Feedback-Guided Reranking for Retrieval-Augmented Code Completion

Anonymous ACL submission

Abstract

Retrieval-augmented code completion aims to enhance code generation by leveraging retrieved code snippets as references, which serves as a core technology to improve development efficiency. However, existing approaches face a critical limitation: the misalignment of preferences between retrievers and generators. To address this issue, we propose **F**eedback-**G**uided **R**eranking for **R**etrieval-augmented Code Completion (FGRR), a novel method that leverages feedback from the generative model to fine-tune the parameters of a reranker. By inserting a reranking module between the retriever and generator, FGRR effectively bridges the preference gap and enhances the generator's ability to utilize the retrieved snippets. Experiments demonstrate that FGRR achieves substantial gains in performance across token-level, line-level, and body-level code completion tasks.

1 Introduction

Code completion is a cornerstone feature in software Integrated Development Environments (IDEs), significantly boosting developer productivity. Previous deep neural networks-based approaches(Liu et al., 2016; Alon et al., 2020; Karampatsis et al., 2020) and pretraining-based approaches(Liu et al., 2020; Svyatkovskiy et al., 2020; Feng et al., 2020; Wang et al., 2021) primarily rely on limited training data for fine-tuning and typically use previously written code as context input. This restricts their ability to effectively handle domain-specific code completion scenarios.

Recently, retrieval-augmented generation (RAG) has emerged as an effective framework for code completion tasks(Lu et al., 2022; Zhang et al., 2023; Guo et al., 2024; Chen et al., 2024). These approaches retrieve relevant code snippets or code structure information(Liu et al., 2024) from a large-scale codebase and incorporate them into the generative model's context. This provides the model with additional reference information, improving the accuracy and contextual relevance of the generated code. However, a critical limitation of existing approaches is the misalignment of preferences between retrievers and generators(Ke et al., 2024; Dong et al., 2024). Since retrievers and generators are typically trained in isolation, the retrieved snippets often fail to meet the specific needs of the generative model, leading to suboptimal code completion performance.

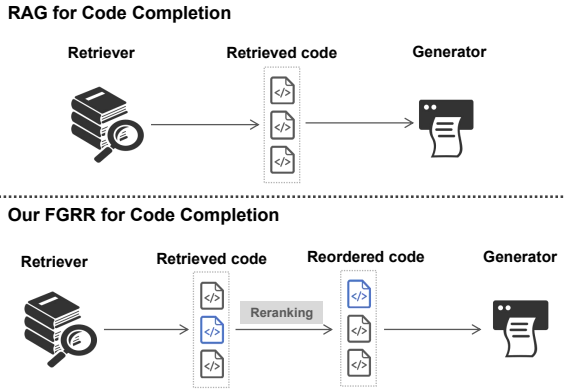

Figure 1: Comparison of two retrieval-augmented code completion frameworks. The upper shows the standard RAG, where the retrieved code is fed to the generator directly. The lower shows our FGRR, where the retrieved code is reordered before being fed into the generator.

In this paper, we propose FGRR, a novel method that leverages feedback from the generative model to optimize the retrieval process. As shown in Figure 1, the core idea is to fine-tune a reranker using the generator's feedback, enabling it to rank code snippets based on their relevance to the generator's needs. FGRR operates through two stages of fine-tuning: supervised fine-tuning and feedback-guided fine-tuning. In the supervised fine-tuning stage, the reranker is trained to match the semantic relevance between query and code snippets. In the feedback-guided fine-tuning stage, we further refines the reranker using feedback from the generator's performance. Experimental results

across token-level, line-level, and body-level code completion tasks show that the generator can receive higher-quality reference code snippets, better aligned with its task, ultimately improving code completion performance.

2 Related Work

Deep Code Completion. Code completion is a crucial factor in improving software development efficiency.Early research shows that deep neural networks(Liu et al., 2016; Alon et al., 2020) and pre-training methods(Liu et al., 2020; Svyatkovskiy et al., 2020; Feng et al., 2020) have made significant advances. While some approaches incorporate code-specific structures like Abstract Syntax Trees (AST) (Kim et al., 2021; Guo et al., 2022), most current research treats source code as token sequences(Nijkamp et al., 2022; Li et al., 2023).

Retrieval-augmented Code Completion. While traditional approaches rely on local context, recent approaches leverage external relevant code snippets(Guo et al., 2024; Chen et al., 2024; Liu et al., 2024; Wang et al., 2024). The ReACC framework(Lu et al., 2022) retrieves similar code for completion, achieving strong results in Python and Java. RepoCoder(Zhang et al., 2023) iterates retrieval and generation to improve completion quality at the repository level. ProCC(Tan et al., 2024) further boosts performance with multi-retrieval and context-based algorithms. These approaches demonstrate the potential of retrieval-augmented techniques to improve code completion.

3 Methods

This section introduces FGRR, a feedback-guided reranking method for retrieval-augmented code completion. The overall framework of FGRR is shown in Figure 2.

3.1 Supervised Fine-Tuning Stage

In the first stage of FGRR, the reranker is fine-tuned to to capture the semantic associations between query and code. It is trained using contrastive learning, where a query is paired with a positive code snippet and several random negative ones. The contrastive loss encourages the model to increase the similarity between positive pairs and decrease it for negative pairs. The loss function is:

$$L_{\text{rerank}} = -\frac{1}{N}\sum_{i=1}^{N}\log\frac{\exp(\delta(q_i, c_i^+))}{\sum_j \exp(\delta(q_i, c_j))}, \quad (1)$$

where N is the batch size, q_i is the query, c_i^+ is the positive code snippet, c_j is the negative code snippet, and $\delta(\cdot)$ represents the similarity function.

3.2 Feedback-Guided Fine-Tuning Stage

In the second stage of FGRR, feedback from the generator is used to fine-tune the reranker. The goal is to align the retriever and generator preferences. The generator is evaluated based on code completion metrics, which serve as feedback for the reranker. The reward for a candidate retrieval is defined as:

$$reward(r_i) = \begin{cases} 1 & \text{if } Metrics(r_i) \geq Metrics(r_j), \\ & \forall r_j \in R, \\ 0 & \text{otherwise.} \end{cases} \quad (2)$$

where $R = \{r_1, r_2, \ldots, r_k\}$ is the set of candidate code snippets, and $Metrics$ refers to the evaluation metric used for code completion.

The reranker's objective is to adjust its ranking probabilities based on the feedback from the generator, prioritizing code snippets that improve the generator's performance. The optimization loss for the reranker is:

$$L = -\sum_{i=1}^{k}(reward(r_i) \times \log P(r_i|C, R)), \quad (3)$$

where k is the number of candidate code snippets, and $P(r_i|C, R)$ is the probability of selecting a code snippet r_i given the context C and retrieved code examples R.

3.3 Inference Stage

The inference stage of FGRR consists of three components: candidate retrieval, reranking, and code generation. The retriever uses a fast retrieval engine, such as Lucene (Białecki et al., 2012), to retrieve relevant code snippets from a large-scale codebase. The retrieval process is:

$$\{r_1, r_2, \ldots, r_k\} = Retriever(C_{context}, D_{codebase}), \quad (4)$$

where $C_{context}$ is the code context and $D_{codebase}$ is the codebase.

The reranker reorders the retrieved snippets, refining the results based on the fine-grained semantic relationships between the context and the candidates. This ensures that the most relevant snippets for the generator are prioritized. The reranking

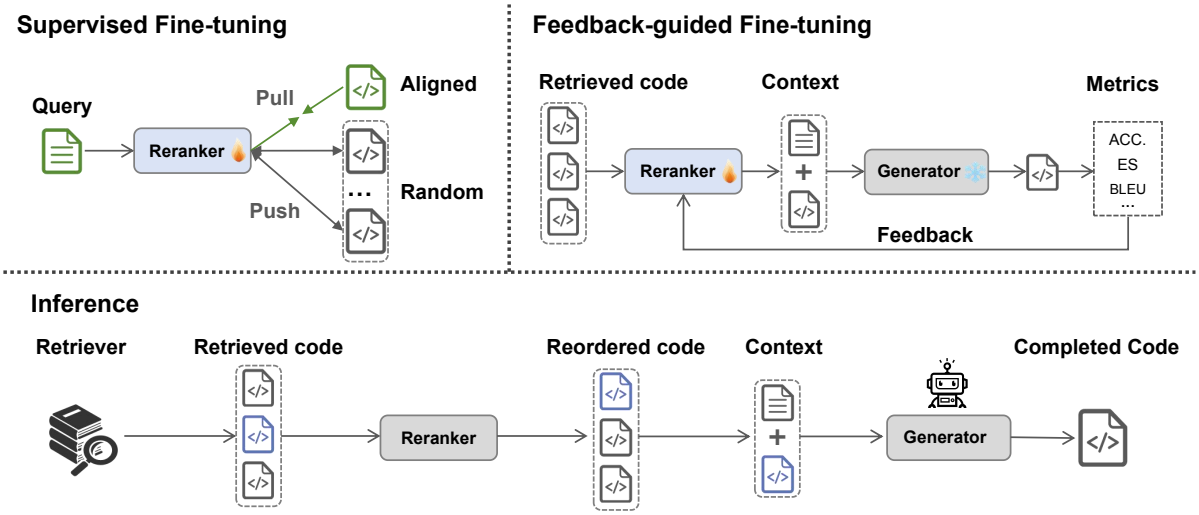

Figure 2: The overall framework of FGRR. In the supervised fine-tuning stage, the reranker is trained to match the semantic relevance between query and code snippets. In the feedback-guided fine-tuning stage, the reranker is further optimized by using feedback from the generator. In the the inference stage, the reranker is inserted to improve the ordering of retrieved code.

process is:

$$\{r_1^*, r_2^*, \ldots, r_k^*\} = Reranker(r_1, r_2, \ldots, r_k), \quad (5)$$

where $\{r_1^*, r_2^*, \ldots, r_k^*\}$ are the reranked top-K candidates.

Finally, the generator uses the context and the reranked reference code snippets to generate a complete code snippet. The code completion process is:

$$\hat{y} = Generator(C_{context}, r_1^*), \quad (6)$$

where $\hat{y}$ is the generated code snippet, and r_1^* is the most relevant reference code example.

4 Experiments

4.1 Research Questions

RQ1: How does FGRR perform in token-level, line-level, and body-level code completion tasks? This question investigates the framework's effectiveness across different granularities of code completion tasks.

RQ2: What is the contribution of each training step in FGRR to its overall performance? We conduct ablation studies to understand how each individual steps contributes to performance enhancement.

RQ3: How effective is the FGRR reranker in alleviating preference gap between in retrieval-augmented code completion framework? This question examines the reranking process, focusing on its optimization of candidate code selections.

4.2 Datasets

This experiment utilizes the Python code completion dataset from CodeXGLUE (Lu et al., 2021), which is derived from the PY150 dataset (Raychev et al., 2016) collected from GitHub. Additionally, the PyTorrent dataset (Bahrami et al., 2021) is employed as the code retrieval library. The detailed statistics of these datasets are provided in Table 1.

Dataset	Training	Validation	Test	Avg. code tokens
PY150	82,143	4,169	5,894	92.2
PyTorrent		2,841,300		90.3

Table 1: The statistics of code completion datasets and retrieval codebase.

4.3 Models and Metrics

We evaluate our method using three open-source models and two closed-source models as base model. The open-source models include Transformer Decoder(TFM)(Vaswani, 2017), Transformer-XL(Tr-XL)(Dai, 2019), and CodeGPT(Lu et al., 2021), while the closed-source models are gpt-3.5-turbo(OpenAI, 2023) and gpt-4o-mini(OpenAI, 2024).

We evaluate the performance of the code completion task using several metrics based on different levels of granularity. The metrics used in this experiment are Accuracy, Edit Similarity(ES)(Wang et al., 2012), Exact Matching(EM), BLEU(Papineni et al., 2002), and CodeBLEU(Ren et al., 2020). These metrics enable a comprehensive assessment of the model's performance from various perspectives.

Model	Token-level Code Completion														
	Identifier Acc.(%)			Separator Acc.(%)			Keyword Acc.(%)			Operator Acc.(%)			Overall Acc.(%)		
	Ori.	RAG	FGRR	Ori.	RAG	FGRR	Ori.	RAG	FGRR	Ori.	RAG	FGRR	Ori.	RAG	FGRR
TFM	29.3	40.6	**43.5 ↑48.5**	73.2	76.8	**77.4 ↑5.7**	52.0	58.9	**60.5 ↑16.3**	43.7	48.5	**48.7 ↑11.4**	52.7	59.2	**60.4 ↑14.6**
Tr-XL	30.9	42.3	**45.2 ↑46.3**	73.6	77.8	**78.5 ↑6.7**	52.9	59.0	**60.5 ↑14.4**	45.5	50.7	**51.2 ↑12.5**	53.5	60.1	**61.5 ↑15.0**
CodeGPT	43.3	50.5	**52.9 ↑22.2**	77.5	80.2	**80.3 ↑3.6**	57.8	63.2	**64.0 ↑10.7**	52.2	56.7	**57.6 ↑10.3**	60.6	65.2	**66.4 ↑9.6**
	Line-level Code Completion														
Model	ES(%)			EM(%)			BLEU-1(%)			BLEU-2(%)			BLEU-4(%)		
	Ori.	RAG	FGRR	Ori.	RAG	FGRR	Ori.	RAG	FGRR	Ori.	RAG	FGRR	Ori.	RAG	FGRR
TFM	40.6	50.2	**52.6 ↑29.6**	1.3	10.3	**12.8 ↑884.6**	29.9	38.5	**41.7 ↑39.5**	21.6	31.5	**34.7 ↑60.6**	17.3	27.5	**30.7 ↑77.5**
Tr-XL	45.1	52.5	**55.6 ↑23.3**	1.4	11.4	**14.9 ↑964.3**	30.3	39.1	**43.0 ↑41.9**	23.9	33.3	**37.6 ↑57.3**	18.4	28.3	**32.6 ↑77.2**
CodeGPT	43.1	55.0	**58.6 ↑36.0**	4.1	14.7	**19.3 ↑370.7**	33.8	42.6	**47.2 ↑39.6**	27.7	37.0	**41.9 ↑51.3**	22.0	31.7	**36.7 ↑66.8**
	Body-level Code Completion														
Model	ES(%)			BLEU-1(%)			BLEU-2(%)			BLEU-4(%)			CodeBLEU(%)		
	Ori.	RAG	FGRR	Ori.	RAG	FGRR	Ori.	RAG	FGRR	Ori.	RAG	FGRR	Ori.	RAG	FGRR
gpt-3.5-turbo	40.5	52.9	**54.7 ↑35.1**	18.8	36.3	**38.2 ↑103.2**	14.6	30.4	**33.3 ↑128.1**	10.8	24.6	**28.1 ↑160.2**	26.8	32.6	**33.8 ↑26.1**
gpt-4o-mini	41.2	54.1	**54.8 ↑33.0**	29.2	40.6	**40.9 ↑40.1**	19.4	32.4	**34.4 ↑77.3**	11.7	24.8	**28.2 ↑141.0**	29.1	31.7	**34.6 ↑18.9**

Table 2: Performance of models on code completion tasks. RAG enhances originals model by incorporating retrieval-based augmentation, while FGRR further improves RAG by applying a reranking mechanism.

5 Results

5.1 RQ1: Effectiveness of FGRR

Table 2 presents the performance of our method on the code completion task, comparing different models and evaluation metrics. We categorize token completion into Identifier, Separator, Keyword, and Operator for a detailed analysis. Results show that our reranking method outperforms both the original model and the retrieval-augmented generation approach, with significant improvements in overall token-level accuracy and substantial gains in line-level and body-level metrics like BLEU and CodeBLEU. The reranking mechanism effectively bridges the preference gap between the retriever and generator, improving code completion results.

5.2 RQ2: Ablation Study

In this ablation study, we assess the impact of key components: feedback-guided fine-tuning, supervised fine-tuning, and the reranking module.. As shown in Table 3, feedback-guided fine-tuning is crucial for optimizing retrieval ranking. Removing the reranking module reduces the model to a basic RAG approach, while removing supervised fine-tuning significantly weakens the reranker's semantic matching ability, performing worse than direct retrieval. The study highlights the importance of both fine-tuning stages in enhancing the reranking process, leading to more accurate and relevant code generation across all completion levels.

5.3 RQ3: Alleviation of Preference Gap

As shown in Figure 3, we compare the number of optimal code snippets retrieved from the top-12 candidates by two methods. The results show that

Token-level Completion	Identifier	Seperator	Keyword	Operaotr	Overall
CodeGPT+FGRR	**52.9**	**80.3**	**64.0**	**57.6**	**66.4**
- Feedback Fine-tuning	51.1	80.2	63.5	56.9	65.5
- Supervised Fine-tuning	45.9	78.1	58.7	52.9	62.3
- Reranking Module	50.5	80.2	63.2	56.7	65.2
Line-level Completion	ES	EM	BLEU-1	BLEU-2	BLEU-4
CodeGPT+FGRR	**58.6**	**19.3**	**47.2**	**41.9**	**36.7**
- Feedback Fine-tuning	55.4	15.1	43.2	37.7	32.6
- Supervised Fine-tuning	51.2	7.7	37.5	31.5	25.8
- Reranking Module	55.0	14.7	42.6	37.0	31.7
Body-level Completion	ES	CodeBLEU	BLEU-1	BLEU-2	BLEU-4
gpt-4o-mini+FGRR	**54.8**	**34.6**	**40.9**	**34.4**	**28.2**
- Feedback Fine-tuning	54.8	32.5	41.7	33.3	25.6
- Supervised Fine-tuning	48.3	28.6	32.5	24.6	17.6
- Reranking Module	54.1	31.7	40.6	32.4	24.8

Table 3: Ablation study of FGRR.

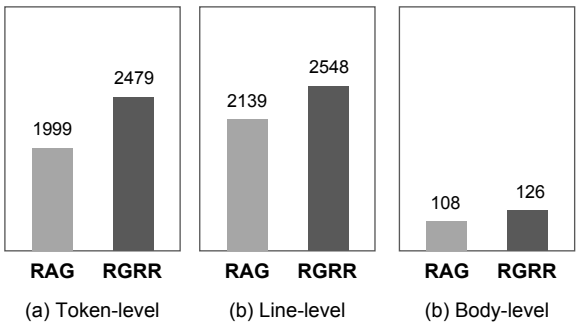

Figure 3: Comparison of retrieving optimal code snippets. Only 500 test samples are used for body-level code completion.

FGRR outperforms the standard RAG in top-1 recall, demonstrating the effectiveness of the reranker in alleviating the preference gap.

6 Conclusion

In this paper, we introduced FGRR, a novel method that improves retrieval-augmented code completion by aligning preference gap of retrievers and generators through feedback-guided reranking. Experimental results show significant performance gains across different code completion tasks.

7 Limitations

FGRR's performance heavily depends on the quality of the feedback provided by the generator. Since the reranker is trained using task-specific metrics, it may vary in effectiveness when applied to different tasks or code domains, potentially limiting its generalization across various code completion scenarios. Additionally, the reranking stage relies on dense embeddings for retrieval, which can become inefficient when dealing with large-scale codebases, as it requires costly computations to reorder a vast number of candidate code snippets. These limitations suggest that future work should focus on improving the efficiency of the reranking process and developing more robust feedback mechanisms for broader task applicability.

References

Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. 2020. Structural language models of code. In *International conference on machine learning*, pages 245–256. PMLR.

Mehdi Bahrami, NC Shrikanth, Shade Ruangwan, Lei Liu, Yuji Mizobuchi, Masahiro Fukuyori, Wei-Peng Chen, Kazuki Munakata, and Tim Menzies. 2021. Pytorrent: A python library corpus for large-scale language models. *arXiv preprint arXiv:2110.01710*.

Andrzej Białecki, Robert Muir, Grant Ingersoll, and Lucid Imagination. 2012. Apache lucene 4. In *Proceedings of the SIGIR 2012 Workshop on Open Source Information Retrieval*, page 17.

Junkai Chen, Xing Hu, Zhenhao Li, Cuiyun Gao, Xin Xia, and David Lo. 2024. Code search is all you need? improving code suggestions with code search. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Zihang Dai. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*.

Guanting Dong, Yutao Zhu, Chenghao Zhang, Zechen Wang, Zhicheng Dou, and Ji-Rong Wen. 2024. Understand what llm needs: Dual preference alignment for retrieval-augmented generation. *arXiv preprint arXiv:2406.18676*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. 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)*, pages 7212–7225.

Qi Guo, Xiaohong Li, Xiaofei Xie, Shangqing Liu, Ze Tang, Ruitao Feng, Junjie Wang, Jidong Ge, and Lei Bu. 2024. Ft2ra: A fine-tuning-inspired approach to retrieval-augmented code completion. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 313–324.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code!= big vocabulary: Open-vocabulary models for source code. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1073–1085.

Zixuan Ke, Weize Kong, Cheng Li, Mingyang Zhang, Qiaozhu Mei, and Michael Bendersky. 2024. Bridging the preference gap between retrievers and llms. *arXiv preprint arXiv:2401.06954*.

Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 150–162. IEEE.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Chang Liu, Xin Wang, Richard Shin, Joseph E Gonzalez, and Dawn Song. 2016. Neural code completion.

Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 473–485.

Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. 2024. Graphcoder: Enhancing repository-level code completion via code context graph-based retrieval and language model. *arXiv preprint arXiv:2406.07003*.

Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*, pages 6227–6240.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

OpenAI. 2023. Gpt-3.5 turbo fine-tuning and api updates.

OpenAI. 2024. Gpt-4o mini: advancing cost-efficient intelligence.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318.

Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices*, 51(10):731–747.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 1433–1443.

Hanzhuo Tan, Qi Luo, Ling Jiang, Zizheng Zhan, Jing Li, Haotian Zhang, and Yuqun Zhang. 2024. Prompt-based code completion via multi-retrieval augmented generation. *arXiv preprint arXiv:2405.07530*.

A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems*.

Wei Wang, Jianbin Qin, Chuan Xiao, Xuemin Lin, and Heng Tao Shen. 2012. Vchunkjoin: An efficient algorithm for edit similarity joins. *IEEE Transactions on Knowledge and Data Engineering*, 25(8):1916–1929.

Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2024. Rlcoder: Reinforcement learning for repository-level code completion. *arXiv preprint arXiv:2407.19487*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. 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.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484.

A Additional Experimental Details

A.1 Comparison of Retrieval Strategies

Due to the large size of the retrieval corpus, dense retrieval methods proved to be inefficient, so we employed the Lucene search engine for preliminary retrieval, obtaining the top 1000 candidate code snippets. We compared three retrieval strategies on code completion tasks. The three retrieval strategies are as follows:

- **Header2Code**: The retriever uses the method header as the query to find matching code snippets.
- **NL2Code**: The retriever uses the comment associated with the code as the query to find relevant code snippets in the codebase.
- **NL2NL**: The retriever searches for semantically similar comments within the codebase and retrieves the corresponding code snippets.

Following the experimental setup of Chen et al. (2024), we compared the results of these strategies, as shown in the Table 4. Based on the performance, we selected the NL2NL strategy as the default retrieval method for the RAG approach, as it yields the best results.

Model	Strategies	Token-level		Line-level	
		Identifier	Overall	ES	BLEU-4
CodeGPT	Original	43.3	60.6	43.1	22.0
	Header2Code	47.8	63.5	51.8	26.6
	NL2Code	44.7	61.5	48.8	22.6
	NL2NL	**50.5**	**65.2**	**55.0**	**31.7**

Table 4: Comparison of three retrieval strategies on token-level and line-level code completion tasks

A.2 Experimental Setup

For our experiments, the reranker is initialized using UniXcoder(Guo et al., 2022), a state-of-the-art code representation model. The training is performed on an NVIDIA 3090 GPU with 24GB of memory. Our approach involves two stages of fine-tuning: supervised fine-tuning and feedback-guided fine-tuning.

In the supervised fine-tuning stage, the reranker is trained to match the semantic relevance between code queries and retrieved snippets. We use the CodeSearchNet(Husain et al., 2019) Python dataset, which contains a large collection of code snippets and their associated queries, as the training data. The reranker is trained for two epochs, with a learning rate of 2e-5 and a batch size of 32.

The feedback-guided fine-tuning stage refines the reranker further by leveraging feedback from the generator's performance in the PY150 code completion dataset. The reranker is trained for one epoch with the same learning rate and batch size settings. We select the top-12 code snippets from the retrieved candidates. These snippets are then evaluated based on their relevance to the generator's performance in the code completion task. The most optimal exemplar, as determined by the generator's completion quality, is chosen. For feedback, token-level code completion uses Accuracy, line-level code completion uses Edit Similarity, and body-level code completion uses BLEU score as the evaluation metric. The reranker is then trained to prioritize this optimal exemplar to enhance its performance in future retrieval tasks.

B Case Study

Take "Generate a function to escape HTML special characters in a string" as an example.

```
# Input Context
def escape_html(s: str) -> str:
    result = ""
    for c in s:
        if c == '<':
            result += "&lt;"
        elif c == '>':
            result += "&gt;"
```

The completion results of model gpt-4o-mini are as follows:

```
# Zero-shot completion by gpt-4o-mini:
        elif c == '&':
            result += "&amp;"
        else:
            result += c
    return result
```

The retrieved top-1 exemplar and completion results are as follows:

```
# Retrieved Exemplar (Top-1):
def escape_quotes(text: str) -> str:
    return text.replace('"', '&quot;').
        replace("'", '&apos;')

# Generated Code:
elif c == '&':
    result += "&amp;"
elif c == '"':
    result += "&quot;"
else:
    result += c
return escape_quotes(result)  #
    Undefined function call
```

The reranked top-1 exemplar and completion results are as follows:

```
# Reranked Exemplar (Top-1):
def escape_html_full(s: str) -> str:
    escapes = {'<':'&lt;', '>':'&gt;', '
        &':'&amp;', '"':'&quot;', "'":'
        &#39;'}
    return ''.join(escapes.get(c, c) for
         c in s)

# Final Completion:
elif c == '&':
    result += "&amp;"
elif c == '"':
    result += "&quot;"
elif c == "'":
    result += "&#39;"
else:
    result += c
return result
```

This case study demonstrates three critical findings through an HTML escaping function generation task. **First**, the zero-shot completion reveals logical gaps in handling domain-specific requirements: while correctly escaping < and >, it fails to address quotes (" and '), creating security vulnerabilities. **Second**, naive retrieval augmentation introduces new risks: the model overfits to the retrieved `escape_quotes` exemplar by generating invalid API calls to undefined external functions. **Third**, our reranking mechanism resolves both issues: the final implementation achieves full coverage of HTML special characters (<, >, &, ", ') through direct pattern transfer from the reranked exemplar's dictionary mapping strategy, while eliminating external dependencies.