RAGFix: Enhancing LLM Code Repair Using RAG and Stack Overflow Posts

Elijah Mansur*, Johnson Chen†, Muhammad Anas Raza‡, Mohammad Wardat‡
*Ohio State University, Columbus, USA, mansur.17@osu.edu
†Rochester Adams High School, Rochester Hills, USA, johnsonchen2026@gmail.com
‡Oakland University, Rochester Hills, USA, {mraza, wardat}@oakland.edu

2024 IEEE International Conference on Big Data (BigData) | 979-8-3503-6248-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/BigData62323.2024.10825785

***Abstract*—Identifying, localizing, and resolving bugs in software engineering is challenging and costly. Approaches to resolve software bugs range from Large Language Model (LLM) code analysis and repair, and automated code repair technology that aims to alleviate the technical burden of difficult to solve bugs. We propose RAGFix, which enhances LLM's capabilities for bug localization and code repair using Retrieval Augmented Generation (RAG) based on dynamically collected Stack Overflow posts. These posts are searchable via a Question and Answer Knowledge Graph (KGQA). We evaluate our method on the HumanEvalFix benchmark for Python using relevant closed and open-source models. Our approach facilitates error resolution in Python coding problems by creating a searchable, embedded knowledge graph representation of bug and solution information from Stack Overflow, interlinking bugs, and solutions through semi-supervised graph construction methods. We use cosine similarity on embeddings based on LLM-synthesized summaries and algorithmic features describing the coding problem and potential solution to find relevant results that improve LLM in-context performance. Our results indicate that our system enhances small open-source models' ability to effectively repair code, particularly where these models have less parametric knowledge about relevant coding problems and can leverage non-parametric knowledge to provide accurate, actionable fixes.**

***Index Terms*—Retrieval-augmented generation, Large Language Models, knowledge graph, Bug detection, Code Repair**

I. Introduction

Detecting, pinpointing, and fixing software bugs is both a challenging and expensive aspect of software engineering [1]. Various strategies are employed to address these issues, including the use of LLMs for code analysis and repair, as well as automated code repair technologies [2]. These advanced methods aim to reduce the technical burden associated with resolving complex bugs, yet often struggle to localize and repair bugs in code. Various technical stacks from low to high-level programming paradigms offer unique challenges for LLMs to solve coding problems for legacy and modern code infrastructure [3]. Software development increasingly grapples with the complexity of debugging, demanding advanced tools for effective resolution.

Recent advancements in natural language processing (NLP) have significantly improved the capabilities of language models. These advancements have paved the way for more sophisticated tools capable of understanding and generating code. However, despite these innovations, the process of bug localization and fixing remains labor-intensive and challenging due to the increasing complexity of software systems [4], [5]. This study addresses these challenges by leveraging RAG and KGQAs to enhance LLMs in handling complex bugs. Our approach involves utilizing small to medium-sized LLMs, such as LLAMA 3 models [6] with 8 billion and 70 billion parameters, to extract and classify features from code, creating a searchable dataset, embedded knowledge graph representation of bug and solution information from Stack Overflow. By conducting similarity searches on these extracted features, we can link relevant solutions to identified bugs, improving the LLMs' ability to provide accurate and actionable fixes. Integrating advanced search techniques and optimizing the use of external knowledge sources, this research aims to reduce debugging time and effort, thus advancing software engineering through more efficient and precise code repair solutions. In this work, we make the following contributions:

- We propose a Stack Overflow vector database, leveraging advanced search techniques to enhance LLM-based code repair through RAG.
- We apply a key feature extraction prompting technique to improve the ability of LLMs to search natural language APIs in a more human-like manner, enhancing their capability to find relevant Stack Overflow information.
- We identify generalizable features of coding bugs, such as algorithms and code summaries, to improve similarity search and bug resolution.
- We introduce a technique called Iterative Feature Extraction, Search, and Database Expansion, which enables the continuous integration of Stack Overflow data into a vector database as users encounter new bugs without sufficiently relevant results in the existing database.
- We present results from applying our technique to smaller LLMs, including the LLAMA3 8B and 70B parameter models, demonstrating that our solution improves LLMs' reasoning abilities in code repair, leading to more effective bug fixes.

The rest of the paper is structured as follows: Section §II explains the literature review, Section §III presents the methodology, Section §IV discusses the evaluation, Section §IV describes the limitations, and Section §VI provides the conclusion and future work.

II. Related Work

Software development faces significant challenges in debugging due to increasing system complexity, necessitating advanced tools for effective bug resolution. Key advancements in NLP, such as the Transformer architecture introduced in "Attention is All You Need" [7], BERT's bidirectional context understanding [8], and specialized models like CodeBERT for programming languages [9], have paved the way for improved tools. Word embeddings, as presented in the Word2Vec paper [10], also enhanced NLP task performance by capturing semantic relationships. Despite these advancements, bug localization and fixing remain labor-intensive. RAG enhances LLM capabilities by utilizing Question and Answer Knowledge Graphs (KGQAs) to retrieve relevant Stack Overflow posts, thereby improving LLMs' ability to handle complex bugs [11]. This study aims to enhance LLM-based code repair by leveraging advanced search techniques and key feature extraction to a Stack Overflow vector database, ultimately improving LLMs' ability to identify and resolve coding bugs. By optimizing these techniques, this research seeks to reduce debugging time and effort, accelerating software development and paving the way for advanced applications in software engineering.

Our proposed system enhances LLMs' ability to identify and repair code bugs through a multi-step approach. Users start by providing a docstring and buggy code, which allows the system to pinpoint potential algorithms and retrieve relevant solutions from sources like Stack Overflow. This process builds on advancements in retrieval-augmented generation and the use of external knowledge sources, as outlined in recent studies [11] [12]. To evaluate the effectiveness of our approach, we use the HumanEvalFix benchmark [13] for Python, comparing LLMs' initial performance with their performance using our system. Our method utilizes LLMs, including LLAMA 3, to classify and extract features from user prompts and buggy code. This approach benefits from the success of models like CodeBERT, which excels in programming language understanding [9], and the embedding techniques introduced by Word2Vec [10].

The system creates a knowledge graph through similarity searches on extracted features, linking relevant solutions to identified bugs. This technique aligns with recent research emphasizing the role of contextual knowledge in bug localization and repair [14] [15] [2]. Initially, similarity searches on extracted features identify related bugs, which are then mapped onto a bug graph and refined into a solution graph linking relevant solutions to the bugs. Our system enhances LLMs' code repair capabilities by creating an embedded knowledge graph for more relevant and effective solutions using semi-supervised graph construction methods [16] [17]. Vector databases capture the input's semantics, enabling efficient similarity searches. We employ cosine similarity on embeddings derived from LLM-synthesized summaries and algorithmic features to locate the most pertinent results, thereby improving LLMs' code repair capabilities and providing accurate results to users. This method aligns with recent advancements in retrieval-augmented generation [11], nearest neighbor knowledge graph embeddings [12], and integrating graph neural networks with retrieval techniques [18] [19]. Moreover, unifying LLMs with knowledge graphs [20] and handling multi-hop question answering [21] further enhance the system's effectiveness in debugging tasks.

Prompt 1

As an expert in information distillation and Python programming, your task is to interpret and understand a Python coding problem with buggy code.
Each coding problem can be broken down into a series of distinct sub-algorithms. Each of these sub-algorithms addresses a specific part of the problem and contributes to the overall solution. It is crucial that each sub-algorithm is unique and does not overlap conceptually with others. Your task is to identify these individual sub-algorithms and describe them in a way that they can be easily searched and understood.
Guidelines:

1) Extract essential information to solve the given problem.
2) Provide a concise, generalized description of each algorithm, not the implementation.
3) Ensure each sub-algorithm addresses a different aspect of the problem.
4) Provide a concise description of each sub-algorithm, under 12 words.
5) Write the descriptions as if searching for their implementation on Google.
6) Each algorithm must be general enough for use in similar problems with different contexts.
7) Include up to 3 novel sub-algorithms per function.
8) Only include unique, non-trivial algorithms relevant to Python.
9) Avoid redundant, easily implemented, and simple algorithms that are answerable quickly.
10) Ensure each sub-algorithm can stand alone and be described independently of the others.
11) It is acceptable to generate fewer than 3 algorithms if the problem only requires one or two.

Here are some examples to follow with different numbers of provided algorithms: Details of easy, medium, and hard examples can be found on a GitHub repository [22].
Ensure your algorithms are brief, precise, and novel.
You will be provided with the following:

1) Function Signature and Docstring.
2) Description of the buggy code that needs fixing.
3) Identification of the type of bug (syntax, logic, etc.).
4) Description of the symptoms of the failure.
5) Test cases to validate the function.

Please only focus on generating sub-algorithms. Ensure that each sub-algorithm can stand alone, is essential to solving the overall problem, and can be described independently of the other sub-algorithms. Important: Please enclose each algorithm in `<algorithm>` algorithm contents `</algorithm>` HTML tags, as explained earlier in the examples. You can only create up to three novel algorithms. Remember to place the algorithm INSIDE the HTML tags.

Using prompt engineering, we carefully craft prompts to ensure that the models generate accurate and relevant feature vectors. Techniques from recent research, such as prompt distillation [23] and automatic chain of thought prompting [24], enhance the system's ability to understand and process complex queries. Buffer of Thoughts [23] identifies

prompt distillation techniques to extract generalizable features, while the Tree of Thoughts framework [21] provides deliberate problem-solving strategies. Retrieval-Augmented Generation for knowledge-intensive tasks [11] further supports our approach by improving model performance in handling complex queries. Creating a system that utilizes these techniques results in more effective and precise bug detection and repair.

III. Methodology

In this paper, we propose a novel system designed to enhance the code repair capabilities of LLMs by utilizing Stack Overflow as a data source. Our methodology, as shown in Figure 1, involves constructing a specialized database populated with technical solutions from Stack Overflow complemented with key extracted features of these posts, which are then used to inform and improve the LLM's performance in bug localization and code repair tasks. We implement an iterative feature extraction, search, and database expansion process, allowing the system to continually refine its understanding of relevant code features. If there are not relevant enough features in our database, we search for more Stack Overflow posts on Google and add these to our database. We use cosine similarity thresholds on embeddings to determine how relevant a stack overflow question is to a user question based on key extracted features. These features, extracted using specific prompt and zero temperature, include summaries of the Stack Overflow posts, in addition to the high-level algorithm that describes the correct implementation of a function in code. The details of iterative feature extraction are shown in Figure 2. By employing human-like natural language queries to search for pertinent technical information, our system aims to improve similarity search and feature generalization. The effectiveness of this approach is evaluated using the HumanEvalFix Python Benchmark [13], demonstrating significant accuracy improvements, particularly for smaller LLMs. Algorithms are an appropriate generalized feature of Stack Overflow post documents because Stack Overflow has an abundance of documentation about the correct algorithmic implementation of a buggy function. Stack Overflow also has an abundance of documentation about project dependency, compile-time, and library exceptions. Custom features can be extracted from these documents that describe them consistently and reliably for vector database search.

A. Preprocessing With Data Loading

A Stack Exchange Query is utilized to extract relevant Python coding bugs and results. Each of these curated posts that have a verified answer is added to the vector database. An LLM utilized feature extraction techniques to summarize the Stack Overflow post's questions and answers. For coding problems, a high-level description of the coding problem is called an algorithm. An algorithm is a specific type of document summary that can be utilized for coding problems that have to do with algorithmic implementation. An algorithm in this setting is the high-level 1 sentence description of what the code implements to achieve the desired outcome. Please reference the Prompt 1 for details on the system prompt utilized for feature extraction.

Prompt 2

You are a professional Python programmer. Fix the following buggy code using the provided context: docstring, function signature, buggy code, cause of the bug, and a relevant stack overflow post. Explain your fix step by step. Trace the provided test cases with detailed algorithm logic to solve the problem. Write the corrected implementation within `[code] [/code]` tokens. Ensure no extra characters or spaces outside these tags. Ensure to make all necessary imports based on the function signature and docstring. Template:

```
‘python
# Your Python imports here
# ALWAYS include the below import
whenever you write code

from typing import List, Tuple

""" This import only. This import is
exactly as it is. MEMORIZE IT.
ALWAYS, I REPEAT ALWAYS INCLUDE
THIS IMPORT exactly it is.
In all code you write, include: """

from typing import List, Tuple
""at the top.  Please DO NOT
import int, float, str, or bool
from typing - because this is incorrect.
 Remember, you must ALWAYS place your
 FINISHED CODE within ``` `` brackets
 before and after the code. Remember
 the brackets []"""
```

Feature extraction using Prompt 1 through an algorithmic summary is a generalizable feature of coding documents because the same LLM with zero temperature will generate similar algorithms for similar coding problems. The LLM has to introduce one to three algorithms that summarize and explain the coding problem and solution introduced by the Stack Overflow post.

B. System at Runtime

The system, at runtime, asks the user to provide the function they are writing with docstring documentation. This function is assumed to be incorrectly implemented. The LLM also provides the bug type and failure symptoms, which are used as additional context for the LLM to identify the coding bug. Based on this context, the large language model utilized feature extraction to create searchable features that describe the user's bug.

The LLM generates a summary of the coding bug, in addition to 1-3 algorithms that describe the potential implementation at a high level that would fix the current bug. The key question that needs to be asked is as follows: *If we were to propose an algorithm(s) to fix the current logical coding bug, what would it be?*

The extracted features are utilized to identify if there are similar enough features in the database that reflect a similar

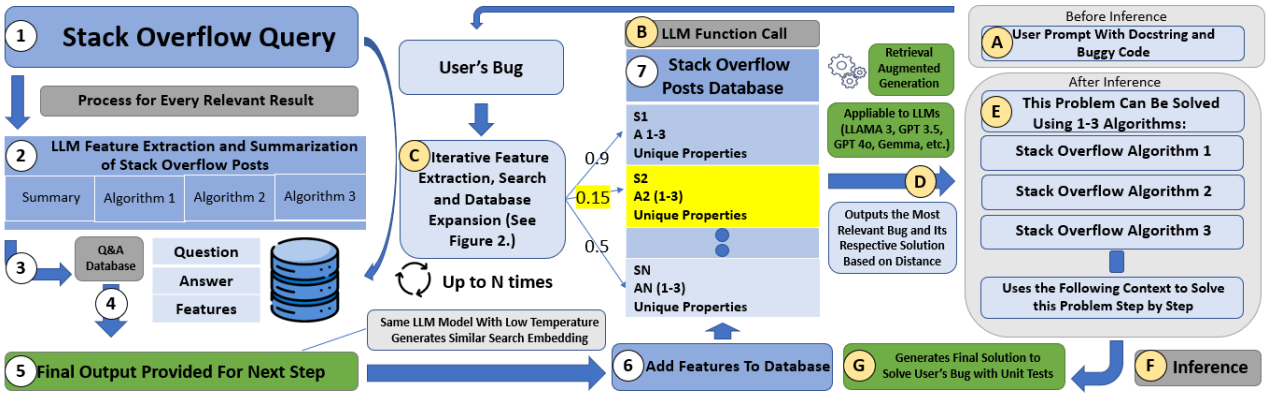

Fig. 1. Overview of the proposed framework: RAGFix. We employ iterative feature extraction with a prompt to enhance code repair for large language models using RAG.

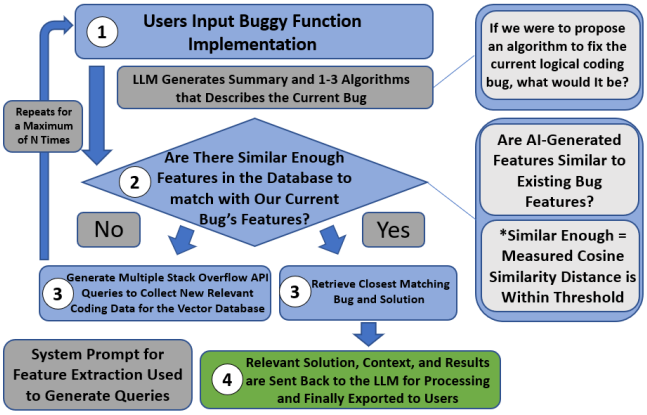

Fig. 2. Iterative Feature Extraction, Search, and Database Expansion

algorithm utilized to solve a coding problem. The goal is to identify if the AI-generated features are similar enough to existing features. This implementation utilizes cosine similarity with a cosine similarity threshold. This paper utilizes Chroma DB as the vector database backend. Chroma DB returns results in the range of [0,1], where 0 is the same document, and 1 is a completely unrelated document. The vector database identifies the most relevant stack overflow post per algorithm using similarity search. This implementation uses a constant cosine similarity threshold Theta as a hyperparameter for the system, and future implementations could use dynamic thresholds. In the current implementation, the threshold was set at 0.195 and different thresholds were experimented with from 0.1 to 0.23.

If the most relevant results are below the set threshold, this indicates there is a relevant stack overflow post in the vector database that is relevant to the user's query. In this case, the most relevant question and answer is returned to the client for additional context to the LLM.

If the most relevant results are not below the set threshold, the process of dynamic search and database expansion is utilized. Google search queries are crafted in the following format: Stack Overflow Algorithm describing potential correct implementation of buggy code Python. This query is used to search Google for relevant stack overflow posts. Since the LLM generates 1-3 algorithms that describe the coding bug, 1-3 searches are used where various result numbers can be considered, anywhere in the range of 2 to 10 results.

For each Stack Overflow post found based on these Google searches, the Stack Exchange API is utilized to extract the question and verify the answer for the relevant post. If a verified answer is not found for the relevant post, this post is ignored. The algorithm used to search for this post is appended to the post question. These results are extracted and added to the vector database, where the algorithm describing the stack overflow post is created as an embedding, and the question-and-answer documentation for the post is used as metadata. Currently, entire stack overflow posts are not being searched in embedding space. Only the algorithm describing the stack overflow post is searched in the embedding space.

Once all of the relevant stack overflow posts are added to the database, the iterative feature extraction, search, and database expansion process is repeated. With the new features in the vector database, the system determines if these new features are relevant enough features for the user bug.

The amount of times iterative search and database updating is run is a hyperparameter that can be set in the project. Currently, this process iterates up to 3 times. Future improvements to this work can consider generating more search queries to filter for more relevant results.

At inference time, the LLM is provided the buggy code, in addition to 1-3 stack overflow posts based on relevant algorithms that may help solve the current coding problem. Using the system prompt provided in Prompt 2, the LLM is guided through three algorithms that could potentially solve the current problem, along with a Stack Overflow post that offers a correct implementation of the algorithm.

The LLM generates the final solution to the user's prompt and this solution is run at run time in a Python code sandbox with the associated dataset test cases. If the provided LLM solution passes all of the test cases, this is considered the

correct code repair of the buggy code. If the provided solution fails any test cases, this is considered a zero-shot failure at resolving the current coding bug.

IV. Evaluation

This paper uses the HumanEvalFix benchmark from the Octopack: Instruction Tuning Code Large Language Models [13] benchmark for Python code repair. There are 164 examples in the dataset. Each example includes the function header, buggy code, bug type, bug description, and test cases for evaluating the generated code. The proposed methodology does not involve training any models; all 164 samples are used for evaluation. This paper implements our solution for the LLAMA-3 8B and 70B parameter models. For the evaluations, a vanilla LLM with zero shot reasoning is evaluated against the LLM that is provided additional context from the stack overflow vector database question and answer repository. The generated code evaluated with the provided test cases is used to calculate an accuracy score seen in the results below. Our proposed method provides valuable context for small to medium-sized LLMs in the form of relevant coding documentation that helps an LLM reason through software repair. We show that our technique has promising early results for small LLMs. We conclude that using a Stack Overflow vector database with advanced search techniques enhances LLMs in code repair by applying RAG. As demonstrated in Figure 3, The proposed technique improves LLMS using Chain of thought (CoT) with Stack Overflow RAG on the HumanEvalFix Python benchmark. Llama3-8b's detection and localization accuracy rises from 41.4% in the Zero Shot setting to 51.2% with RAGFix, while Llama3-70b improves from 72.5% to 78.0%. This demonstrates the effectiveness of CoT with RAG in enhancing model performance. Using key feature extraction techniques, we improved LLMs' ability to search natural language APIs efficiently. Overall, identifying generalizable features of coding bugs has improved our system's similarity search, which enhanced bug resolution.

A. Results and Discussion

To thoroughly evaluate our approach, we develop several research questions that guide our investigation.

1) RQ1. What data source and data arrangement strategy is relevant for LLM code repair with RAG? : The primary data source for LLM code repair with RAG is Stack Overflow, given its extensive repository of coding questions and verified answers. The arrangement strategy involves dynamically collecting and curating relevant posts into a vector database, enriched with extracted features such as problem summaries and algorithmic descriptions. This structured knowledge graph allows for efficient retrieval of contextually relevant solutions, enhancing the LLM's ability to address specific coding issues. Utilizing cosine similarity in embeddings of these features ensures that only the most pertinent information is used, thus optimizing the bug localization and repair process.

2) RQ2. How can an LLM craft human-like natural language API queries (e.g., Stack Overflow) to search for relevant technical information?: An LLM can craft human-like natural language API queries by leveraging prompt engineering techniques and pre-trained models. By generating high-level summaries and algorithmic descriptions of the coding problem, the LLM can formulate precise search queries that mimic human inquiry patterns. This involves creating search queries that reflect the specific bug symptoms and potential solutions, thus enabling the retrieval of highly relevant Stack Overflow posts. The iterative process of feature extraction, search, and database expansion ensures that the queries evolve to target the most accurate and useful technical information.

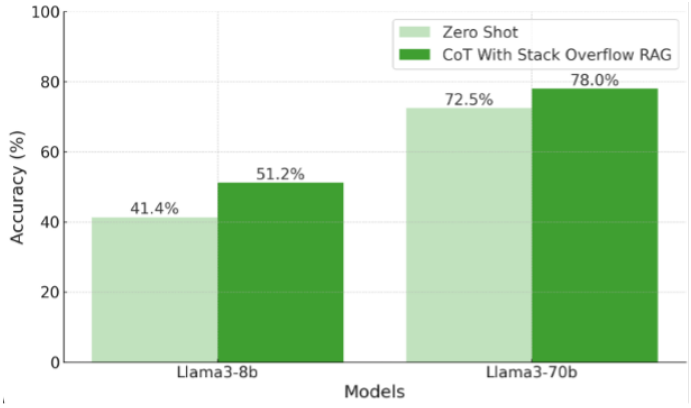

Fig. 3. LLM Model Performance on HumanEvalFix Python Benchmark

3) RQ3. What are relevant and generalizable features that can be extracted from code to improve similarity search on technical documentation?: Relevant and generalizable features extracted from code include high-level algorithmic summaries, problem descriptions, and the nature of the bugs and their symptoms. These features, generated by the LLM, provide a consistent and structured representation of the coding issues, facilitating effective similarity searches in the technical documentation. By focusing on these abstracted features rather than specific code details, the system can identify and retrieve solutions that are applicable across different contexts and coding environments, thus enhancing the LLM's ability to generalize and solve a wide range of coding problems.

V. Limitations

Our current implementation requires a docstring for any buggy function that needs to be fixed, which can be a limitation if users do not provide one. Additionally, while the generalizable features we extract work well for algorithmic coding problems, different features may be needed for other types of coding issues, such as compile-time and runtime errors often seen on Stack Overflow. In the future, we aim to enhance our technique and apply it to LLMs for solving more complex coding problems. Currently, our system is limited to small- to medium-sized LLMs, ranging from 8 billion to 70 billion parameters. We plan to explore larger models in the future. Furthermore, our approach has only been evaluated on the

HumanEvalFix benchmark for Python. We intend to expand our evaluation to include other benchmarks and programming languages, such as Java and C, to generalize our approach to addressing more challenging coding problems with LLMs.

VI. Conclusions and Future Work

This research presents a novel system that enhances Large Language Models' (LLMs) code repair capabilities by leveraging Retrieval-Augmented Generation (RAG) and a dynamically curated database of Stack Overflow posts. By constructing an embedded knowledge graph and utilizing cosine similarity on LLM-synthesized embeddings, the proposed system significantly improves bug localization and resolution. Evaluation using the HumanEvalFix benchmark indicates that our approach effectively refines the performance of various LLMs, including smaller models with limited parametric knowledge. The system's ability to create and utilize a searchable knowledge graph, along with advanced search techniques and prompt engineering, demonstrates its potential to reduce debugging time and enhance software development processes. Our methodology not only addresses the challenges of bug fixing but also advances the application of LLMs in software engineering, paving the way for more efficient and precise debugging tools. Future research will expand our vector database to include more programming languages and provide additional context. Moreover, we plan to refine the transformer model for generating human-like Stack Overflow search queries to achieve more relevant and accurate results. We also aim to develop benchmarks to specifically target managing project dependencies and compile-time errors using Stack Overflow's extensive knowledge base. User studies will provide feedback for iterative improvements, which will improve the conditioning of the system architecture and advance LLM-based code repair.

VII. Data Availability

Our approach, benchmarks, and evaluation results are available on a GitHub repository [22].

VIII. Acknowledgement

This research work was conducted at Oakland University in the REU program and supported by the NSF grant #CNS-2349663. Any opinions, findings, and conclusions or recommendations expressed in this work are those of the author(s) and do not necessarily reflect the views of the NSF.

References

[1] A. Zeller, *Why programs fail: a guide to systematic debugging*. Morgan Kaufmann, 2009.

[2] S. Omari, K. Basnet, and M. Wardat, "Investigating large language models capabilities for automatic code repair in python," *Cluster Computing*, pp. 1–15, 2024.

[3] M. Piastou, "Overcoming challenges in applying ai guidance to complex and legacy codebases," *Journal of Artificial Intelligence Research*, vol. 4, no. 1, pp. 312–331, 2024.

[4] M. Wardat, W. Le, and H. Rajan, "Deeplocalize: Fault localization for deep neural networks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 251–262.

[5] M. Wardat, B. D. Cruz, W. Le, and H. Rajan, "Deepdiagnosis: automatically diagnosing faults and recommending actionable fixes in deep learning programs," in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 561–572.

[6] M. AI, "Llama 3," https://github.com/llegomark/meta-llama-3, 2024, gitHub repository, [Accessed 2024-11-15].

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[8] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[9] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[10] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[11] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[12] P. Wang, X. Xie, X. Wang, and N. Zhang, "Reasoning through memorization: Nearest neighbor knowledge graph embeddings," in *CCF International Conference on Natural Language Processing and Chinese Computing*. Springer, 2023, pp. 111–122.

[13] N. Muennighoff, Q. Liu, A. Zebaze, Q. Zheng, B. Hui, T. Y. Zhuo, S. Singh, X. Tang, L. Von Werra, and S. Longpre, "Octopack: Instruction tuning code large language models," *arXiv preprint arXiv:2308.07124*, 2023.

[14] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 510–520.

[15] M. Wardat, B. D. Cruz, W. Le, and H. Rajan, "An effective data-driven approach for localizing deep learning faults," *arXiv preprint arXiv:2307.08947*, 2023.

[16] S. Khoshraftar and A. An, "A survey on graph representation learning methods," *ACM Transactions on Intelligent Systems and Technology*, vol. 15, no. 1, pp. 1–55, 2024.

[17] L. Wang, J. Luo, S. Deng, and X. Guo, "Rocs: Knowledge graph embedding based on joint cosine similarity," *Electronics*, vol. 13, no. 1, p. 147, 2023.

[18] C. Mavromatis and G. Karypis, "Gnn-rag: Graph neural retrieval for large language model reasoning," *arXiv preprint arXiv:2405.20139*, 2024.

[19] S. Pan, L. Luo, Y. Wang, C. Chen, J. Wang, and X. Wu, "Unifying large language models and knowledge graphs: A roadmap," *IEEE Transactions on Knowledge and Data Engineering*, 2024.

[20] J. Jiang, K. Zhou, W. X. Zhao, and J.-R. Wen, "Unikgqa: Unified retrieval and reasoning for solving multi-hop question answering over knowledge graph," *arXiv preprint arXiv:2212.00959*, 2022.

[21] S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[22] Laboratory for Software Innovation, 2024. [Online]. Available: https://github.com/Laboratory-software-Innovation/RAGFix

[23] L. Yang, Z. Yu, T. Zhang, S. Cao, M. Xu, W. Zhang, J. E. Gonzalez, and B. Cui, "Buffer of thoughts: Thought-augmented reasoning with large language models," 2024.

[24] Z. Zhang, A. Zhang, M. Li, and A. Smola, "Automatic chain of thought prompting in large language models," *arXiv preprint arXiv:2210.03493*, 2022.