Hierarchical Code Abstraction via Unified GraphRAG on Multi-Grained Graphs

1st Zhongtian Bao
Key Laboratory of DISSec
College of Computer Science
Nankai University
Tianjin, China
zhongtianbao@mail.nankai.edu.cn
ORCID: 0000-0001-6618-5325

2nd Wenjian Ding
Key Laboratory of DISSec
College of Computer Science
Nankai University
Tianjin, China
wjding@mail.nankai.edu.cn
ORCID: 0009-0006-7498-5289

3rd Jun Wang
Ludong University
Shandong, China
junwang@mail.nankai.edu.cn
ORCID: 0000-0001-8932-6661

4th Yao Zhang
School of Statistics and
Data Science, LPMC,
KLMDASR & AAIS,
Nankai University
Tianjin, China
yaozhang@nankai.edu.cn
ORCID: 0009-0003-3071-0513

5th Zhenglu Yang
Key Laboratory of DISSec
College of Computer Science
Nankai University
Tianjin, China
yangzl@nankai.edu.cn
ORCID: 0000-0001-9528-965X

6th Zhe Sun
Graduate School of Medicine
Faculty of Health Data Science
Juntendo University
Japan
z.sun.kc@juntendo.ac.jp
ORCID: 0000-0002-6531-0769

7th Andrzej Cichocki
Systems Research Institute
Polish Academy of Sciences and
UMK
Poland
cichockiand@gmail.com
ORCID: 0000-0002-8364-7226

***Abstract*—Code summarization plays a crucial role in helping developers quickly grasp the purpose and behavior of source code. However, the majority of existing approaches concentrate solely on the function level, overlooking the richer contextual signals and hierarchical relationships that naturally exist within large software repositories. To address these shortcomings, we present a new hierarchical code summarization framework built upon GraphRAG. Our key observation is that the community structures revealed by GraphRAG inherently align with the layered organization of real-world software systems—for instance, functions forming classes, and classes composing files. Leveraging this insight, we construct a multi-level code summary graph and dynamically retrieve the most relevant subgraph to supply a large language model with structured and contextually coherent information. This design alleviates the context fragmentation commonly seen in traditional retrieval-augmented generation pipelines. Comprehensive experiments demonstrate that our framework markedly improves summary quality, especially when applied to complex or large-scale codebases.**

***Index Terms*—Code Summarization, GraphRAG, Large Language Model**

Corresponding author: Yao Zhang (yaozhang@nankai.edu.cn), Zhenglu Yang (yangzl@nankai.edu.cn)

I. Introduction

The task of code summarization, which automatically produces succinct natural language descriptions for source code segments, serves as a foundational component in software maintenance, program comprehension, and collaborative development workflows [1]. Research in this area has undergone substantial progression: initial studies relied on information-retrieval techniques and manually designed templates [2], [3]; subsequent work adopted deep learning models, particularly recurrent neural networks (RNNs) [4]; and, more recently, large-scale pre-trained models such as CodeT5 [5], CodeLlama [6], and DeepSeek-Coder [7] have become prominent. Nevertheless, a notable limitation of current research is that the majority of existing methods focus predominantly on the function- or class-level granularity [8]–[10]. While these approaches offer valuable insights into isolated program components, they overlook the broader logical structure, design patterns, and architectural principles that arise within large-scale software systems.

The rapid development of Large Language Models (LLMs) has significantly advanced the field of code summarization,

enabling the production of summaries that exhibit greater fluency, contextual relevance, and logical consistency [11]. Recent LLM-driven research in this area has largely progressed along three major directions: (1) fine-tuning pre-trained models on curated code–summary corpora [5]–[7], (2) devising prompt engineering techniques to better steer general-purpose LLMs toward code comprehension tasks [12], and (3) utilizing Retrieval-Augmented Generation (RAG) frameworks, which incorporate semantically relevant code fragments as supplementary context during summary generation [13].

Although RAG has proven effective in reducing hallucination in large language models, conventional vector-based RAG methods are ill-suited to the inherently structured and highly interdependent nature of source code. Because these approaches segment semantically related components—often distributed across multiple files—into isolated textual chunks, they inevitably introduce context fragmentation, thereby limiting the model's ability to capture the global dependencies crucial for comprehensive code understanding. While prior work has explored structural representations such as Abstract Syntax Trees (ASTs) combined with Graph Neural Networks (GNNs) [14], [15], such methods remain predominantly focused on the function-level granularity.

In contrast, the present study aims to develop a unified framework capable of performing code summarization across multiple hierarchical levels. Recent efforts in this direction [13], [16] have shown promising results; however, they generally depend on rudimentary aggregation strategies that fall short of modeling the rich semantic dependencies that span functions, classes, files, and entire repositories.

To address these shortcomings, we introduce a retrieval-centric hierarchical summarization framework grounded in GraphRAG [17]. The core idea behind our method is that the hierarchical community structure produced by GraphRAG aligns naturally with the multi-level organization of software systems. This formulation enables the retrieval and integration of a structured, multi-level subgraph as contextual input to the LLM, thereby effectively alleviating the fragmentation challenges that characterize traditional RAG-based pipelines. The main contributions of this work are summarized as follows.

- We investigate the problem of code summarization in large-scale software repositories, with particular emphasis on alleviating context fragmentation caused by existing methods that fail to model the intricate semantic dependencies spanning multiple structural levels.
- We introduce a hierarchical summarization framework grounded in the GraphRAG community detection paradigm. It adaptively consolidates semantic information to generate coherent summaries across varying granularities from functions and classes to files and full repositories.
- We perform comprehensive evaluations on several benchmark datasets, and the empirical results indicate that our framework achieves significant improvements over state-of-the-art baselines in both summarization quality and generation efficiency.

II. Related Work

Code summarization has evolved from early information retrieval and handcrafted templates [2], [3] to deep learning techniques employing recurrent neural networks [4]. The rapid advancement of LLMs [11] has further enhanced summary fluency, with research primarily focusing on fine-tuning pre-trained models like CodeT5 [18], CodeLlama [19], and DeepSeek-Coder [7]. Other strategies include prompt engineering [12] and Retrieval-Augmented Generation [13] to provide auxiliary context.

Despite these advances, a persistent limitation is that most methods operate primarily at the function or class level [8]–[10], neglecting broader architectural dependencies. While some studies incorporate structural representations like Abstract Syntax Trees via Graph Neural Networks [14], [15], their modeling remains largely confined to individual program units. Recent attempts toward multi-level summarization [13], [16] often rely on simplistic aggregation mechanisms that suffer from context fragmentation. This work builds upon the hierarchical community structure of GraphRAG [17] to bridge the gap between local logic and global project structure. Following standard practices, performance is evaluated using ROUGE [20], BERTScore [21], and LLM-as-a-Judge frameworks [22].

III. Our Framework

This section provides a detailed description of our proposed hierarchical code summarization framework, Code-GraphRAG, as depicted in Figure 1. The objective of Code-GraphRAG is to generate multi-granularity code summaries by constructing a comprehensive code knowledge base that is rich in structural information, while leveraging the advanced GraphRAG technique to deliver multi-dimensional context to LLMs. Our methodology is organized into three primary stages: (1) the construction and optimization of the GraphRAG knowledge base, (2) the graph-based retrieval of relevant code summaries, and (3) the final generation of the target code summary.

A. Building the Code Summary GraphRAG

Traditional RAG methods rely on flat vector databases, a structure that inadequately captures the inherent hierarchy and complex dependencies within code. To address this issue, we construct a retrieval-augmented knowledge base using GraphRAG, whose content consists of code summaries. This approach allows us to explicitly model various code entities, including functions, classes, files, and projects, and their rich structural and semantic relationships.

Formulation We define GraphRAG as a property graph G=(V,E,ψ_V,ψ_E), where V is the set of nodes incorporating code entities at different granularities, i.e., $V = V_{func} \cup V_{class} \cup V_{file} \cup V_{proj}$ which represent the node sets for functions, classes, files, and projects, respectively. Each node in the graph corresponds to an individual source code entity.

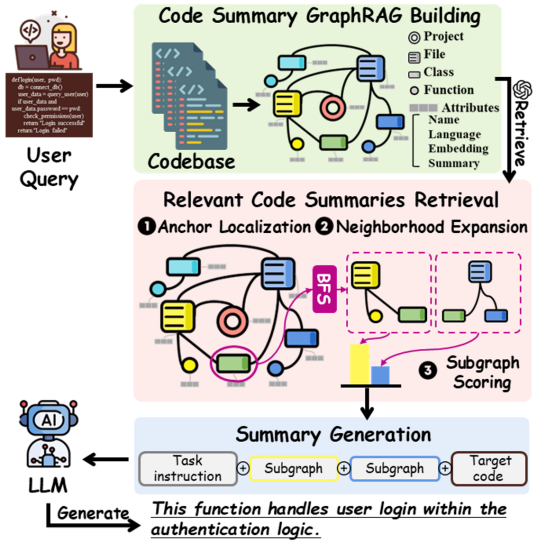

Fig. 1. An overview of Code-GraphRAG that contains three main components: building of code summary GraphRAG (Section II-A), retrieval of relevant summaries (Section II-B), and target code summary generation (Section II-C).

Node attributes ψ_V include source code text, entity name, programming language, code embedding vector, and the ground-truth summary obtained from the dataset.

E is the set of edges representing the relations between nodes, with attributes ψ_E that can include information such as weights. These edges model the critical connections within the codebase by capturing several kinds of interactions. These include hierarchical containment (e.g., a project containing files, which in turn contain classes and functions), functional dependencies (e.g., method invocation), and object-oriented relationships (e.g., class inheritance or implementation). In a nutshell, these connections are derived from static analysis of the source code to map the software's architecture.

Construction. This process begins with parsing the source code to identify all code entities and create corresponding nodes. Subsequently, based on the parsed code structure (such as call graphs, class diagrams), structural edges are established between nodes to complete the initial construction of the knowledge base. This knowledge base is then stored in a high-performance graph database, e.g., Neo4j.

Optimization and Completion. The initially constructed GraphRAG may suffer from issues such as sparse information and missing relationships. To address this, we have designed optimization and completion mechanisms to enhance the density and retrieval effectiveness of the knowledge base.

1) *Summary Completion.* For the nodes missing human-written summaries (especially high-level class or file nodes), we use a pre-trained code summarization model to generate a pseudo summary. This pseudo summary is marked with special tags and stored as an attribute of the node.

2) *Semantic Relationship Enhancement.* To capture non-explicit logical connections between codes, we introduce *semantically similar* edges. We employ advanced code representation models, i.e., GraphCodeBERT, to encode the source code c_i of each code node v_i into high-dimensional vectors $emb(c_i)$. Then, we measure the semantic relevance of the pairwise nodes c_i and c_j as follows:

$$sim(c_i, c_j) = \frac{emb(c_i) \cdot emb(c_j)}{|emb(c_i)||emb(c_j)|}. \quad (1)$$

A *semantically similar* edge is added to connect them iff $sim(c_i, c_j) > \tau$. This strategy can significantly enrich the connectivity of the graph, enabling the retrieval module to go beyond direct structural dependencies and discover code snippets that are functionally similar yet lack direct associations.

B. Retrieval of Relevant Code Summaries

When receiving a new target code c_q for summarization, this stage aims to precisely extract the most relevant context subgraph S_G from the GraphRAG. This process follows a strategy of the anchor localization, the neighborhood expansion, and the subgraph scoring.

Anchor Localization. First, the target code c_q is encoded into a query vector $emb(c_q)$. Using an efficient nearest neighbor search, the top-k nodes in the knowledge base that are most semantically similar to $emb(c_q)$ are retrieved. These nodes are referred to as anchors nodes, i.e., V_{anchor}.

Neighborhood Expansion. Centered on the anchor nodes, a Breadth-First Search (BFS) is performed on the graph to collect candidate subgraphs with rich contextual information.

Subgraph Scoring. To select the optimal subgraph from multiple candidate subgraphs, we have designed a composite scoring function. This function takes into account not only the subgraph's semantic relevance to c_q but also its inherent structural information. Given a candidate subgraph $S_G(V_S, E_S)$, its final score is assessed based on its semantic relevance score and structural cohesion score:

$$Score(S_G|c_q) = \frac{1}{|V_{anchor}|} \sum_{v_i \in V_{anchor}} sim(c_i, c_q) + \frac{1}{|V_S|} \cdot \frac{\sum_{v_i, v_j \in V_S} A_{ij} \cdot w_{ij}}{\sum_{v_k \in V} deg(v_k)}. \quad (2)$$

The first part of Eq. (2) calculates the average semantic similarity between the query code and the code of all anchor nodes. The second part measures the internal structural cohesion of the subgraph. A_{ij} is the element of the adjacency matrix of the subgraph S_G, which is equal to 1 if nodes v_i and v_j are connected in the subgraph, and 0 otherwise. w_{ij} is the weight of the edge. $deg(v_k)$ is the degree of node v_k in the entire graph, used as a normalization factor.

C. Target Code Summary Generation

This step leverages the retrieved optimal context subgraph S_G^* to guide the LLM in generating the final summary. We compose a complex prompt by combining the task instruction, the serialized context, and the target code. The serialization of the context preserves the structural information of the graph.

To enable the LLM to better understand and utilize this structured context, we define a context contribution-weighted prompt construction mechanism. The final input to the LLM,

P, consists of multiple components, where the importance of each snippet is determined by its score from the retrieval process:

$$P = Instruct_{task} \oplus \left(\bigoplus_{v_i \in V_{S_G^*}} w_i \cdot Serialize(v_i) \right) \oplus c_q, \quad (3)$$

where $\oplus$ is the text concatenation operation. $Instruct_{task}$ is the task instruction, e.g., "Please generate a summary for the following code". $Serialize(v_i)$ is a function that serializes node v_i and its summary information into text. w_i is the weight of node v_i in the context, which can be calculated based on the node's similarity to the query code or its centrality within the subgraph (e.g., PageRank score), where $w_i \propto sim(c_i, c_q)$. This means context information more relevant to the target code will occupy a more prominent position in the prompt.

We send the constructed weighted prompt to an LLM, i.e., GPT-4. When processing this prompt, the LLM can not only see the relevant code and summary examples, but also implicitly perceive which contextual information is more worthy of attention. This mechanism enables the model to integrate information more intelligently, generating a summary that is more accurate and insightful compared to cases with no context or simple context.

IV. Experimental Evaluation

A. Dataset

We conduct the experimental evaluation on the benchmark FILE-CS dataset [23] and CodeSearchNet dataset [24]. The FILE-CS dataset is a file-level code summarization dataset for the Python programming language. It comprises 98,236 pairs of source code files and their corresponding summaries. The CodeSearchNet dataset encompasses six programming languages: Go, Java, JavaScript, PHP, Python, and Ruby. For our experiments, we restrict the focus to the Python subset and randomly sampled 1,000 instances from the training split.

B. Implementations Details

Our baseline RAG models first use ASTs to decompose file-level code into function-level representations. Then, pre-trained models, such as CodeT5 [18] and GraphCodeBERT [25], are used to generate embeddings from the last hidden states of these functions, aggregated via mean pooling. These embeddings are indexed using the Faiss library. During testing, we query the Faiss index to retrieve the top-3 most similar functions for a given input. These retrieved functions are then provided as context to an LLM to generate final summaries.

C. Main Results

We assess the Code Summarization performance using the ROUGE metric [20]. We also employ BERTScore [21] to evaluate the experimental results which leverages a pretrained BERT model to compute the similarity between the generated summaries and the reference labels. In addition to the above reference-based metrics, we introduce the LLM-as-a-Judge measure that leverages GPT-4o and DeepSeek through prompt-based assessment [22].

As shown in Table I, the incorporation of additional information through GraphRAG leads to a substantial improvement in code summarization performance. We compare the results by using CodeT5 [18], GPT-4o, DeepSeek, and Claude. The current findings indicate that DeepSeek achieves the superior performance, with a ROUGE-1 score of 0.2953 compared to 0.2163 in the baseline experiment on the FILE-CS Dataset. Based on this observation, we further employ Deepseek as the main backbone LLM and evaluate the performance of different generation LLM such as CodeT5, CodeLlama [19], and GraphCodeBERT [25] to generate the target code summarization. As presented in Table I, the result produced by GraphCodeBERT achieves the best performance on ROUGE-1, i.e., 0.2987, for the FILE-CS Dataset.

The performance on the CodeSearchNet dataset improves substantially across all metrics. The ROUGE-1 score is increased from 0.2567 to 0.3053, while the BERTScore, GPT-Judged Score, and DeepSeek-Judged Score reach 0.6054, 0.5687, and 0.5010, respectively.

To further validate the generalizability of the proposed approach, we performed comparative experiments on the Java portion of the CodeSearchNet dataset. As illustrated in Figure II, our method yields a notable performance gain, improving the ROUGE-1 score from 0.2356 to 0.3384.

D. Hierarchy Programing Analysis

In the proposed framework, the source code utilized for the Code Summarization task is hierarchically partitioned into three distinct levels of granularity—repository, class, and function. This hierarchical design enables the model to capture both high-level structural dependencies and fine-grained semantic details, facilitating a more comprehensive understanding of code context. To quantitatively evaluate the contribution of each granularity and the effect of their integration, we conducted an extensive ablation study, with the experimental results summarized in Figure 2. The results clearly demonstrate that incorporating information from multiple granularities leads to consistent and significant improvements over the baseline model. These findings suggest that multi-level representations effectively enhance the model's ability to generate more accurate and context-aware summaries, underscoring the importance of hierarchical granularity modeling in code summarization tasks.

E. Parameter Analysis

In the GraphRAG retrieval process, the hop count of the search range serves as a crucial hyperparameter. The results in Table III demonstrate that the 2-hop setting achieves the best performance. Its ROUGE-1 score of 0.2864 and BertScore of 0.5738 demonstrate a strong performance on retrieving the sufficient information and filtering extraneous noise. Moreover, the results indicate that increasing the number of hops allows more related information to be retrieved, yet it also introduces a large amount of irrelevant information into the

TABLE I

PYTHON CODE SUMMARIZATION PERFORMANCE OF THE COMPARED METHODS ON THE FILE-CS AND CODESEARCHNET DATASETS. RESULTS ARE REPORTED USING ROUGE-1, ROUGE-2, ROUGE-L, BERTSCORE, GPT JUDGE SCORE, AND DEEPSEEK JUDGE SCORE TO COMPREHENSIVELY EVALUATE THE QUALITY OF THE GENERATED CODE SUMMARIES.

Retrival Model +	**Reference-Based Metrics**			**LLM-based Evaluators**		
Generation Model	Rouge-1	Rouge-2	Rouge-L	BERTScore	GPT Judge	DeepSeek Judge
FILE-CS Dataset						
Baseline						
CodeT5 + CodeT5	.2163	.0273	.1696	.5084	.4656	.4267
CodeLlama + CodeLlama	.2263	.0312	.1756	.5263	.4812	.4463
GraphCodeBERT + GraphCodeBERT	.2301	.0289	.1723	.5163	.4793	.4521
GPT-4o + GPT-4o	.2876	.0874	.2063	.5579	.5589	.4893
DeepSeek-v2 + DeepSeek-v2	.2953	**.0897**	.1986	.5681	.5921	**.5284**
Claude-3 Sonnet + Claude-3 Sonnet	.2765	.0812	.1905	.5493	.5863	.5103
Qwen2.5-14B + Qwen2.5-14B	.2789	.0825	.1925	.5526	.5897	.5134
Ours (Code-GraphRAG)						
DeepSeek-v2 + CodeT5	.2765	.0823	.1900	.5543	.5769	.5069
DeepSeek-v2 + CodeLlama	.2864	.0853	**.2124**	.5675	.5864	.5278
DeepSeek-v2 + GraphCodeBERT	**.2987**	.0876	.2053	**.5763**	**.5971**	.5163
CodeSearchNet Dataset						
Baseline						
CodeT5 + CodeT5	.2567	.0674	.1935	.5364	.4863	.4169
CodeLlama + CodeLlama	.2674	0712	.2056	.5493	.4921	.4209
GraphCodeBERT + GraphCodeBERT	.2702	.0705	.2089	.5467	.4952	.4215
GPT-4o + GPT-4o	.2876	.0884	.2431	.5891	.5241	.4698
DeepSeek-v2 + DeepSeek-v2	.3012	**.0980**	.2602	.5954	.5569	.4963
Claude-3 Sonnet + Claude-3 Sonnet	.2956	.0915	**.2610**	.5876	.5432	.5002
Qwen2.5-14B + Qwen2.5-14B	.2987	.0965	.2602	.5821	.5502	.4963
Ours (Code-GraphRAG)						
DeepSeek-v2 + CodeT5	.2971	.0902	.2543	.5936	.5469	.4896
DeepSeek-v2 + CodeLlama	**.3053**	.0956	.2567	**.6054**	.5512	.4932
DeepSeek-v2 + GraphCodeBERT	.2989	.0952	.2512	.5985	**.5687**	**.5010**

TABLE II

JAVA CODE SUMMARIZATION PERFORMANCE OF THE COMPARED METHODS ON THE CODESEARCHNET DATASETS. RESULTS ARE REPORTED USING ROUGE-1, ROUGE-2, ROUGE-L, BERTSCORE, GPT JUDGE SCORE, AND DEEPSEEK JUDGE SCORE TO COMPREHENSIVELY EVALUATE THE QUALITY OF THE GENERATED CODE SUMMARIES.

Retrival Model +	**Reference-Based Metrics**			**LLM-based Evaluators**		
Generation Model	Rouge-1	Rouge-2	Rouge-L	BERTScore	GPT Judge	DeepSeek Judge
CodeSearchNet Dataset						
Baseline						
CodeT5 + CodeT5	.2356	.0384	.1826	.5269	.4952	.4539
CodeLlama + CodeLlama	.2418	.0398	.1886	.5346	.5012	.4620
GraphCodeBERT + GraphCodeBERT	.2412	.0379	.1856	.5226	.5001	.4596
GPT-4o + GPT-4o	.3015	.0569	.2157	.5612	.5284	.4963
DeepSeek-v2 + DeepSeek-v2	.3102	.0596	.2202	.5732	.5324	.5023
Claude-3 Sonnet + Claude-3 Sonnet	.3120	.0592	.2068	.5710	.5226	.5005
Qwen2.5-14B + Qwen2.5-14B	.3134	.0586	.2052	.5654	.5201	.4985
Ours (Code-GraphRAG)						
DeepSeek-v2 + CodeT5	.3320	.0658	.2269	.5963	.5520	.5123
DeepSeek-v2 + CodeLlama	**.3384**	**.0681**	.2235	**.6023**	**.5541**	**.5201**
DeepSeek-v2 + GraphCodeBERT	.3310	.0671	**.2291**	.5945	.5512	.5156

subsequent LLM processing. This suggests that while the 1-hop expansion captures immediate dependencies (e.g., a containing class), the optimal 2-hop setting is crucial for assembling a more complete architectural context, such as interactions between collaborating modules. Conversely, the performance degradation at the 3-hop setting indicates that the subgraph expands into overly generic, low-level utilities, diluting the prompt with noisy information that distracts from the core logic to be summarized.

To further investigate the sensitivity of our approach to generation parameters, we conducted experiments focusing on the temperature hyperparameter. The temperature value in the API determines the randomness of token sampling during generation, thereby influencing the trade-off between determinism and diversity in the model's outputs. As presented in Table III, we observed that setting the temperature either too low or too high leads to suboptimal performance. Empirically, a temperature of 0.5 was found to provide the best balance between the consistency and the creativity, yielding

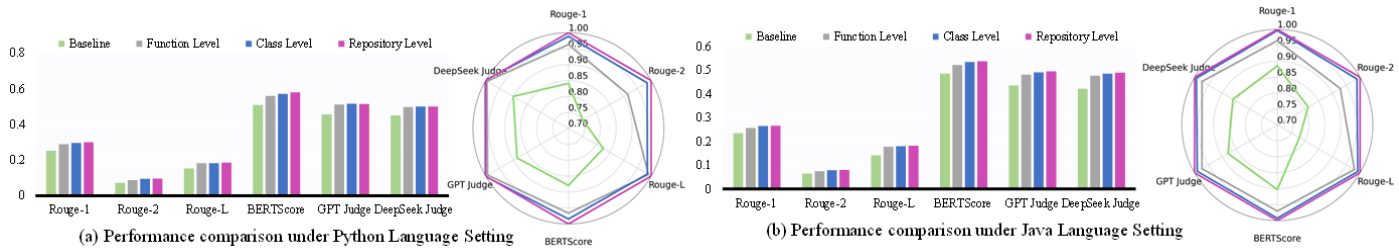

Fig. 2. Results of the ablation study evaluating the performance improvements achieved by incorporating information from different levels of granularity (repository, class, and function).

TABLE III
HYPERPARAMETER EVALUATION OF DIFFERENT HOP COUNTS ON A SMALL SUBSET OF THE FILE-CS DATASET.

Method	Rouge-1	Rouge-2	Rouge-L	BERTScore	GPT Judge	DeepSeek Judge
0-hop	.2189	.0241	.1705	.5067	.4269	.4141
1-hop	.2763	.0812	.1856	.5693	.4978	.4702
2-hop	**.2864**	**.0871**	**.2015**	**.5738**	.5064	**.4829**
3-hop	.2715	.0756	.1953	.5612	**.5069**	.4825
temp=0	.2567	.0824	.1795	.5568	.4863	.4615
temp=0.25	.2610	.0856	.1815	.5658	.4886	.4657
temp=0.5	**.2623**	.0848	**.1820**	**.5712**	**.4920**	**.4721**
temp=0.75	.2615	**.0859**	.1818	.5689	.4912	.4718
temp=1.0	.2605	.0842	.1805	.5672	.4905	.4695

the highest overall performance on the task. These experimental results highlight the importance of appropriately tuning the temperature parameter to ensure both the fluency and the informativeness in the generated summaries.

V. CONCLUSION

In this paper, we have proposed Code-GraphRAG, a novel framework that leverages GraphRAG's hierarchical community structure to address the intrinsic organizational complexity of software projects. By explicitly modeling the relationships between code entities—spanning from functions and classes to files and entire repositories, our approach systematically mitigates the persistent challenge of context fragmentation inherent in traditional vector-based RAG pipelines. Extensive experimental results on two benchmark datasets demonstrate that Code-GraphRAG significantly improves the performance.

ACKNOWLEDGMENT

This project has received funding from the National Natural Science Foundation of China (Nos.62306156, 72571150,and 62106091);the Tianjin Municipal Science and Technology Bureau (No.25JCZDSN00020);and the Shandong Provincial Natural Science Foundation (No.ZR2025MS1078).

REFERENCES

[1] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys*, vol. 51, no. 4, pp. 1–37, 2018.
[2] S. Haiduc, J. Aponte, L. Moreno *et al.*, "On the use of automated text summarization techniques for summarizing source code," in *Working Conference on Reverse Engineering*, 2010, pp. 35–44.
[3] G. Sridhara, E. Hill, D. Muppaneni *et al.*, "Towards automatically generating summary comments for java methods," in *ASE*, 2010, pp. 43–52.
[4] S. Iyer, I. Konstas, A. Cheung *et al.*, "Summarizing source code using a neural attention model," in *ACL*, 2016, p. 2073–2083.
[5] Y. Wang, H. Le, A. Gotmare *et al.*, "Codet5+: Open code large language models for code understanding and generation," in *EMNLP*, 2023, pp. 1069–1088.
[6] B. Rozière, J. Gehring, F. Gloeckle *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2024.
[7] D. Guo, Q. Zhu, D. Yang *et al.*, "Deepseek-coder: When the large language model meets programming – the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.
[8] M. Li, H. Yu, G. Fan *et al.*, "Classsum: a deep learning model for class-level code summarization," *Neural Comput. Appl.*, vol. 35, no. 4, p. 3373–3393, 2022.
[9] M. Malhotra and K. C. Jitender, "Class level code summarization based on dependencies and micro patterns," in *International Conference on Inventive Communication and Computational Technologies*, 2018, pp. 1011–1016.
[10] C. Haoling, Y. Huiqun, F. Guisheng *et al.*, "Class summarization generation technology based on hierarchical representation and context enhancement," *Journal of Computer Research and Development*, vol. 61, no. 2, pp. 307–323, 2024.
[11] M. Chen, J. Tworek, H. Jun *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.
[12] W. Sun, Y. Miao, Y. Li *et al.*, "Source code summarization in the era of large language models," in *ICSE*, 2025, pp. 1882–1894.
[13] V. Makharev and V. Ivanov, "Code summarization beyond function level," *arXiv preprint arXiv:2502.16704*, 2025.
[14] Y. Wan, Z. Zhao, M. Yang *et al.*, "Improving automatic source code summarization via deep reinforcement learning," in *ASE*, 2018, pp. 397–407.
[15] R. Cai, Z. Liang, B. Xu *et al.*, "TAG : Type auxiliary guiding for code comment generation," in *ACL*, 2020, pp. 291–301.
[16] N. Dhulshette, S. Shah, and V. Kulkarni, "Hierarchical repository-level code summarization for business applications using local llms," *arXiv preprint arXiv:2501.07857*, 2025.
[17] D. Edge, H. Trinh, N. Cheng *et al.*, "From local to global: A graph rag approach to query-focused summarization," *arXiv preprint arXiv:2404.16130*, 2025.
[18] Y. Wang, W. Wang, S. Joty *et al.*, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.
[19] B. Roziere, J. Gehring, F. Gloeckle *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.
[20] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81.
[21] T. Zhang, V. Kishore, F. Wu *et al.*, "Bertscore: Evaluating text generation with bert," *arXiv preprint arXiv:1904.09675*, 2019.
[22] L. Zheng, W.-L. Chiang, Y. Sheng *et al.*, "Judging llm-as-a-judge with mt-bench and chatbot arena," in *NeurIPS*, 2023, pp. 46 595–46 623.
[23] Y. Wang, Y. Huang, D. Guo *et al.*, "Sparsecoder: Identifier-aware sparse transformer for file- level code summarization," in *IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2024, pp. 614–625.
[24] H. Husain, H.-H. Wu, T. Gazit *et al.*, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.
[25] D. Guo, S. Ren, S. Lu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.