Integrating Retrieval-Augmented Generation for Enhanced Code Reuse: A Comprehensive Framework for Efficient Software Development

Kai Wang[1, 2, *], Yujie Ding[1, 2, *], Shuai Jia[1, 2, ✉], Tianyi Ma[2], Yin Zhang[1], and Bin Cao[3]
[1]College of Computer Science and Technology, Zhejiang University, Hangzhou, China
[2]HiThink Research, Hithink RoyalFlush Information Network Co., Ltd., Hangzhou, China
[3]College of Computer Science, Zhejiang University of Technology, Hangzhou, China
{wangkai7,dingyujie2,jiashuai,matianyi}@myhexin.com, zhangyin98@zju.edu.cn, bincao@zjut.edu.cn

2024 IEEE Smart World Congress (SWC) | 979-8-3315-2086-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/SWC62898.2024.00205

***Abstract*—With the rapid development of ubiquitous computing, the demand for efficient software development is growing stronger. Code reuse is an effective means to enhance software development efficiency, significantly reducing the development cycle. Although existing large language models for code generation can assist developers in quickly writing code, they primarily aid in generating generic code and are unable to produce specific business code, making code reuse indispensable. In this paper, We propose a retrieval-augmented generation-based framework to streamline code reuse. This involves extracting reusable code units, generating natural language summaries, and matching queries with these summaries using search and reranking techniques. We conducted extensive experiments from both code search and code generation perspectives, the experimental results demonstrate that our approach significantly improves the efficiency and quality of code reuse.**

***Index Terms*—Code Reuse, Code Search, Code Generation**

I. Introduction

The rise of ubiquitous sensors, devices, networks, and information is creating a smart world where computational intelligence is embedded throughout the environment, providing reliable and relevant services [1]. This pervasive intelligence is transforming the computing landscape, enabling new applications and systems, and expanding the realm of possibilities. As ubiquitous computing grows, the need for efficient and rapid development of smart applications becomes critical. High-efficiency software development is essential to ensure applications are reliable and can fully leverage the intelligent infrastructure.

Code reuse [2] is a key technique for efficient software development, enabling the use of existing components to build new applications rather than developing everything from scratch [3] [4]. By reusing code, developers can reduce development time and costs, allowing them to focus on the unique aspects of their applications rather than reinventing common functionalities [5]. Additionally, reusing well-tested and proven code enhances the reliability and stability of new systems, crucial in environments where computational intelligence must operate seamlessly and consistently [6].

*The two authors contribute equally to this work

Despite the emergence of Large Language Models (LLMs) for code generation, such as CodeX [7], StarCoder [8], and Code-Llama [9], which have significantly enhanced developers' efficiency and accuracy in writing code, these models are only capable of generating general-purpose code. They struggle to produce code tailored to specific business contexts, such as specialized scenarios in finance, industry, and gaming. This limitation poses a significant constraint for developers [10]. Consequently, even with the advent of these large language models, code reuse technology remains indispensable in software development.

Traditional code reuse methods require developers to manually search through development documentation or directly search the source code, which is a highly time-consuming process. Therefore, quickly locating reusable code is crucial in code reuse. This task involves matching the corresponding code through natural language queries—a subtask of the code search problem. The simplest method is to compare the query with the text in the code, such as words in the project metadata [11], field and method signatures [12], or all identifiers in the code [13], [14]. However, due to the low correspondence between code language and natural language, the accuracy of these methods is low. An effective approach is to use word embeddings to project natural language and code language into the same vector space [15]. However, this requires a large amount of labeled data. Additionally, due to the variability of natural language, fully projecting code language and natural language into the same vector space is challenging. In summary, quickly locating reusable code is a pressing issue in code reuse. Due to these two issues, many developers prefer to rewrite code rather than reuse existing code, leading to repetitive work that significantly reduces development efficiency and prolongs the development cycle.

To address the challenges in code reuse mentioned above, we propose a retrieval-augmented generation-based framework for code reuse, named RAGCR, which effectively enhances the efficiency of code reuse, as illustrated in Figure 1. Initially, we extract reusable units from code repositories, considering functions as the primary components since they are the most common code functional blocks. Subsequently, the LLM

2993-396X/24/$31.00 ©2024 IEEE
DOI 10.1109/SWC62898.2024.00205

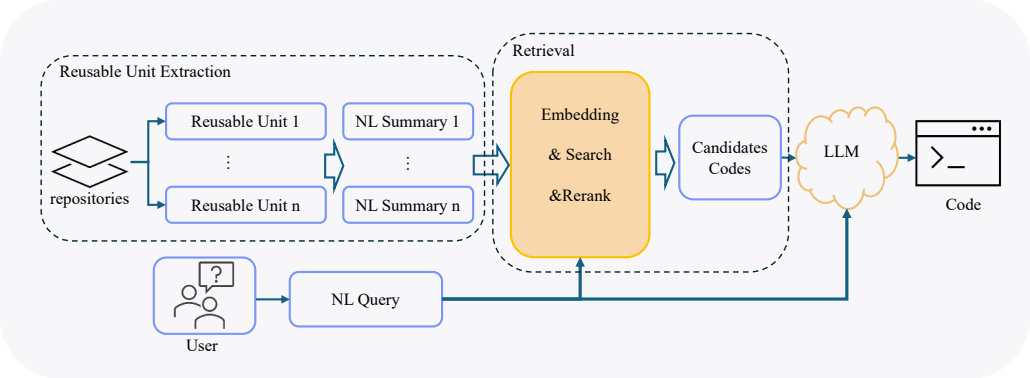

Fig. 1: The proposed retrieval-augmented generation-based framework for code reuse.

generates a natural language summary for each reusable unit, describing its code functionality. Next, we precisely match the natural language query with the summaries generated in the first step, using embedding, search, and re-ranking procedures to identify the best matching reusable unit candidates. To enhance the effectiveness of the rerank process, we train a rerank model with the code-related natural language dataset. In the final step, the selected reusable units and the user's query are input into the LLM to generate the final code that meets the requirements of user. This framework significantly reduces the manual effort and time required by developers in the code search and reuse process, substantially improving the efficiency of code reuse through efficient automated matching and generation techniques.

Finally, to validate the effectiveness of our framework, we conduct extensive experiments in two areas: code search and reusable code generation. The results demonstrate that our proposed method can effectively retrieve the corresponding reusable code from the repository based on the queries. Furthermore, the quality of the final code generated through our reuse framework significantly surpasses that generated by GPT4-o.

In brief, the main contributions of this paper are summarized as follows:

- We propose a novel code reuse framework that effectively enhances code reuse efficiency, improves the quality of generated code, and reduces development complexity.
- We propose extracting reusable units from code repositories and generating natural language summaries for these units, which can effectively match user queries.
- We fine-tuned the Embedding and Rerank model on code-related natural language, which improved the rerank performance.
- We conducted extensive experiments, and the results demonstrated that our method effectively enhances code search and reuse.

II. Related Work

In this section, we will present our related work in terms of code search, code reuse and code generation.

Code Search. With the volume of source code growing explosively, code search has become a routine task for developers to extract useful information from the existing codebase [16]. There are various ways to query code, such as natural language-based queries [17], queries based on existing programming languages [18], and custom query languages [19]. The work by Gu et al. [20] was the first to propose an end-to-end natural language to code search engine, embedding method names, API sequences, and method bodies through recurrent neural networks. Additionally, pretrained models have been applied to code search by fine-tuning the pretrained model to specific programming languages and queries [21].

Code Reuse. Code reuse involves leveraging existing code to aid in current software development, eliminating the need to create projects from scratch [22]. Research indicates that code reuse can enhance system quality, understandability, and maintainability [23]. Hironori et al [24] studied object-oriented software systems and introduced a method that extracts components from existing software into reusable JavaBean components. Mancoridis et al. [25] were the first to propose a search-based optimization approach to find optimal solutions based on module relationships in the source code. Rathee et al. [26] proposed a search-based optimization technique to identify reusable components. Kabir et al. [27] proposed a hierarchical clustering technique to identify components from the source code.

Code Generation. The rapid development of large language models has propelled advancements in the field of code generation, with the use of these large models to automate code generation tasks becoming a focal point. Several prominent code generation models have emerged, such as Codex [7], StarCoder [8], and Code-Llama [9]. These models can generate corresponding code based on developer instructions, significantly enhancing development efficiency. However, despite their capabilities, these powerful models can only generate generic code and cannot tailor the code to specific business functionalities. Hayati et al. [28] proposed to integrate information retrieval techniques into the code generation task. Parvez et al. [29] proposed REDCODER, which utilizes retrieval to augment the input of the generation model, aiming to maximize the utilization of supplementary

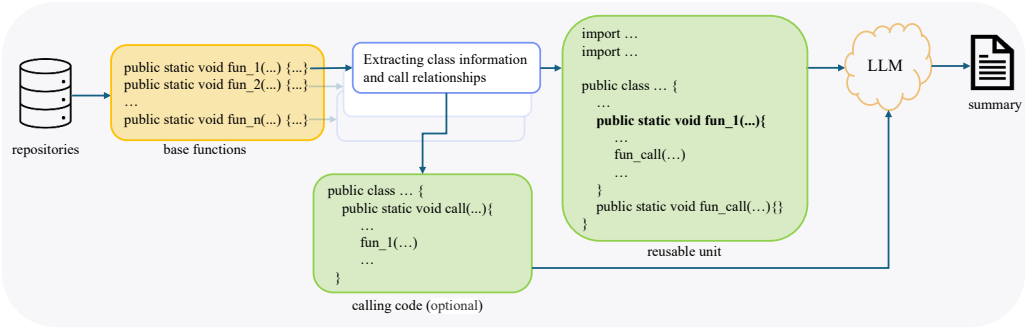

Fig. 2: Reusable unit extraction.

information. Zan et al. [30] introduced CodeGenAPI, which retrieves private APIs from documentation to enhance code generation.

III. Framework

In this section, we will introduce our proposed reuse framework, which comprises three components: reusable unit extraction, code retrieval, and code generation.

A. Reusable Unit Extraction

The first step in code reuse is identifying reusable code from the code repository. Hence, we need to define reusable units as the basis for subsequent code reuse. In this paper, we define a reusable unit extracted from the code repository, as illustrated in Figure 2. Firstly, considering that functions are the most common basic logical blocks in the code and each function has a specific functionality, we use functions as the primary components of the reusable units. For each extracted function, we also gather the class information to which the function belongs, including the variables used within the class and import information. For example, in the case of function 1 shown in the figure, if function 1 internally calls an external function, fun_call, the definition of fun_call should also be included in the reusable unit. In other words, for function fun_1, the reusable unit retains all related content within the class, ensuring the preservation of all information pertaining to fun_1, thereby enabling accurate reuse in the future. Additionally, we extract the code that calls fun_1 to illustrate its invocation context.

Considering that class file paths contain higher-level architecture names, which are often named by functionality and may assist in summarizing function features, we retain the file path of the function. Subsequently, we input the reusable unit, the calling code, and the file path of the class in which the function resides into the LLM to obtain a natural language summary of the reusable unit. Additionally, given that LLM outputs can be unstable, we use 8000 reusable unit-summary pairs as training data to fine-tune the LLM. This approach ensures that the LLM can accurately generate natural language summaries of reusable units according to the given rules, thereby enhancing the stability and usability of the outputs of LLM for this task.

B. Code Retrieval

During the code search phase, our objective is to identify the most similar reusable units from a large collection. This process involves three key steps: Embedding, Search, and Rerank. Initially, we convert the natural language descriptions of user queries and the summaries of reusable units into vector representations using BGE-embedding [31]. However, because user queries and reusable unit summaries are not ordinary natural language but rather specialized code-related language, we need to fine-tune the embedding model to better handle code-specific language and improve embedding accuracy. To achieve this, we collect a substantial number of code samples from open-source repositories and organized these code functionality summaries as shown in Figure 2. Subsequently, we augmented these summary data using a LLM, enhancing the diversity and comprehensiveness of the dataset. Each functionality summary corresponds to both positive and negative samples, which helps the model better understand and differentiate the similarities between different code functionalities. Subsequently, we optimized the embedding model using the following loss function.

$$\min . \sum_{(p,q)} -\log \frac{e^{\langle \mathbf{e}_p, \mathbf{e}_q \rangle / \tau}}{e^{\langle \mathbf{e}_p, \mathbf{e}_q \rangle / \tau} + \sum_{Q'} e^{\langle \mathbf{e}_p, \mathbf{e}_{q'} \rangle / \tau}}.$$

where p and q are the paired texts, $q' \in Q'$ is a negative sample. This loss function aims to maximize the similarity between positive sample pairs while minimizing the similarity between negative sample pairs, thereby enhancing the performance of model in practical applications. Through this optimization process, we expect the embedding model to more accurately capture the subtle differences in code functionality descriptions, ultimately improving the effectiveness of code search.

In the search phase, we use the common cosine similarity to calculate the similarity between the query vectors and the reusable unit vectors. The vector-based retrieval process using embeddings can efficiently retrieve highly relevant reusable units through a certain degree of semantic similarity. However, due to the inherent complexity and ambiguity of semantics, as well as potential noise in high-dimensional vector similarity matching, the retrieval might recall some low-relevance candidates. Therefore, we introduce a rerank model that considers

 # role:
You are an experienced Java programmer. To facilitate quick code lookup, you need to generate a summary for the given code, describing its specific functionality.

 # rules:
- Describe its specific functionality. Do not directly explain each line of code.
- Use professional programming language to articulate the description.
- Ensure logical consistency and use precise terminology.
- Keep the summary concise and to the point.

 # output format
Generate a summary only; do not output additional information.

Fig. 3: Conclusion prompt.

semantic relationships from contextual information, further optimizing the results based on the initial vector retrieval. Similar to the embedding process, we also construct training data related to code language for fine-tuning the rerank model. Unlike the embedding model, the rerank model is a cross-encoder model, utilizing a cross-entropy loss function. This model evaluates the relevance between pairs of input texts and cannot generate embeddings, making it particularly suited for relevance queries among small samples. By fine-tuning the BGE-rerank model, we significantly improved the accuracy of matches.

C. Code generation

During the code generation phase, it is crucial to first construct suitable and precise prompts to ensure the model comprehends our requirements. Subsequently, after identifying potential reusable code units, we input these units along with the specific query into the large language model. By doing so, the model can integrate the existing code units with the query content, generating the final efficient and reusable code segments. This approach enhances both development efficiency and code quality.

IV. Experiment

In this section, we will discuss our experimental results. We conduct experiments focusing on two aspects: code search and code reuse generation. Additionally, we compare our results with those of the current best-performing large language models to demonstrate the effectiveness of our approach.

A. Code Search

Dataset. We conduct experiments on three datasets, namely CSN [32], AdvTest [33] and CosQA [34]. The CSN dataset is constructed from the CodeSearchNet dataset across six languages, filtered manually to remove low-quality queries. It includes 8,122 Go entries, 10,955 Java entries, 3,291 JavaScript entries, 14,014 PHP entries, 14,918 Python entries, and 1,261 Ruby entries. The AdvTest dataset contains 19,000 entries of Python data from CSN, with standardized Python functions and variable names to better test the model's understanding and generalization abilities. The CosQA dataset consists of 20,604 search logs from Microsoft Bing, annotated by at least three human annotators for each log.

Baselines. For code search, we compare our method with following approaches:

TABLE I: Experimental result of code search

Model	CosQA	AdvTest	CSN
RoBERTa	60.3	18.3	61.7
CodeBERT	65.7	27.2	69.3
GraphCodeBERT	68.4	35.2	71.3
SYNCOBERT	-	38.3	74
PLBART	65.0	34.7	68.5
CodeT5-base	67.8	39.3	71.5
UniXcoder	70.1	41.3	74.4
RAGCR	**83.1**	**92.2**	**83.4**

- RoBERTa [35], which is an encoder-only pre-trained model on text corpus with MLM.
- CodeBERT [36], which pre-trained on NL-PL pairs using both MLM and replaced token detection.
- GraphCodeBERT [32], that leverages data flow to enhance code representation.
- SYNCOBERT [37], which incorporates AST by edge prediction and contrastive learning.
- PLBART [38], which is based on BART and pre-trained on 470M Python and 210M Java functions, and 47M NL posts from StackOverflow using denoising objective.
- CodeT5-base [39], adapted from T5, considers the crucial token type information from identifiers and allows multi-task learning on downstream tasks.
- UniXcoder [40], which utilizes mask attention matrices with prefix adapters to control the behavior of the model and leverages cross-modal contents like AST and code comment to enhance code representation.

Experimental setting. First, we construct a partial code-summary instruction dataset and fine-tune a 7B model. We initially use this fine-tuned model to summarize the code in the test datasets. Next, we employ an embedding model to generate semantic vectors for the queries and code summaries. Finally, we perform similarity matching using cosine similarity to obtain code similarity rankings. The summary prompt we used is as shown as Figure 3.

Evaluation metrics. We follow the CSN official evaluation metric to calculate the Mean Reciprocal Rank (MRR) for each pair of test data over a fixed set of 999 distractor codes.

Experimental result. The experimental results of the code search are shown in Table I. These results demonstrate that our method significantly outperforms other models in terms of Mean Reciprocal Rank (MRR) on the CosQA, AdvTest, and CSN datasets. On the CosQA dataset, our method achieves an MRR of 83.1, markedly higher than the second-best UniXcoder's 70.1. On the AdvTest dataset, our method achieves an MRR of 92.2, more than double that of the second-best UniXcoder, which scored 41.3. Similarly, on the CSN dataset, our method leads with an MRR of 83.4, outperforming other models. These results indicate that our approach offers significant advantages in accuracy, relevance, and reliability in the code search task, particularly in complex and challenging test data, showcasing exceptional performance and robust generalization capabilities.

The experimental results on various languages in the CSN

TABLE II: Results of code search over six programming languages on CSN

Model	Ruby	Javascript	Go	Python	Java	Php	Overall
RoBERTa	58.7	51.7	85.0	58.7	59.9	56.0	61.7
CodeBERT	67.9	62.0	88.2	67.2	67.6	62.8	69.3
GraphCodeBERT	70.3	64.4	89.7	69.2	69.1	64.9	71.3
SYNCOBERT	72.2	67.7	91.3	72.4	72.3	67.8	74.0
PLBART	67.5	61.6	88.7	66.3	66.3	61.1	68.5
CodeT5-base	71.9	65.5	88.8	69.8	68.6	64.5	71.5
UniXcoder	74.0	68.4	91.5	72.0	72.6	67.6	74.4
RAGCR	**80.1**	**71.5**	**94.0**	**93.7**	**80.2**	**81.0**	**83.4**

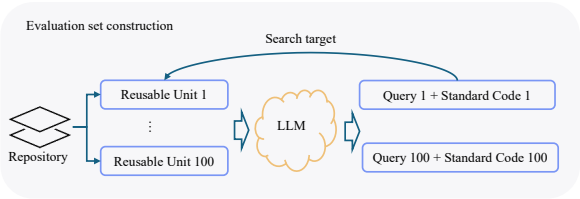

Fig. 4: Evaluation set construction.

dataset are shown in Table II. Our method demonstrates outstanding performance in terms of MRR across all six programming languages in the CSN dataset. Specifically, it achieves an MRR of 80.1 in Ruby, significantly outperforming other models; 71.5 in JavaScript; an impressive 94.0 in Go, far surpassing other models; an exceptional 93.7 in Python; and high scores of 80.2 and 81.0 in Java and PHP, respectively. Overall, our method achieves an MRR of 83.4, consistently outperforming other comparative models, showcasing exceptional cross-language code search capabilities and robust generalization performance.

B. Code Reuse

In this section, we will discuss the results of our code reuse experiments. To demonstrate the effectiveness of our approach, we conduct experiments using real open-source code repositories and compare our method against the most powerful LLMs currently available. The experimental results indicate the effectiveness of our proposed code reuse method.

Evaluation set construction. To validate the effectiveness of code reuse, we obtain ten code repositories from open-source git websites for this experiment, all repositories can be found on https://gitee.com. Using these ten repositories, we construct a code reuse evaluation set for subsequent experiments, As shown in Figure 4. Firstly, we adopt the reusable unit extraction method proposed in this paper to extract the fundamental reusable units from multiple code repositories. During this process, we filter out the reusable units containing only a small amount of code. Next, we randomly select 100 samples from the reusable units of each repository and use an LLM to modify the code of these units while generating corresponding instructions. These instructions served as query statements, with the corresponding code as the standard codes. These instructions and code comprise the query set, which forms the basis for all subsequent experiments. The ideal goal of code reuse is to retrieve Reusable Unit 1 using Query 1 and then reuse the code of Reusable Unit 1 to generate code that approximates Standard Code 1.

TABLE III: The code retrieval accuracy on real code repositories

Repositories	Top-3 Acc	Top-3 Acc with Rerank
LinkWeChat	0.67	0.72
REBUILD	0.99	1.00
redragon-erp	0.63	0.67
smart-web2	0.81	0.82
aeaicrm	0.79	0.80
psi-master	1.00	1.00
yuqing	0.99	1.00
BMS	0.88	0.89
zhaoxinpms	0.80	0.86
Car-eye-server	0.68	0.78

Evaluation method. To retrieve and reuse reusable units to generate the desired code, we employ two evaluation metrics: code retrieval accuracy and win rate. We feed the code reuse results from RAGCR and other models into the LLM, enabling it to determine the winner and calculate the win rate.

Experimental setting. In this experiment, we set the number of reusable code candidates to 3. During the reusable code summarization stage, we use the 7B model for summarization. Finally, to generate the reusable code, we employ the Qwen2 72B model, and the generated results were evaluated using GPT4-o.

Baselines. We use Qwen2 72B, Llama3 70B, DeepSeek Coder 236B, and GPT4-o as baselines for the experiment.

Experimental result. Since the previous section has validated the effectiveness of our proposed code retrieval method, this section focuses solely on experimenting with the rerank effect in practical scenarios. The experimental results are shown in Table III. From the table, we can observe that the rerank method improve the Top-3 accuracy in most repositories. Notably, in the LinkWeChat and Car-eye-server repositories, the improvement reaches 5% and 10%, respectively. In the REBUILD, psi-master, and yuqing repositories, although the accuracy was already close to 100% without rerank, the method still yield slight improvements. Overall, rerank effectively enhances the performance of the code retrieval system, particularly in repositories with lower initial accuracy.

Table IV presents the comparison results between our method and other models. We use GPT4-o to evaluate which of the two input code snippets is closer to the standard code. The experimental results demonstrate that our proposed method outperforms the comparison methods in almost all test repositories. This is because other models generate code directly based on queries, whereas our method employs code reuse to generate the queried code. These results indicate that our code reuse approach is highly effective. Furthermore, even in repositories with lower code retrieval accuracy, such as LinkWeChat and redragon-erp, our win rate remains nearly one hundred percent. This suggests that even if our method retrieves code that does not correspond exactly to the query, the retrieved results still assist the model in generating useful

TABLE IV: Win rates of code reuse methods compared to other models

vs. Model	repositories									
	LinkWeChat	REBUILD	redragon-erp	smart-web2	aeaicrm	psi-master	yuqing	BMS	zhaoxinpms	Car-eye-server
vs. GPT4-o	0.99	0.99	1	1	0.99	1	0.99	1	1	0.98
vs. DeepSeek Coder 236B	0.99	1	1	0.99	1	1	1	1	1	0.99
vs. Qwen2 72B	1	1	0.97	0.99	1	1	0.98	0.99	1	0.99
vs. Llama3 70B	0.99	0.99	0.97	1	1	1	0.99	1	0.99	0.98

TABLE V: Win rates after adjusting code order in prompt

vs. Model	repositories									
	LinkWeChat	REBUILD	redragon-erp	smart-web2	aeaicrm	psi-master	yuqing	BMS	zhaoxinpms	Car-eye-server
vs. GPT4-o	0.6	0.75	0.7	0.57	0.77	0.67	0.63	0.63	0.72	0.55
vs. DeepSeek Coder 236B	0.7	0.79	0.61	0.6	0.79	0.78	0.66	0.58	0.74	0.67
vs. Qwen2 72B	0.64	0.75	0.56	0.53	0.64	0.64	0.5	0.54	0.68	0.61
vs. Llama3 70B	0.64	0.76	0.61	0.61	0.73	0.69	0.6	0.74	0.72	0.55

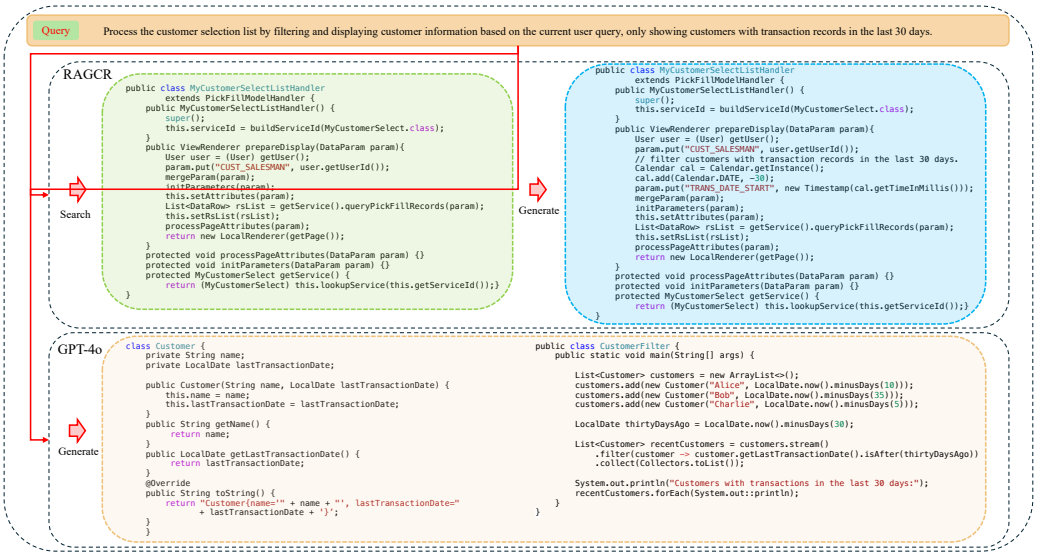

Fig. 5: Comparison of results between RAGCR and GPT-4o on the example.

code.

Simultaneously, when allowing GPT4-o to evaluate the results, the sequence of code in the prompt may affect the evaluation outcome, as GPT4-o tends to focus on the latter part of the input prompt. To eliminate the influence of code order, we conduct an ablation experiment where the code generated by our method is placed at the beginning of the input prompt, while the results generated by other models are placed later in the prompt. The results, shown in Table V, indicate that adjusting the code sequence does indeed impact the evaluation of GPT4-o. However, even when our method's generated code was placed at the beginning of the input prompt, it still outperforms other methods. This demonstrates the high effectiveness of the code reuse approach we proposed.

Finally, we present an example comparing code generation through code reuse and direct use of GPT4-o, as shown in Figure 5. This query comes from the validation set aeaicrm [41]. The top side of the figure represents the code generated through the code reuse process, while the bottom side shows the code generated directly by GPT4-o. It can be seen that the code generated through the code reuse process not only reuses the retrieved code but also accurately generates the code corresponding to the query. In contrast, GPT4-o fails to generate the code corresponding to the query.

V. Conclusion

The advancement of ubiquitous computing has accelerated the development of software engineering, making rapid software development an urgent priority in contemporary practices. One effective approach to expedite development is code reuse, which involves utilizing existing code for new projects. To address the challenges associated with code reuse, we proposes retrieval-augmented generation-based framework for code reuse that involves the extraction of reusable units, natural language summarization of code, vector search and re-rank. We conduct sufficient experiment, the results demonstrate that our proposed method can effectively retrieve the corresponding reusable code from the repository based on queries. Furthermore, the quality of the final code generated through our reuse framework significantly surpasses that generated by GPT4-o.

REFERENCES

[1] K. Lyytinen and Y. Yoo, "Ubiquitous computing," *Communications of the ACM*, vol. 45, no. 12, pp. 63–96, 2002.

[2] S. Haefliger, G. Von Krogh, and S. Spaeth, "Code reuse in open source software," *Management science*, vol. 54, no. 1, pp. 180–193, 2008.

[3] A. Mockus, "Large-scale code reuse in open source software," in *First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07: ICSE Workshops 2007)*. IEEE, 2007, pp. 7–7.

[4] J. Krüger and T. Berger, "An empirical analysis of the costs of clone-and platform-oriented software reuse," in *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2020, pp. 432–444.

[5] M. Sojer and J. Henkel, "Code reuse in open source software development: Quantitative evidence, drivers, and impediments," *Journal of the Association for Information Systems*, vol. 11, no. 12, pp. 868–901, 2010.

[6] T. Ishihara, K. Hotta, Y. Higo, and S. Kusumoto, "Reusing reused code," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 457–461.

[7] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[8] R. Li, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, L. Jia, J. Chim, Q. Liu *et al.*, "Starcoder: may the source be with you!" *Transactions on Machine Learning Research*.

[9] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[10] X. Gu, M. Chen, Y. Lin, Y. Hu, H. Zhang, C. Wan, Z. Wei, Y. Xu, and J. Wang, "On the effectiveness of large language models in domain-specific code generation," *ACM Transactions on Software Engineering and Methodology*, 2023.

[11] C. Mcmillan, D. Poshyvanyk, M. Grechanik, Q. Xie, and C. Fu, "Portfolio: Searching for relevant functions and their usages in millions of lines of code," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 4, pp. 1–30, 2013.

[12] E. Hill, L. Pollock, and K. Vijay-Shanker, "Improving source code search with natural language phrasal representations of method signatures," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 524–527.

[13] M. Lu, X. Sun, S. Wang, D. Lo, and Y. Duan, "Query expansion via wordnet for effective code search," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 545–549.

[14] L. Martie and A. Van der Hoek, "Sameness: An experiment in code search," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 76–87.

[15] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.

[16] C. Sadowski, K. T. Stolee, and S. Elbaum, "How developers search for code: a case study," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 2015, pp. 191–201.

[17] J. Gu, Z. Chen, and M. Monperrus, "Multimodal representation for neural code search," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 483–494.

[18] Y. Fujiwara, N. Yoshida, E. Choi, and K. Inoue, "Code-to-code search based on deep neural network and code mutation," in *2019 IEEE 13th International Workshop on Software Clones (IWSC)*. IEEE, 2019, pp. 1–7.

[19] J. Lawall, D. Palinski, L. Gnirke, and G. Muller, "Fast and precise retrieval of forward and back porting information for linux device drivers," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 15–26.

[20] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 933–944.

[21] P. Salza, C. Schwizer, J. Gu, and H. C. Gall, "On the effectiveness of transfer learning for code search," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1804–1822, 2022.

[22] L. Heinemann, F. Deissenboeck, M. Gleirscher, B. Hummel, and M. Irlbeck, "On the extent and nature of software reuse in open source java projects," in *Top Productivity through Software Reuse: 12th International Conference on Software Reuse, ICSR 2011, Pohang, South Korea, June 13-17, 2011. Proceedings 12*. Springer, 2011, pp. 207–222.

[23] M. D. Papamichail, T. Diamantopoulos, and A. L. Symeonidis, "Measuring the reusability of software components using static analysis metrics and reuse rate information," *Journal of Systems and Software*, vol. 158, p. 110423, 2019.

[24] H. Washizaki and Y. Fukazawa, "A technique for automatic component extraction from object-oriented programs by refactoring," *Science of Computer programming*, vol. 56, no. 1-2, pp. 99–116, 2005.

[25] A. Marx, F. Beck, and S. Diehl, "Computer-aided extraction of software components," in *2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 183–192.

[26] A. Rathee and J. K. Chhabra, "A multi-objective search based approach to identify reusable software components," *Journal of Computer Languages*, vol. 52, pp. 26–43, 2019.

[27] S. Kebir, A.-D. Seriai, S. Chardigny, and A. Chaoui, "Quality-centric approach for software component identification from object-oriented code," in *2012 Joint working IEEE/IFIP conference on software architecture and european conference on software architecture*. IEEE, 2012, pp. 181–190.

[28] S. A. Hayati, R. Olivier, P. Avvaru, P. Yin, A. Tomasic, and G. Neubig, "Retrieval-based neural code generation," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018, pp. 925–930.

[29] M. R. Parvez, W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Retrieval augmented code generation and summarization," in *Findings of the Association for Computational Linguistics: EMNLP 2021*, 2021, pp. 2719–2734.

[30] D. Zan, B. Chen, Z. Lin, B. Guan, W. Yongji, and J.-G. Lou, "When language model meets private library," in *Findings of the Association for Computational Linguistics: EMNLP 2022*, 2022, pp. 277–288.

[31] S. Xiao, Z. Liu, P. Zhang, and N. Muennighoff, "C-pack: Packaged resources to advance general chinese embedding," 2023.

[32] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, L. Shujie, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," in *International Conference on Learning Representations*.

[33] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

[34] J. Huang, D. Tang, L. Shou, M. Gong, K. Xu, D. Jiang, M. Zhou, and N. Duan, "Cosqa: 20,000+ web queries for code search and question answering," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 2021, pp. 5690–5700.

[35] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[36] 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," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[37] X. Wang, Y. Wang, F. Mi, P. Zhou, Y. Wan, X. Liu, L. Li, H. Wu, J. Liu, and X. Jiang, "Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation," *arXiv preprint arXiv:2108.04556*, 2021.

[38] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conference of the NAACL: Human Language Technologies*, 2021, pp. 2655–2668.

[39] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[40] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2022, pp. 7212–7225.

[41] https://gitee.com/agileai/aeaicrm.