소프트웨어 엔지니어링 분야에서 Retrieval-Augmented Generation 구현에 대한 고찰

디미트리, 류덕산
전북대학교 소프트웨어공학과
cita18@jbnu.ac.kr, duksan.ryu@jbnu.ac.kr

Survey on Implementing Retrieval-Augmented Generation in Software Engineering

Dimitri Romain Bekale Be Ndong, Duksan Ryu
Department of Software Engineering
Jeonbuk National University

요약

Retrieval-Augmented Generation (RAG)은 외부 지식 소스를 활용하여 대규모 언어 모델(LLM)의 정확성과 문맥 이해 능력을 향상시키는 혁신적인 기술로 주목받고 있다. 본 포괄적인 문헌 검토는 소프트웨어 엔지니어링(SE) 분야에서 RAG의 적용 사례, 장점, 도전 과제, 그리고 모범 사례를 심층적으로 분석하는 것을 목표로 한다. 이를 위해 총 24편의 소프트웨어 엔지니어링 관련 과학 논문을 체계적으로 검토하였다. 연구 결과, RAG는 SE 도구의 성능을 개선하는 동시에 계산 비용, 문맥 크기, 외부 데이터베이스의 품질, 지식 충돌 등과 같은 실질적인 문제점을 노출시켰다. 이 연구는 하이브리드 검색 전략, 정제된 데이터베이스 구축, 그리고 에이전트 기반 RAG 접근법이 SE 분야에서 최적의 해결책으로 활용될 수 있음을 강조한다. 이를 통해 실무자와 연구자들은 개발 및 배포 프로세스를 개선할 수 있는 실행 가능한 통찰력을 얻을 수 있다.

Abstract

Retrieval-augmented generation (RAG) is revolutionizing large language models (LLMs) by incorporating external knowledge sources to deliver accurate, context-aware outputs. This comprehensive literature review aims to investigate the existing implementations of RAG in software engineering (SE), as well as its benefits, challenges, and best practices. The study shows that RAG enhances SE tools while revealing significant challenges such as computational cost, context size, and knowledge conflict. We highlighted hybrid search, refined databases, and agentic RAG as optimal approaches when leveraging RAG in SE. This study gives practitioners and researchers actionable insights to refine their development and deployment processes.

Keywords: Software Engineering, Large Language Models, Retrieval-Augmented Generation.

1. Introduction

Artificial Intelligence (AI) powered software engineering tools have been introduced as an effective and cost-efficient solution to tackle the challenges practitioners face as software becomes more complex [1]. In the past decade, Large Language Models (LLMs) have revolutionized software engineering by providing unprecedented code generation, documentation, and intelligent assistance capabilities [2]. However, these models often struggle with domain-specific knowledge, contextual relevance, and maintaining up-to-date information. Retrieval Augmented Generation (RAG) emerges as a promising approach to address these limitations, bridging the gap between generative AI capabilities and precise, context-aware software engineering solutions. RAG represents a sophisticated methodology that enhances traditional language models by dynamically incorporating external knowledge repositories during the generation process [3].

However, implementing RAG in Software Engineering (SE) presents unique challenges. Issues such as the scalability of retrieval systems, the relevance of retrieved knowledge, and alignment with the LLM's parametric knowledge and retrieved content require careful consideration. Additionally, the computational overhead of integrating retrieval mechanisms with LLMs can impact performance, especially in real-time applications. This article explores the common characteristics of system retrieval in SE, its advantages, and the practical challenges inherent in its implementation. It also proposes optimal strategies for effectively integrating RAG into software engineering tools. By understanding these factors, developers and researchers can unlock the full potential of RAG-powered tools to revolutionize the software engineering landscape.

2. Background

2.1. Automated software engineering

As software systems and software engineering tasks have become increasingly complex over the past decades, there has been a need to develop automated tools to assist practitioners. Those tools powered by advanced deep learning (DL) models ensure efficiency and productivity while minimizing errors in the software development lifecycle process. The recent emergence of LLMs has offered researchers unprecedented opportunities to refine and enhance existing technologies, resulting in their adoption across nearly all

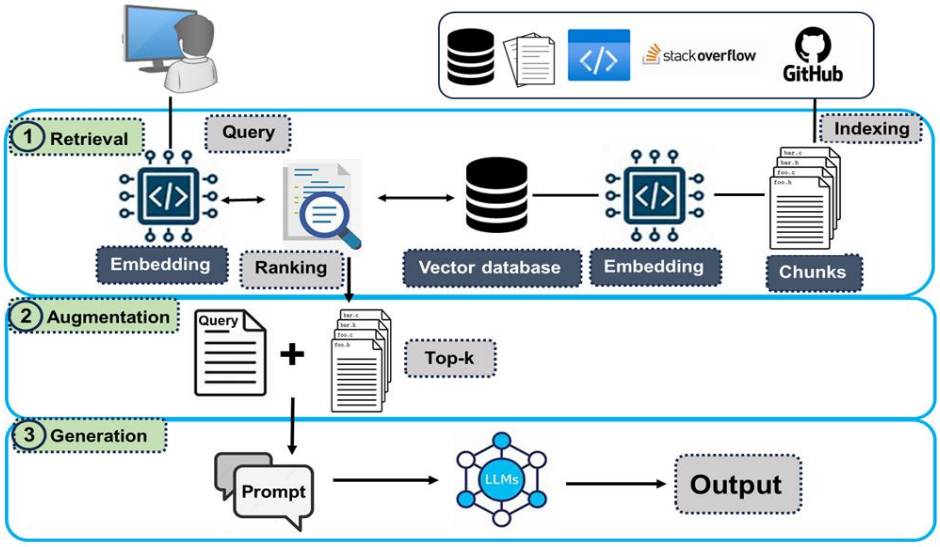

Fig. 1. The retrieval Augmented Generation workflow

areas of software engineering. Code task-related LLMs (CLMs) represent one significant advancement in their ability to understand natural language, code syntax, and code semantics. They revolutionized software development with their increasingly powerful code-text learning capacities, enabling automated tasks such as code generation, code translation, code summarization, code completion, clone detection, program repair, automated testing, vulnerability detection, etc. [4].

Several recent studies proposed new frameworks that leverage CLMs to perform these tasks from simple prompts, using either natural language, code, or images as input. However, CLMs are initially trained on large amounts of data and must be fine-tuned to perform downstream tasks effectively. This fine-tuning process is not only computationally expensive and time-consuming but also significantly influenced by the quality of the data used. Another challenge of this approach is the necessity of repeating it each time updates are made, for example, when adding new documentation. Several techniques were introduced to overcome this limitation, including few-shots learning and RAG. RAG is being widely adopted in LLMs-based tools in SE, consequently highlighting the need for a thorough review of this technology.

2.2. Retrieval Augmented Generation

While LLMs have incredible capabilities to produce realistic content, such as text, images, or code snippets, they also have limitations. Often, LLMs confidently generate content that appears correct but is factually inaccurate, a phenomenon known as hallucination. And because their knowledge is based solely on their training data, they struggle with up-to-date information and any task requiring domain-specific knowledge beyond that training data. RAG addresses the issues outlined above by referencing external knowledge, incorporating up-to-date facts, and integrating domain-specific context alongside the user's query to enhance the quality and relevance of LLM-generated content. RAG has already been widely used; its integration with LLMs advanced technologies like chatbots, a software application that retrieves relevant, up-to-date information from a private database to address users' queries or complaints, is an example. RAG is also increasingly used in SE, where it significantly helps improve LLM-powered automated tools by automatically retrieving contextually relevant data from documentation, code repositories, code search engines, or forums [3].

As shown in Fig. 1, a basic RAG framework consists of three major processes: retrieval, augmentation, and generation. The retrieval step involves a retriever collecting context or relevant knowledge from a designated vector database, given a user's query input of the LLM. The vector database is constructed by embedding chunks of data collected from an external source. The process is performed by measuring the similarities between the query and the documents in the database. The most relevant documents to the query are then ranked and retrieved (top-*k*). It is worth noting that in this section, we refer to documents, any information, data, or content existing in the knowledge base. The augmentation step combines retrieved information with the query and passes it to the generator as a prompt. This integration may incorporate additional prompts or explanatory content. The generation task denotes simply generating a context-aware augmented output. The architecture of the generator depends on the downstream task, but typically, in SE, generators are code task-related LLMs and general-purpose LLMs. The growing trend of incorporating RAG technology in SE gives rise to the necessity of a more profound understanding of its outcomes. This study thereby aims to offer a comprehensive understanding of the benefits of integrating RAG into LLM-based tools for software engineering, examine the challenges encountered, and explore various strategies for optimization.

3. Methodology

In our study's initial phase, we established research objectives and formulated preliminary research questions, acknowledging their potential evolution as we gathered insights from selected studies. We developed a detailed review protocol that outlined our inclusion and exclusion criteria, search methodology, terminology, quality assessment standards, data extraction procedures, and synthesis methods.

To develop our search approach, we identified databases and search engines likely to contain relevant research papers. We formulated search queries using Boolean operators and key-

words, documenting all search terms and their combinations. This search strategy underwent continuous refinement to maximize the retrieval of pertinent literature.
Our study selection process followed three distinct phases. Initially, we screened titles and abstracts according to our inclusion criteria, identifying potentially relevant studies for further examination. The second phase involved an in-depth analysis of these studies, applying our predetermined criteria to determine final inclusion or exclusion. In the third phase, we conducted a thorough quality assessment, evaluating each study's methodological rigor, depth of analysis, and potential biases.
Following these stages, we proceeded with data extraction, synthesis, and manuscript writing. While these steps may appear sequential, it is worth noting that the process was inherently iterative, continuously refined, and improved throughout the study.

4. Research Setup

4.1. Research questions

This systematic literature review was conducted in conformity with the instructions provided by [5]. Seven areas of SE were targeted: software vulnerability detection, code summarization, code generation, code search, code completion, automated program repair, and bug identification. The review protocol was developed to answer the subsequent questions:

RQ1: How is RAG commonly implemented in software engineering?

This question explores the typical characteristics of a RAG framework in SE by highlighting the different elements and methods used to retrieve, augment, and generate the desired output, namely code, text, or class (buggy or clean). It also describes the nature of the external database used to collect information.

RQ2: How does incorporating a retrieval system affect the performance of LLM-based tools in software engineering?

This question examines how retrieval systems affect the effectiveness of the tools they are integrated into. We delve into the benefits of using this technology.

RQ3: What potential limitations could emerge when leveraging retrieval systems?

This question aims to review the challenges encountered when incorporating RAG.

RQ4: What strategies could enhance the effectiveness of RAG implementation in software engineering?

This question highlights the best practices and optimization methods for empowering a retrieval system.

4.2. Search strategy

Search terms were formed by concatenating primary keywords with secondary keywords (Table 1). The primary keywords were separated by the Boolean operator “OR”, then the operator “AND” was added, followed by the subsequent secondary keywords. For example, “rag OR retrieval AND code OR bug OR program repair.” The resulting terms were then used to search relevant papers in the databases shown in Table 1. The publication period was restricted to one year, from November 2023 to November 2024.

4.3. Selection criteria

Based on the search strategy described previously, a total of 45 papers were selected for initial examination based on the inclusion criteria listed in Table 2. Following the initial assessment, 21 papers were discarded according to the exclusion criteria mentioned in Table 2. Then, 24 articles were selected for thorough analysis.

4.4. Quality assessment

We followed the quality assessment checklist provided by [5] to assess the quality of the selected studies. The checklist offers questions to evaluate methodological rigor, empirical evidence quality, software engineering relevance, etc. We also included an evaluation of the clarity of RAG implementation, namely the characteristics of the components.

4.5. Data extraction and data synthesis

An Excel file table was created for each domain of the chosen studies, and the data extracted from these tables aimed to address the research questions directly. The table summarized the details of the implementation, benefits, challenges, and best practices identified in the studies. The authors' names, titles, and publication details were also cited. The data synthesis aimed to aggregate and combine findings from various studies to formulate suitable answers and address the research questions. By combining multiple studies with similar viewpoints and results, we strengthened the evidence for the research, leading to more conclusive answers to the research questions.

5. Results and Discussion

This section presents the results of the four research questions proposed earlier, derived from the selected studies.

5.1. Description of primary studies

A total of 24 studies were selected for this review. They mostly used available datasets and provided the source code for replicability. The leading digital library from which the papers were retrieved is arXiv (11), followed by IEEE Xplore (4), ACL Anthology (3), ACM Digital (4), Science Direct (1), and Springer (1) of the studies. Table 3 provide an overview of the selected studies and their subfields in SE. Twenty-one studies were categorized into seven groups based on their related topics. A separate category was also created for general-purpose studies on RAG to encompass the remaining research. This collection includes a paper on software engineering highlighting the challenges faced when using retrieval systems in the field [6] and studies outside software engineering that provide valuable insights into common challenges when using large language models.

5.2. RQ1: How is RAG commonly implemented in software engineering?

The concept of RAG involves collecting additional data from a vector database that subsequently serves as context to augment the final output of a LLM given an input query. The main difference between different domains is the input query and retrieved data. In SE, the input and the external knowledge are mostly code snippets or text explaining the desired code in natural language. Consequently, external databases are represented mainly by programming language library documentation, programming solutions, and code snippets from code repositories or forum posts, as well as their explanation in natural language, online tutorials, execution feedback, and any API meta-data (function descrip-

tion, parameters, and their descriptions). These data are then embedded by a CLM or a general information retriever and turned into a vector database. Two types of retrievers are then used to collect this information: sparse and dense retrievers [SS10, SS17]. Sparse retrievers are general-purpose information retrieval algorithms. They are word-based matching and retrieve information based on the overlap of terms between a query and a document. The most widely used sparse retriever in the selected studies by far is BM25 [7]. Dense retrievers, on the other hand, embed the query and the documents into continuous vector space. They are trainable and highly adaptable and can also learn semantic similarities. Most studies use a transformer-based CLM like CodeT5 [8], pre-trained or fine-tuned to retrieve the most relevant context. Pretrained models are enriched through few-shots or contrastive learning, and fine-tuned models are fine-tuned with domain-specific examples. The augmentation process consists of filling an existing prompt template or automatically generated template with the query combined with the retrieved top-k most relevant contexts. The generation step is reached, where the LLM generates the augmented output. Studies use general-purpose LLMs and CLMs for generation, such as OpenAI GPT models, CodeT5+ [9], or Starcoder [10] [SS2].

Table 1: Search strategy

Database / Search engines	**Primary keywords**	**Secondary keywords**
ACM Digital Library	Retrieval augmented generation	Code
ArXiv	RAG	Software engineering
Google Scholar	Retrieval system	Bug
IEEE Xplore	Retrieval	Program repair
Science Direct		Commit message
Springer		
ACL Anthology		

Table 2: Selection criteria

Inclusion criteria	**Exclusion criteria**
Empirical studies implementing RAG in SE tools.	Studies focused on RAG for general AI papers.
Empirical studies on RAG architecture and general purpose used cases.	Studies with insufficient analysis on the impact of the retrieval system.
Systematic literature review of RAG.	Studies lacking empirical evidence or practical implementations.
Empirical studies on autonomous SE.	Questionable experimental results.

5.3. RQ2: How does incorporating a retrieval system affect the performance of LLM-based tools in software engineering?

All selected studies showed that incorporating RAG into LLM-based tools enhances their performance. SS9 reported that RAG allowed their tool to be 7% to 30% more accurate. They fine-tuned an encoder-decoder CLM with domain-specific samples and used the trained encoder part as the retriever. This strategy helped retrieve better examples to assist the generator. SS8 revealed an improvement of up to 7.26 points compared to using the tool without a retrieval component. More impressively, SS16 reported an improvement of up to 72% when integrating their RAG approach into different CLMs, and one-shot learning with general-purpose LLM by incorporating a retrieved context yielded a net improvement of up to 102%. Their framework's success lies in using a hybrid (sparse + dense) retriever and a high-quality database source. SS4 reported that RAG outperformed a model fine-tuned on a much larger dataset at syntax correctness.

Table 3. Selected studies

Subfield	**Study ID.**	**Paper**	**Reference no.**
Code generation and Code completion	SS1	Wu, et al. (2024)	[11]
	SS2	Tan, et al. (2024)	[12]
	SS3	Wang, et al. (2024)	[13]
	SS4	Bassamzedeh, et al. (2024)	[14]
	SS5	Guo, et al. (2025)	[15]
	SS21	Yu, et al. (2024)	[16]
	SS6	Bui, et al. (2024)	[17]
	SS7	Dutta, et al. (2024)	[18]
	SS8	Tang, et al. (2024)	[19]
	SS9	Lu, et al. (2024)	[20]
	SS22	Yu-chen, et al. (2024)	[21]
	SS23	Wei, et al. (2024)	[22]
Code Search	SS10	Chen, et al. (2024)	[23]
	SS11	Haochen, et al. (2024)	[24]
	SS12	Gou, et al. (2024)	[25]
	SS13	Liu, et al. (2024)	[26]
T.G.*	SS14	Shin, et al. (2024)	[27]
V.A.**	SS15	Daneshvar, et al. (2024)	[28]
C.M.G.***	SS16	Zhang, et al. (2024)	[29]
Bug report	SS17	Yilmaz, et al. (2024)	[30]
	SS18	Harzevili, et al. (2024)	[31]
Others	SS19	Barnett, et al. (2024)	[6]
	SS20	Xu, et al. (2024)	[32]
	SS24	Yunfan, et al. (2023)	[33]

*T.G.: test generation.
**V.A.: vulnerability augmentation.
***C.M.G.: commit message generation

5.4. RQ3: What potential limitations could emerge when leveraging retrieval systems?

5.4.1. Computational cost

The selected studies mostly used dense retrievers over sparse retrievers, as SS3 showed that sparse retrievers underperform due to their inability to learn semantic similarities. However, dense embedding models are computationally expensive because they are trained to learn those similarities. SS3 and SS9 showed that dense models require up to 5x larger index storage and add nearly 100x latency to embed documents.

5.4.2. Context size

A second limitation is the length of the documents retrieved to augment the generation process. SS7 reported that lengthy documents with over 3000 tokens and a number of retrieved contexts exceeding 5 lower the model's performance. SS4 and SS8 highlight that increasing the number of API descriptions confused the LLM, resulting in poor accuracy.

5.4.3. Quality of database

The success of a retrieval system is highly reliant on the quality of the context retrieved. SS3 shared that the genera-

tor struggled with low-quality context. SS14 investigated the impact of the document provided on the retrieval element. It showed that mixed documents were not as valuable, probably because of excess noise. Therefore, a noisy external knowledge base, which contains a high amount of irrelevant or redundant information, compromises the effectiveness of the retrieval system. SS5 reported that web search, the most widely used RAG method, is not as effective because the retrieved contexts can be noisy, leading the LLM to be misled by irrelevant or contradictory information that affects its output. An example is the platform StackOverflow, where even though a relevant code snippet and its explanation can be retrieved, excessive chats and arguments may lead to noisy contexts.

5.4.4. Knowledge conflict

While providing a list of correct API recommendations along with relevant examples of code snippets to the generator, SS1 surprisingly found that the generator still made a significant amount of errors. The authors suggested that the LLM completely disregarded the provided context. Existing knowledge conflicts in LLM can explain this phenomenon. LLMs can deliberately decide, for some reason, to disregard external knowledge and use their parametric knowledge when solving a given task. This is known as context-memory conflict. SS20 proposes an extended comprehensive study on the subject. [34] identified 3 types of knowledge conflict causes: (1) misinformation conflict deriving from misleading facts and false narratives within the training dataset, (2) temporal conflict occurring when the knowledge changes over time, leading to outdated LLM's parametric knowledge, and (3) semantic conflict arising when words with multiple meaning cause ambiguous interpretation.

5.5. RQ4: What strategies could enhance the effectiveness of RAG implementation in software engineering?

5.5.1. Choosing relevant knowledge sources

The relevance of the external knowledge source is a critical step when building a retrieval system. SS5 compared different data sources and found that explained code, namely code snippets and their documentation, was the most relevant source to enhance the performance of the generator and the retriever. They also showed that when using execution feedback as context, the effectiveness of the retrieval system was highly dependent on how well-explained error messages are and whether the missing element is correctly pointed out. SS4 also reported that adding API descriptions, functions, and parameter descriptions significantly improves the model's performance. SS15 used a clustering approach to group data according to their nature and function, for example, the type of vulnerabilities or commit messages; this facilitates the retrieval of more relevant information. Filtering the noise in the database is an important step as well. SS22 proposed a data preprocessing method to filter the most pertinent information from the document in the source. The technique consists of segmenting text focusing on meaningful units. The method allowed a 63-point increase in retrieval accuracy.

5.5.2. Hybrid search

SS16 investigated the effect of combining information retrieval (BM25) with transformer-based dense retrievers. In their proposed approach, the two retrievers calculate the similarities between the query and the documents separately and then retrieve a combination of the most relevant contexts, a process called hybrid retriever or hybrid search. Compared with single retrievers and a scenario where no retriever is used, hybrid retriever exhibits up to 60% improvement. Likewise, SS6 combined a lexical retriever (using Byte-Pair Encoding BPE [35]) with a semantic retriever (using UniXCoder [36]) and subsequently compared the performance of their framework with several popular techniques. The ablation study showed that the hybrid approach outperforms the others by up to 4.8%. These results suggest that using a hybrid search enhances the quality of the retrieved context.

5.5.3. Retriever and generator fine-tuning

Fine-tuning the retriever and generator with sufficient task-specific samples can boost the output. SS9 proposed a framework that jointly fine-tunes the retriever and the generator iteratively over the retrieved context. A fine-tuned retriever collects relevant information from the knowledge source. The retrieved context and the input are used to train the generator, which then assesses the relevance of the provided context by calculating the generation loss for the generation output. Following a defined threshold, a score lower than the threshold results in feedback to the retriever, which will use that information to retrieve more relevant context. The results show that this training strategy improves the generation, and their approach outperforms the best-performing baselines by up to 30%.

5.5.4. Agentic RAG

Agentic RAG employs a multi-agent-based retrieval system to augment the generator's output. Integrating multiple agents, such as rewriters, re-rankers, and others, can significantly improve outcomes. SS13 used a multi-agent system retrieval where one agent would understand the query, and the other would transform the knowledge gained from the first agent into a graph query to enhance the retrieval of relevant information. SS18 built a three agents-based framework for bug detection: one agent for bug detection, the second for bug cause analysis, and the last one for patch generation. These two frameworks were shown to be more efficient than the regular RAG system. SS11 and SS2 demonstrated the benefits of rewriting natural language queries to increase the chances of locating an ideal match in the database, as well as normalizing the style of database code to align with a given input snippet. This approach resulted in a retrieval accuracy improvement of up to 35%. Rewriter agents can be used in two roles: one can refine and clarify query inputs or format code into a suitable structure before retrieval (pre-retrieval). Another can rewrite retrieved documents to better fit the context before ranking (post-retrieval) [SS24].

Meanwhile, the re-ranker agent enhances relevance by prioritizing key content and placing it strategically at the prompt's edges for emphasis [SS24, SS21, SS23]. Additionally, it filters out redundant information, enabling the generator to work with the most pertinent context. SS12 further confirmed that incorporating a re-ranker significantly improves generator output. A network of autonomous agents can be used at different steps of the RAG process with adequate interaction and management mechanisms. However, this is beyond the scope of this study. Future work on this topic should emphasize building a multi-agent RAG pipeline where agents will effectively address the challenges faced in the current system configuration.

6. Threats to validity

A potential validity threat to this study is that a single researcher conducted the entire systematic literature review process. Although one person was responsible for selecting and reviewing the papers, weekly meetings were held with a professor to monitor the process and provide guidance on improving the research protocol. Another threat to this systematic review's validity is that most papers selected for this work (11 out of 24) are in the prereview stage and preprint, retrieved from arXiv [37] due to the restricted period. To mitigate this issue, the results and methodological rigor were double-checked, and only high-quality papers were selected.

7. Conclusion

This study provides a comprehensive systematic literature review on the advanced potential of Retrieval-Augmented Generation (RAG) in enhancing the capabilities of Large Language Model-based software engineering tools. We thoroughly examined 24 papers published between November 2023 and November 2024 to highlight the implementation details, benefits, challenges, and best practices for integrating RAG in this field. Our findings indicate that through effective retrieval mechanisms, RAG demonstrates significant accuracy and domain adaptability advantages by addressing challenges like costly fine-tuning, inaccurate outputs, and context awareness. However, the study also highlights critical challenges that must be addressed, such as the quality of the data source, LLM knowledge conflict, the retriever's computational complexity, and limitations in retrieved document size. We proposed optimal approaches as a roadmap for practitioners and researchers to refine RAG implementations, paving the way for more robust, efficient, and reliable software engineering solutions.

Acknowledgments

This research was supported by "Basic Science Research Program" (NRF-2022R1I1A3069233) through the National Research Foundation of Korea(NRF) funded by the Ministry of Education (MOE), and the MSIT(Ministry of Science and ICT), Korea, under the ITRC(Information Technology Research Center) support program(IITP-2025-2020-0-01795) supervised by the IITP(Institute for Information & Communications Technology Planning & Evaluation) and the Nuclear Safety Research Program through the Korea Foundation Of Nuclear Safety(KoFONS) using the financial resource granted by the Nuclear Safety and Security Commission(NSSC) of the Republic of Korea. (No. 2105030)

References

[1]. Daniel Russo. "Navigating the Complexity of Generative AI Adoption in Software Engineering", ACM Transactions on Software Engineering and Methodology, Vol. 33, 5, Article 135, pp. 1-50, 2024.

[2]. Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang, "Large Language Models for Software Engineering: A Systematic Literature Review", ACM Transactions on Software Engineering Methodology, Vol. 33, No. 8, Article 220, pp. 1-79, 2024.

[3]. Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks", Advances in Neural Information Processing Systems, Vol. 33, pp. 9459–9474, 2020

[4]. Jiang, Juyong, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. "A Survey on Large Language Models for Code Generation." ArXiv preprint arXiv:2406.00515, 2024.

[5]. Barbara Kitchenham, Pearl Brereton, David Budgen, Mark Turner, John Bailey, Stephen Linkman. "Systematic literature reviews in software engineering - A systematic literature review", Information and Software Technology, Vol. 51, pp. 7-15, 2009.

[6]. Scott Barnett, Stefanus Kurniawan, Srikanth Thudumu, Zach Brannelly, Mohamed Abdelrazek, "Seven Failure Points When Engineering a Retrieval Augmented Generation System." 2024 IEEE/ACM 3rd International Conference on AI Engineering – Software Engineering for AI (CAIN), pp. 194-199, 2024.

[7]. Stephen Robertson and Hugo Zaragoza. "The Probabilistic Relevance Framework: BM25 and Beyond." Foundations and Trends in Information Retrieval, Vol. 3: No. 4, pages 333–389, 2009.

[8]. Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation." In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, pages 8696–8708, 2021.

[9]. Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. "CodeT5+: Open Code Large Language Models for Code Understanding and Generation". In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pages 1069–1088, 2023.

[10]. Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya,

Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, Harm de Vries.“StarCoder: may the source be with you!” ArXiv, abs/2305.06161, 2023.

[11]. Wu, Yixi, Pengfei He, Zehao Wang, Shaowei Wang, Yuan Tian, and Tse-Hsun Chen. “A Comprehensive Framework for Evaluating API-oriented Code Generation in Large Language Models.” ArXiv preprint arXiv:2409.15228, 2024.

[12]. Tan, Hanzhuo, Qi Luo, Ling Jiang, Zizheng Zhan, Jing Li, Haotian Zhang, and Yuqun Zhang. “Prompt-based Code Completion via Multi-Retrieval Augmented Generation.”ArXiv preprint arXiv:2405.07530, 2024.

[13]. Zhiruo Wang, Zora, Akari Asai, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. “CodeRAG-Bench: Can Retrieval Augment Code Generation?.” ArXiv e-prints arXiv-2406, 2024.

[14]. Bassamzadeh Nastaran and Chhaya Methani. “A Comparative Study of DSL Code Generation: Fine-Tuning vs. Optimized Retrieval Augmentation.” ArXiv preprint arXiv:2407.02742, 2024.

[15]. Yucan Guo, Zixuan Li, Xiaolong Jin, Yantao Liu, Yutao Zeng, Wenxuan Liu, Xiang Li, Pan Yang, Long Bai, Jiafeng Guo, and Xueqi Cheng. “Retrieval-Augmented Code Generation for Universal Information Extraction.” Natural Language Processing and Chinese Computing, Lecture Notes in Computer Science, Vol. 15360, pp. 30-42, 2025.

[16]. Xinran Yu, Chun Li, Minxue Pan, and Xuandong Li. “DroidCoder: Enhanced Android Code Completion with Context-Enriched Retrieval-Augmented Generation”. In 39th IEEE/ACM International Conference on Automated Software Engineering, pp. 681-693, 2024.

[17]. Tuan-Dung Bui, Duc-Thieu Luu-Van, Thanh-Phat Nguyen, Thu-Trang Nguyen, Son Nguyen, and Hieu Dinh Vo. “RAMBO: Enhancing RAG-based Repository-Level Method Body Completion.”ArXiv preprint arXiv:2409.15204, 2024.

[18]. Avik Dutta, Mukul Singh, Gust Verbruggen, Sumit Gulwani, and Vu Le. “RAR: Retrieval-augmented retrieval for code generation in low-resource languages”. In Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, pp. 21506–21515, 2024.

[19]. Xunzhu Tang, Liran Wang, Yonghui Liu, Linzheng Chai, Jian Yang, Zhoujun Li, Haoye Tian, Jacques Klein, and Tegawende F. Bissyande. “In-Context Code-Text Learning for Bimodal Software Engineering.” ArXiv preprint arXiv:2410.18107, 2024.

[20]. Hanzhen Lu and Zhongxin Liu. “Improving Retrieval-Augmented Code Comment Generation by Retrieving for Generation.” IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 350-362, 2024.

[21]. Yu-Chen Lin, Akhilesh Kumar, Norman Chang, Wenliang Zhang, Muhammad Zakir, Rucha Apte, Haiyang He, Chao Wang, Jyh-Shing Roger Jang. “Novel Preprocessing Technique for Data Embedding in Engineering Code Generation Using Large Language Model.” IEEE LLM Aided Design Workshop, pp. 1-5, 2024.

[22]. Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. “GraphCoder: Enhancing Repository-Level Code Completion via Coarse-to-fine Retrieval Based on Code Context Graph.” In 39th IEEE/ACM International Conference on Automated Software Engineering, pp. 570-581, 2024.

[23]. Junkai Chen, Xing Hu, Zhenhao Li, Cuiyun Gao, Xin Xia, and David Lo, “Code Search Is All You Need? Improving Code Suggestions with Code Search.” IEEE/ACM 46th International Conference on Software Engineering (ICSE), pp. 880-892, 2024.

[24]. Haochen Li, Xin Zhou, and Zhiqi Shen. 2024. “Rewriting the Code: A Simple Method for Large Language Model Augmented Code Search”. In Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics, Vol. 1: Long Papers, pp. 1371–1389, 2024.

[25]. Qianwen Gou, Yunwei Dong, Yujiao Wu, Qiao Ke. “RRGcode: Deep hierarchical search-based code generation.” Journal of Systems and Software, Vol. 211, pp. 111982, 2024

[26]. Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Shieh, and Wenmeng Zhou. “Codexgraph: Bridging large language models and code repositories via code graph databases.” ArXiv preprint arXiv:2408.03910, 2024.

[27]. Jiho Shin, Reem Aleithan, Hadi Hemmati, and Song Wang. “Retrieval-Augmented Test Generation: How Far Are We?.” ArXiv preprint arXiv:2409.12682, 2024.

[28]. Daneshvar Seyed Shayan, Yu Nong, Xu Yang, Shaowei Wang, and Haipeng Cai. “Exploring RAG-based Vulnerability Augmentation with LLMs” ArXiv abs/2408.04125, 2024.

[29]. Zhang Linghao, Hongyi Zhang, Chong Wang,

and Peng Liang. “RAG-Enhanced Commit Message Generation.” ArXiv preprint arXiv:2406.05514, 2024.

[30]. E. Halit Yilmaz, İ. Hakki Toroslu and Ö. Köksal, “A Comparative Study of Contemporary Learning Paradigms in Bug Report Priority Detection.” in IEEE Access, Vol. 12, pp. 126577-126586, 2024.

[31]. Harzevili, Nima Shiri, Mohammad Mahdi Mohajer, Jiho Shin, Moshi Wei, Gias Uddin, Jinqiu Yang, Junjie Wang, Song Wang, Zhen Ming, and Nachiappan Nagappan. “Checker Bug Detection and Repair in Deep Learning Libraries.” ArXiv preprint arXiv:2410.06440, 2024.

[32]. Rongwu Xu, Zehan Qi, Zhijiang Guo, Cunxiang Wang, Hongru Wang, Yue Zhang, and Wei Xu. “Knowledge Conflicts for LLMs: A Survey.” In Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, pp. 8541–856, 2024.

[33]. Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. “Retrieval-augmented generation for large language models: A survey.” ArXiv preprint arXiv:2312.10997, 2023.

[34]. Zhaochen Su, Jun Zhang, Xiaoye Qu, Tong Zhu, Yanshu Li, Jiashuo Sun, Juntao Li, Min Zhang, and Yu Cheng. “Conflictbank: A benchmark for evaluating the influence of knowledge conflicts in LLM.” ArXiv preprint arXiv:2408.12076, 2024.

[35]. Philip Gage. “A New Algorithm for Data Compression”. The C Users Journal archive, Vol. 12, pp. 23-38, 1994.

[36]. Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics, Vol. 1, pp. 7212–7225, 2022.

[37]. Arxiv http://arxiv.org.