based Code Refinement	86.71 (↑11.02)	79.87(↑25.95)
D-QURE(ours)	**88.23 (↑12.98)**	**85.36(↑34.62)**

Methods	MBPP+(%)	Human Eval(%)
First Pass Code **PI Score**	80.85	78.26
Iterative Code Refinement	87.63 (↑8.38)	90.48 (↑15.61)
NL feedback based Code Refinement	87.80 (↑8.59)	91.89(↑17.41)
D-QURE(ours)	**89.72 (↑10.98)**	**93.81(↑19.87)**

Table 1: Comparison of code refinement accuracy (left) and Partial Improvement (PI) score (right) across different methods on MBPP+ and HumanEval datasets. The table displays the initial accuracy of the seed code and the improved accuracy for Iterative Code Refinement, NL Feedback Based Code Refinement, and D-QURE. The percentage increase in accuracy is shown in parentheses. D-QURE demonstrates the highest improvement in accuracy on both datasets, with a significant increase of 12.98% on MBPP+ and 34.62% on HumanEval.

the effectiveness of our debugging framework. After generating seed programs using Llama3 (AI April 18, 2024). We proceeded with the following dataset-specific evaluations: the HumanEval dataset consists of 164 Python instances (104 correct), while the MBPP+ dataset includes 399 instances (312 correct). Detailed initial prompts for seed program generation are provided in Appendix. To balance computational feasibility and the potential for meaningful improvements, we set the maximum number of debugging iterations to five. The refined prompts, which are also detailed in the Appendix, were generated based on an iterative refinement process designed to enhance the debugging framework. Additional implementation details, including exact prompts, generation parameters, and iterative refinement processes, are provided in the Appendix.

4.1.1 Performance Metrics

In this study, we employ several key metrics to evaluate the effectiveness of our models. The primary metrics includes the accuracy for each pass, the score for each pass, and the count of task IDs that pass after each iteration. Partial Improvement(**PI**) Score is calculated as the ratio of correctly passed test cases to the total number of test cases.

4.2 Methods

4.2.1 D-QURE: Dynamic Question-Based Buggy Code Refinement

D-QURE initiates with five different prompt templates. For each pass, the process involves generating questions, evaluating their relevance, and refining the *buggy code* using the *relevant questions* and *task description*. The scores for each template are calculated, and the best-performing template's accuracy and score are recorded as the pass@1 results. The Prompt Refiner refines the question generator prompt to create five new templates for the next pass.

4.2.2 Natural Language Bug Feedback Based Code Refinement

This approach involves consist of two phases. Initially, a natural language bug summary of the code is generated. This summary is then provided to the next stage, where the *buggy code* is rectified based on the identified *bug summary*. The performance is evaluated after each iteration, and the accuracy for each iteration is reported.

4.2.3 Iterative Code Refinement

In the iterative code refinement approach, each *task* is paired with its corresponding *buggy code*, which is then passed to the LLM to generate corrected code. If the model fails to produce a correct solution in the first pass, the buggy code along with the task is sent to the next pass for further refinement. This process continues iteratively until the code passes the test or all passes are completed. After each pass, the performance is evaluated, and the accuracy for that pass is reported.

4.3 Results

We compare D-QURE with two baseline models: Natural Language Bug Feedback Based Code Refinement and Iterative Code Refinement, evaluating their performance on the MBPP+ and HumanEval datasets. D-QURE consistently outperforms both models.

In terms of accuracy (Table 1), D-QURE maintains higher pass rates across all passes, achieving pass@5 accuracy of **88.23%** for MBPP+ and pass@4 accuracy of **85.36%** for HumanEval. This demonstrates a significant improvement over the other two methods. Figure 4 illustrating the PI score plot reveals that D-QURE achieves superior scores in each iteration, reflecting its robust code refinement capabilities. As shown in Figure 3, the pass rates for each iteration further underscore D-

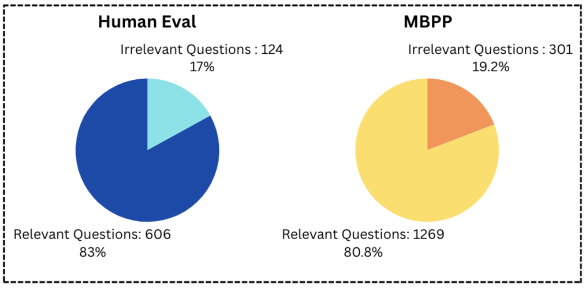

Figure 5: Distribution of questions labeled as 'Relevant' (approved by the relevancy checker LLM for code refinement) versus 'Non-Relevant' (filtered out as irrelevant).

QURE's efficacy, as it achieves consistently higher pass counts compared to the baseline models. A detailed examination shows that D-QURE maintains high accuracy and score levels throughout the iterations. In contrast, the Iterative Code Refinement model, while initially showing improvement, reaches a saturation point where further iterations fail to yield significant gains. The Natural Language Bug Feedback Based Code Refinement model also shows a similar trend, with initial improvements followed by a plateau and eventual decline in scores. This highlights the limitation of these methods in sustaining long-term improvements. D-QURE, however, demonstrates a continuous upward trajectory, indicating its robustness and effectiveness in iterative bug fixing and code refinement. D-QURE achieves an increase in accuracy by **34.62%** on HumanEval and **12.98%** on MBPP+ compared to the first pass seed code generation accuracy, reflecting a **1.35x** improvement on HumanEval and a **1.13x** improvement on MBPP+. These results validate the effectiveness of D-QURE in refining buggy code and highlight its potential as a superior tool for automated code refinement and bug fixing. It states that our approach is more effective than conventional methods, as evidenced by the consistently higher performance metrics across both datasets.

5 Conclusion and Future Work

5.1 Conclusion

In this work, we introduced **D**ynamic **Qu**estion-based Code **Re**finement (**D-QURE**), a novel framework designed to enhance the debugging capabilities of Large Language Models (LLMs). It systematically generates and evaluates contextually relevant questions to guide the code refinement process, addressing the limitations of traditional debugging methods that often rely on static prompts or extensive model fine-tuning.
Our experiments on the MBPP+ and Human Eval dataset revealed several key findings:

1. D-QURE consistently outperforms baseline models in terms of accuracy, performance improvement score, and pass rates across multiple iterations.

2. While traditional methods tend to plateau or degrade in performance over time, D-QURE maintains its effectiveness, demonstrating robust iterative improvement.

3. The question-driven approach enables a more nuanced and targeted debugging process, particularly effective in handling complex logic flows and intricate data operations.

These results underscore the potential of D-QURE to revolutionize automated code correction, setting a new benchmark for efficiency and adaptability in software development practices. Hopefully these findings will contribute to the advancement of large language models in the field of automatic debugging.

5.2 Limitations

The framework's performance heavily depends on the quality of test cases and initial prompts. Poor test cases or prompts can lead to inaccurate evaluations and weaker refinements. Additionally, while the question generator prompt undergoes refinement, other prompts in the framework are not subject to the same level of improvement, which may impact the overall performance.

5.3 Future Work

Future improvements can focus on enhancing the evaluator by enabling it to dynamically generate test cases rather than relying solely on predefined ones. Integrating reinforcement learning techniques for prompt refinement could further optimize the model's performance by continuously improving prompt quality. Additionally, incorporating runtime execution feedback would provide real-time insights, facilitating more precise code modifications. These advancements aim to make the framework more robust and effective in automated debugging and code refinement.

References

Open AI. March 14, 2024. ChatGPT-4. https://openai.com/index/chatgpt/. Accessed on: July 18, 2024.

Katherine Hermann-Sean Welleck Amir Yazdanbakhsh Aman Madaan, Niket Tandon and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. *NeurIPS*.

Jen-tse Huang Zhouruixin Zhu Lingming Zhang Cheryl Lee, Chunqiu Steven Xia and Michael R. Lyu. 2024. A unified debugging approach via llm-based multi-agent synergy. *arXiv:2404.17153*.

Junru Shao Claire Cardie and Xinya Du. 2017. Learning to ask: Neural question generation for reading comprehension. *arXiv:1705.00106*.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktaschel. 2023. Promptbreeder: Self-referential self-improvement prompt evolution. *arXiv:2309.16797*.

Google and Deepmind. May 14, 2024. Gemini. https://deepmind.google/technologies/gemini/. Accessed on: July 18, 2024.

Kenton Lee Jacob Devlin, Ming-Wei Chang and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv:1810.04805*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, Zhang, and Lingming. 2023. EvalPlus. https://github.com/evalplus/evalplus. Accessed on: July 11, 2024.

Adam Fisch Tianyu Gao and Danqi Chen. 2021. Making pre-trained language models better few-shot learners. *arXiv:2107.03374v2*.

Yuekang Li Hongyu Zhang Chunrong Fang Yi Liu Gelei Deng Yang Liu Weisong Sun, Yun Miao and Zhenyu Chen. 2024. Source code summarization in the era of large language models. *arXiv:2407.07959v1*.

Quanjun Zhang, Chunrong Fang, Weisong Sun, Yan Liu, Tieke He, Xiaodong Hao, and Zhenyu Chen. 2024. Appt: Boosting automated patch correctness prediction via fine-tuning pre-trained models. *IEEE Transactions on Software Engineering*, 50(3):474–494.

Appendix: Prompt used in D-QURE

This section provides a detailed description of the various prompts used in our workflow, outlining their purpose and usage.

Question Generation Prompt

This prompt is utilized to create a series of questions based on the task description and the initial buggy code. The main goal is to pinpoint areas of the code that may be problematic or require further investigation. By generating these questions, the model can identify potential issues and focus on specific parts of the code that need attention, enhancing the overall debugging process. The template for this prompt can be found in Figure 6, which illustrates the structure used for generating these questions.

Question Relevancy Prompt

Once a series of questions has been generated, this prompt is employed to assess their relevance. It filters out questions that are least likely to assist in identifying and correcting bugs, thus streamlining the debugging process. This filtering step ensures that only the most pertinent questions are considered, reducing extraneous context and improving the efficiency of subsequent code modifications. The template for this prompt can be found in Figure 7, which illustrates the structure used for generating the relvancy score for each of the questions.

Bug Rectification Prompt

This prompt is designed to address and correct the identified issues in the buggy code. By incorporating the relevant questions generated previously and the task description, the LLM is guided to make precise and effective modifications. This prompt helps ensure that the corrections made to the code are directly informed by the questions and context provided. The template for this prompt can be found in Figure 8, which illustrates the structure used for generating the rectified code.

Mutation Prompt

In this stage, the prompts are refined iteratively to improve their quality and relevance. The refinement process involves comparing different prompts based on criteria such as minimum edit distance and maximum score difference. By evolving the prompts over multiple iterations, this step aims to enhance the overall effectiveness of the debugging process and improve the accuracy of the generated code. The template for this prompt can be found in Figure 9, which illustrates the structure of the prompt.

Iterative Bug Rectification Prompt

This prompt serves as a reference point for evaluating the progress of the prompt refinement process across multiple iterations. It establishes a baseline against which improvements in code quality and debugging accuracy can be measured. Maintaining this baseline helps in tracking the effectiveness of the iterative refinement process and ensures that the enhancements are quantifiable.

Natural Language Feedback generation Prompt

The Summary Creation Prompt is used to generate a comprehensive overview of the entire debugging process. It includes details such as the initial problem, identified bugs, questions generated, and the final corrected code. This summary provides a detailed account of the iterative refinement process, helping to consolidate the information and present a clear narrative of the debugging workflow.

Natural Language Feedback based Bug rectification Prompt

Using the insights gained from the summary of the debugging process, this prompt further refines the code. By leveraging the summary's information, the LLM is enabled to make more informed and accurate modifications. This approach aims to increase the likelihood of passing all test cases by utilizing the comprehensive understanding derived from the summary to guide code corrections.The template for this prompt can be found in Figure 10, which illustrates the structure used for generating the rectified code based on the Natural Language feedback for the given task and the buggy code.

Role: You are a meticulous code reviewer.

Task: Evaluate the buggy code and devise 5 specific questions to identify the bug.

Input buggy code:

```
def long_words(n,, sentence):

"""Write a function to find words that are longer than n characters from a given list of words.

assert long_words(3,"python is a programming language")==['python','programming','language']"""

    words = sentence.split()

    return [word for word in words if len(word) > n]
```

Pseudocode Creation: Break down the steps in pseudocode.

Questions: Develop 5 questions from your pseudocode targeting key code segments, unexpected behaviors, and variable states. Analyze dependencies and inconsistencies. Rank questions by their effectiveness in detecting the bug.

Write the questions enclosed in <Question1></Question1> tags, the second question between <Question2></Question2> and so on. Don't write anything else in your response.

Model Output:

Question1: Does the function long_words(n, sentence) correctly handle multiple sentences in the input sentence?

Question2: Is the absence of the parameter name following 'n' in the function definition intentional?

Question3: Can the function long_words(n, sentence) deal with punctuation marks attached to the words in the sentence?

Question4: Are there any edge cases (e.g., empty sentence, n equal to zero, etc.) that the function does not handle?

Question5: Would the function long_words(n, sentence) behave as expected if the input sentence contains special characters like newline, tab, etc.?

Figure 6: Question Generation Prompt: One of the prompt templates used to generate a series of questions based on the task description and the initial buggy code. The purpose is to identify areas of the code that might be problematic and need further investigation.

Observe the buggy code implementation of the following TASK: Write a function to find words that are longer than n characters from a given list of words.
assert long_words(3,"python is a programming language")==['python','programming','language']

The Buggy Code :

```
def long_words(n,, sentence):

   words = sentence.split()

   return [word for word in words if len(word) > n]
```

The bug can be either be Logical error or Syntax error. Evaluate the following questions to determine if they are relevant and helpful in identifying the bug type (Logical Error or Syntax Error) in the provided code.

The questions are as follows:

Question1: Does the function long_words(n, sentence) correctly handle multiple sentences in the input sentence?
Question2: Is the absence of the parameter name following 'n' in the function definition intentional?
Question3: Can the function long_words(n, sentence) deal with punctuation marks attached to the words in the sentence?
Question4: Are there any edge cases (e.g., empty sentence, n equal to zero, etc.) that the function does not handle?
Question5: Would the function long_words(n, sentence) behave as expected if the input sentence contains special characters like newline, tab, etc?

Assign Score 1: if the question is relevant and helpful or Score 0 if the question is not relevant or not helpful. \

Model Output:

Question1: Does the function long_words(n, sentence) correctly handle multiple sentences in the input sentence? Score: 0

Question2: Is the absence of the parameter name following 'n' in the function definition intentional? Score: 1

Question3: Can the function long_words(n, sentence) deal with punctuation marks attached to the words in the sentence? Score: 1

Question4: Are there any edge cases (e.g., empty sentence, n equal to zero, etc.) that the function does not handle? Score: 1

Question5: Would the function long_words(n, sentence) behave as expected if the input sentence contains special characters like newline, tab, etc? Score: 0

Figure 7: Question Relevancy Prompt: This prompt filters the generated questions to determine their relevance. Only the questions that are most likely to help identify and correct bugs are kept, reducing unnecessary context and improving the efficiency of subsequent code modifications.

Observe the buggy code implementation of the following

TASK: """Write a function to find words that are longer than n characters from a given list of words.
assert long_words(3,"python is a programming language")==['python','programming','language']

The buggy solution with a bug guaranteed is:

```
def long_words(n,, sentence):
    words = sentence.split()
    return [word for word in words if len(word) > n]
```

These are the few questions which will be helpful in identifying the bug in the code:

Question1: Is the absence of the parameter name following 'n' in the function definition intentional?
Question2: Can the function long_words(n, sentence) deal with punctuation marks attached to the words in the sentence?
Question3: Are there any edge cases (e.g., empty sentence, n equal to zero, etc.) that the function does not handle?

Use the provided information to identify the bug and then generate the rectified code with the bug resolved. Your task is to correct the buggy code and give the correct code enclosed in <correct_code></correct_code> tags.

Model Output:

```
def long_words(n, sentence):
    words = [''.join(e for e in word if e.isalpha()) for word in sentence.split()]
    return [word for word in words if len(word) > n]
```

Figure 8: Bug Rectification Prompt: This prompt is used to correct the identified buggy code. By incorporating relevant questions and the task description, the LLM is guided to make precise modifications to the code.

```
I am giving these two prompts for a language model designed for the task of
question generation.

### QUESTION GENERATION PROMPT1 ###
**Role**: You are a thorough code reviewer.
**Task**: Inspect the buggy code and craft 5 precise questions to find the bug.
**Input buggy code**: BUGGY CODE.
**Pseudocode Creation**: Illustrate the steps in pseudocode.
**Questions**: Formulate 5 questions from your pseudocode aimed at specific
code sections, unexpected behaviors, variable states, and dependencies. Rank
questions according to their effectiveness in detecting the bug.

### QUESTION GENERATION PROMPT2 ###
**Role**: You are a meticulous code reviewer.
**Task**: Evaluate the buggy code and devise 5 specific questions to identify the
bug.
**Input buggy code**: BUGGY CODE.
**Pseudocode Creation**: Break down the steps in pseudocode.
**Questions**: Develop 5 questions from your pseudocode targeting key code
segments, unexpected behaviors, and variable states. Analyze dependencies and
inconsistencies. Rank questions by their effectiveness in detecting the bug.

### TASK ###
{}

### BUGGY CODE ###
{}

Accuracy score is defined as the Average of (number of passed test cases )/(Total
test cases). Accuracy scores for PROMPT1 is 89.14% and PROMPT2 is 88.14%.
Refine the QUESTION GENERATION PROMPT using the following logic:
1. Identify Prompts' Performance: Compare the accuracy scores of existing
prompts.
- Higher score indicates a more effective prompt.
- Lower score indicates a less effective prompt.
2. Analyze Effective Prompts: Focus on the structure and content of prompts with
higher scores to understand what makes them effective.
3. Identify Improvement Areas: Examine prompts with lower scores to pinpoint
weaknesses or gaps.
4. Combine Strengths and Address Weaknesses: Create new prompts by
incorporating successful elements from high-scoring prompts and addressing
issues found in low-scoring prompts.

Don't write anything else in your response except the list of redefined Question
Generation prompt to improve the score.

### 5 REFINED QUESTION GENERATION PROMPT LIST ###
```

Figure 9: Mutation Prompt: Prompts are refined iteratively to enhance the quality and relevance of the generated code. By comparing prompts based on the minimum edit distance and maximum score difference, the Prompt generates set of 5 new question generation prompts.

For the given

Task: <text>

with its Buggy code implementation: <code>.

Based on the given Bug summary: <bug_summary_text>. Generate the rectified code enclosed within <correct_code></correct_code> tags.

Don't write anything else in your response besides code.

Figure 10: Bug Summary Based Rectification: This prompt generated the summary of the buggy code to further refine the code. By leveraging the insights gained from the summary, the LLM can make more informed modifications to the code, ensuring a higher likelihood of passing all test cases.