RepoLC: Repository-Level Code Completion with Light Compressor

Anonymous ACL submission

Abstract

Current approaches commonly integrate repository-level code completion with retrieval-augmented generation. Specifically, private code repositories are utilized as retrieval databases, which aim to supply relevant code chunks to a large language model (LLM). However, incorporating multiple retrieved code chunks into an LLM will increase the cost of inference. This not only decreases the efficiency of the LLM but also impairs the user experience. To address this, we introduce **RepoLC**, which uses a **L**ight module to **C**ompress the retrieved code, thereby reducing the inference cost of LLMs. We insert a Semantic Compressor Encoder (SCE) between the retriever and the generator. Specifically, SCE compresses the retrieved code chunks into fewer high-level tokens and then projects them to the semantic space of the LLM. We propose a two-stage training scheme to train the overall pipeline through semantic alignment and task alignment. Experimental results demonstrate that our approach achieves significant improvements on multiple datasets. Compared to other methods, our approach incurs in minimal loss and achieves an inference time that is almost as efficient as that of in-file processing.

1 Introduction

In modern software development, the repository-level code completion (Zhang et al., 2023; Li et al., 2017a; Liu et al., 2020; Guan et al., 2024; Wu et al., 2024) is crucial for enhancing programmer productivity and streamlining the coding process. Despite the remarkable advancements and applications of large language models (LLMs) (Touvron et al., 2023; Brown et al., 2020; Guo et al., 2025; Dubey et al., 2024) in code understanding and generation (Roziere et al., 2023; Guo et al., 2024; Gong et al., 2024), notable performance bottlenecks still persist in tasks that require cross-file context and specialized task scenarios.

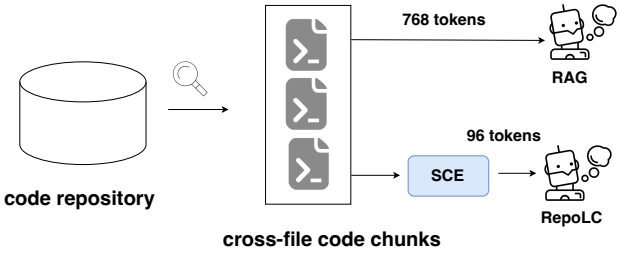


Figure 1: The shows differences between RepoLC and RAG. The retrieved cross-file code chunks, with 768 tokens, can be compressed to 96 via SCE, cutting model inference cost and accelerating the entire process.

For repository-level code completion, many studies have introduced retrieval-augmented generation (RAG) (Gao et al., 2023a; Fan et al., 2024; Cheng et al., 2024c; Zeng et al., 2024) to provide LLMs with additional context and enhance their generation capabilities. However, the direct concatenation of multiple retrieved code chunks into LLM results in a substantial increase in context length. This augmentation can adversely impact the model's inference speed, thereby significantly diminishing user experience and efficiency. Furthermore, it may lead to the "lost in middle" phenomenon within the LLM (Liu et al., 2024b; Yoran et al., 2023; Chen and Shu, 2024; Yu et al., 2023a). Presently, as the demand for private code repositories surges, accelerating this process has become imperative.

Our work focuses on how to enable LLM to achieve better results with shorter contexts. Inspired by relevant methods in NLP (Cheng et al., 2024d; Rau et al., 2024; Chevalier et al., 2023; Zou et al., 2024) , we envision achieving this goal by introducing compression for repository-level code completion. We attempted to utilize some plug-and-play framework (Pan et al., 2024; Jiang et al., 2023b; Ge et al.) for the task. However, the experimental results are not satisfactory. Given that code represents a distinct form of text sequence,

compressing it from the hard-prompt perspective poses challenges in determining the optimal degree of compression for code chunks. We show some compression case in Table 8. For soft-prompt compression methods, alignment is crucial. Directly using pooling to compress the matrix may lead to potential semantic loss or alteration. When soft-prompts lack alignment, it becomes difficult for the LLM to understand the compressed information, making it challenging to achieve satisfactory performance in repository-level code completion. In addition, using an LLM for compression incurs additional time costs.

Based on these challenges, we introduced RepoLC and for the first time attempted to adopt soft-prompt for compression in repository-level code completion. The difference between RepoLC and RAG is shown in Figure 1. We inserted the Semantic Compressor Encoder (SCE) module between the retriever and the generator. This module can extract high-level semantic tokens from the retrieved code chunks, and subsequently the generator will complete the code completion task based on the compressed tokens.

Specifically, our SCE module consists of a light-weight encoder and a projector. The light encoder is utilized to encode code blocks into high-level semantic features. Subsequently, through a pooling operation, the semantic features matrix is mapped to compressed tokens with a specific size. The light-weight encoder and simple operations further reduce the time cost of compression. Subsequently, via a projector, compressed tokens are aligned to the semantic space of the LLM.

To enable the SCE to serve LLMs effectively, we propose a two-stage training scheme. In the first phase, we aim to enable the LLM to understand the output of the SCE. We train the SCE by leveraging the semantic loss of the raw code chunks restored from the compressed ones using the LLM, aiming to align their semantic spaces. After obtaining favorable representations from SCE, we perform task alignment between the SCE and the LLM within the RAG. The SCE compresses the retrieved code chunks respectively, and we train the LLM to complete the code based on the compressed context.

We conduct experiments on multiple datasets, and the results demonstrate the excellent performance of RepoLC. Although we train on Python corpora, RepoLC still exhibits great generalization ability across other programming languages. Additionally, in the ablation experiments, we explore more training combinations. We find that for the entire RepoLC, merely fine-tuning the SCE while keeping the LLM frozen can still yield satisfactory results.

Our contributions are as follows:

- We introduce a novel RAG framework called RepoLC. We utilize a light module to compress the retrieved chunks for LLMs.
- We design a two-stage training scheme that includes semantic alignment and task alignment to enable SCE to serve LLMs.
- Through extensive experiments, we verify the effectiveness of our method. Our approach optimizes the efficiency of RAG to approach in-file and compared with other methods, it does not sacrifice excessive accuracy.

2 Related Work

2.1 Repository-level Code Completion

Traditional code completion techniques primarily focus on code suggestions at the level of individual files or projects (Li et al., 2017a; Tang et al., 2023; Liu et al., 2020) , whereas repository-level code completion extends the scope to the entire code repository (Shrivastava et al., 2023; Cheng et al., 2024a; Liu et al., 2024a; Phan et al., 2024) , aiming to provide more comprehensive and accurate suggestions. RepoCoder (Zhang et al., 2023) enhances the quality of retrieved code chunks by introducing iterative retrieval , while RLCoder (Wang et al., 2024) trains the retriever based on feedback signals from downstream tasks via a generator, eliminating the need for new datasets. DraCo (Cheng et al., 2024b) constructs a graph representation of the entire repository to facilitate information retrieval through structural insights. REPOFUSE (Li et al., 2017b) introduces two types of contextual information to further refine the relevance between retrieved code and incomplete code. These approaches optimize retrieval performance, but Repoformer (Di Wu, 2024) suggests that retrieved code is not always useful. Consequently, Repoformer trains LLM to autonomously determine whether to retrieve code, further improving effectiveness. This investigation, to a certain degree, improves the efficiency of the task and bolsters the performance of LLMs.

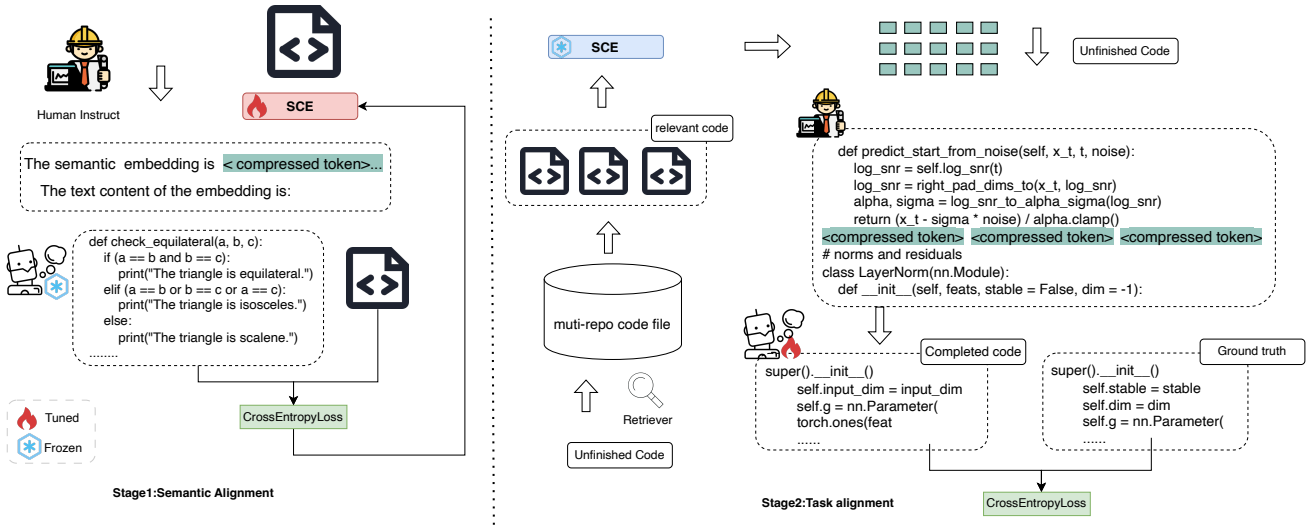


Figure 2: The training strategy of RepoLC

2.2 Retrieval-augmented Generation

To address issues such as hallucinations and knowledge updates in LLM, the RAG was proposed (Lewis et al., 2020) . Equipping LLMs with an external knowledge base has led to improved performance on various NLP tasks (Wang et al., 2023; Han et al., 2024; Gao et al., 2023b) . In the NLP domain, numerous works have optimized components such as retrieval source (Yan et al., 2024; Li et al.), retriever (Asai et al., 2023; Jiang et al., 2023c; He et al., 2024) , query optimization (Dhuliawala et al.; Zhou et al.), context curation (Yu et al., 2023b; Ma et al.; Cui et al., 2023; Chevalier et al., 2023) and so on. Long context is a common issue in NLP tasks, and many approaches focus on context compression (Rau et al., 2024; Jiang et al., 2023b; Pan et al., 2024; Cheng et al., 2024d) or noise reduction (Zhang et al., 2024; Chen and Shu, 2024; Yu et al., 2023a; Xu et al., 2024) to improve performance.

However, code is a special text sequence, and many code tokens have no meaning in NLP. Many compression methods cannot retain the information required for the LLM to complete a certain piece of code. At the same time, many large-scale compressors incur additional time costs. This paper explores how to use lightweight models to compress code for LLMs.

3 Methodology

3.1 Definition of RAG

In repository-level code completion, we refer to the standard RepoCoder (Zhang et al., 2023) workflow and followed the main steps of chunking, indexing, retrieval, and generation. The naive RAG pipeline consists of the following three core components:

Retrieval database. In repository-level code completion, the external database is composed of code files from private code repositories. Following the RepoCoder (Zhang et al., 2023), we divided the code files in the repository into chunks of fixed granularity (e.g., every 10 lines of code). These chunks are then embedded into a vector space and stored in a database.

Retriever. The retriever's task is to search the database to identify code chunks most similar to the current unfinished code. To ensure that the query (unfinished code) and the code chunks stored in the retrieval database are in the same semantic space, the same encoder is applied to encode the query. Subsequently, by computing the cosine similarity of their embeddings, the top-k code chunks with the highest similarity to the query are retrieved.

Generator. The generator is typically a LLM. In RAG pipeline, the generator's inputs are the retrieved code chunks and the code context to be completed. The generator performs the code completion task in an autoregressive manner.

$$Input = [C_R] \texttt{<pad>} [Re_{code}] \texttt{<pad>} [C_L]$$

where C_R is the context code in right, Re_code is retrieved code chunks, C_L is the context code in left .

3.2 Design of Semantic Compressor Encoder

Our objective is to enhance code completion efficiency by shortening the context while maintain-

ing superior performance. To attain this objective, we put forward RepoLC. It utilizes a lightweight module to serve the LLM, which is responsible for compressing the retrieved code chunks. The component we inserted is called the SCE, and it comprises two primary components:

- Encoder: This component initially transforms code chunks into high-level features, capturing their semantics. Subsequently, average pooling is applied to convert these features into configurable fixed-length tokens.
- Projector: It is a multi-layer linear neural network, used to align the output of the encoder with the semantic space of the LLM.

To enable SCE to operate within the RAG, we have devised a two-stage training strategy: semantic alignment and task alignment. The training strategy is shown in Figure 2. During the semantic alignment phase, the output of SCE is harmonized with the semantic space of the LLM. Subsequently, in the task alignment stage, SCE is integrated into the RAG pipeline. This integration allows the LLM to adapt to code completion based on compressed tokens. This configuration ensures that the tokens extracted by SCE preserves vital information for the LLM, thus improving the LLM's performance in code completion.

3.3 Semantic Alignment

We integrate two models, distinct in training corpus and model architecture, into a single pipeline, where the output of one model serves as the input for the other. Hence, this approach inevitably introduces various discrepancies, particularly semantic gaps. Specifically, an LLM may fail to accurately comprehend the high-level semantic information extracted by the other model. Additionally, significant details are often lost during the process of information compression. This semantic mismatch and information loss directly impair the overall system performance. Therefore, our core objective is to preserve the original information to the greatest extent possible.

To achieve this goal, we design a method in which SCE compresses code chunks into tokens that the LLM can understand. We concatenate the instruction, the compressed tokens, and the raw code chunk and input this concatenated data into the LLM. The precise prompt is detailed in Table 7. Our training objective is to make the LLM restore the original code chunks as accurately as possible based on the tokens compressed by SCE. Accordingly, we directly use the cross-entropy loss to maximize the similarity between the generation result of the LLM and the original code snippet. The calculation formula is

$$\mathcal{L}_{\text{recon}} = -E_{x\sim\mathcal{D}} \sum_{t} \log P_{\text{LLM}}(x_t \mid z_{<t}, SCE(chunks))$$

where x_t is the token at time step t given previous tokens $z_{<t}$ and the projected compressed embedding of the original retrieved $chunks$.

During the training process, we freeze the parameters of the LLM to keep the semantic space of the LLM frozen, and only optimize the parameters of SCE.

3.4 Task Alignment

After completing the semantic alignment phase, the LLM has the capability to generate original text using compressed high-level semantic representations. However, at this stage, the model's generative capacity is confined to restorative tasks and has not been aligned with specific repository-level code completion. Therefore, further task-alignment training is required to enable the LLM to integrate contextual information and the compressed tokens thereby more accurately accomplishing code completion tasks.

Due to the powerful in-context learning ability of LLMs, providing them with some relevant knowledge can significantly enhance their performance. However, in RepoLC, the form of input to the LLM has changed remarkably. It has shifted from directly inputting the raw code chunks to inputting a combination of compressed semantic representations and context. This change may be inconsistent with the original input form of the LLM, potentially leading to performance degradation.

To mitigate this contradiction, we train the SCE in the actual pipeline of RAG. In the first stage, we have obtained relatively ideal compressed representations. Therefore, we freeze the parameters of SCE. Leveraging low rank adaptation (LoRA), we opt to fine-tune only a few of the LLM's parameters. By doing so, we fine-tune the LLM to adapt it to soft-prompt based tasks while preventing the model from losing its generalization ability.

Given the provided code, we use a retriever to fetch the top-k similar code chunks from the retrieval database. These code chunks are respec-

tively compressed using the SCE to obtain multiple representations. Next, these compressed tokens are sorted according to their similarity before compression, thus forming the compressed external information. After that, we concatenate the right-context of the code, the compressed external information, the left-context, and the ground truth, input the concatenated content into the LLM, and train the LLM to complete the ground truth. In this stage, we still adopt the cross-entropy loss, and its formula is:

$$\mathcal{L}_{\text{LLM}}(\theta) = -\sum_{i=1}^{N} \log P(y_i|\text{context}, \mathbf{E}_{\text{comp}}; \theta)$$

where the context is the unfinished code and $\mathbf{E}_{\text{comp}}$ is the compressed retrieved information.

4 Experiment Setup

4.1 Datasets

For the experiment, we utilized three datasets: CrossCodeEval (Ding et al., 2024), RepoEval (Zhang et al., 2023), and CrossCodeLongEval (Di Wu, 2024). These datasets were selected due to their coverage of multiple programming languages and their focus on complex cross-file code completion tasks, providing a comprehensive benchmark to evaluate model performance across diverse programming scenarios. Specifically, for the semantic alignment stage, we used the top-10 retrieved code chunks from CrossCodeEval and RepoEval to train SCE. For task alignment, we trained at the function and chunk level on CrossCodeLongEval and validated the models on both CrossCodeEval and RepoEval.

4.2 Setup

In SCE, the encoder employs two small-parameter models: Bert (Devlin, 2018) and CodeBert (Feng et al., 2020). Bert is the foundational encoder model in NLP, while CodeBert is based on the more advanced RoBerTa (Liu, 2019) architecture and fine-tuned on code corpus. We used the base versions of both models. In fact, any encoder model can also be implemented within our framework. The reason for choosing these two models is that they are the most representative ones.

For the generator, we utilized various models with different pre-training methods and parameter sizes. The latest models likes Llama3.2-3B base (Dubey et al., 2024), while the traditional models include CodeGEN-Mono (2B, 6B) (Nijkamp et al., 2022) , Deepseek-Coder-6.7B-base (Guo et al., 2024), and CodeLlama-7B-hf (Roziere et al., 2023).

For the retriever, we directly relied on the retrieval results from previous works. RepoCoder uses a sparse bag-of-words model with the Jaccard index to calculate relevance, while CrossCodeEval employs UniXcoder as its retrieval model and measures relevance using cosine similarity.

For details of other parameter settings, versions, baseline descriptions, etc., they are shown in the Appendix A.

4.3 Evaluation Metrics

Consistent with established methodologies in code completion research, we assess our line and API completion datasets using two metrics: Exact Match (EM) and Edit Similarity (ES). The EM score, a binary indicator, assigns a value of 1 when the predicted code matches the ground truth code exactly, and 0 otherwise. In contrast, the ES score is computed as 1 minus the ratio of the edit distance to the maximum length of the two strings, yielding a value between 0 and 1, where 1 indicates a perfect match and values closer to 0 indicate greater dissimilarity.

5 Experimental Results and Analysis

5.1 Main Results

The comparative results of performance are presented in Table 1. In the performance evaluation conducted on the RepoEval and CrossCodeEval-python datasets, our results indicate that the inclusion of relevant code chunks from the repository significantly enhances the model's performance. Furthermore, SCE improves the representation provided to the LLMs, thereby augmenting the performance of the RAG model. This indicates that RepoLC can be applied to a wide range of LLM.

However, when evaluated on the CrossCodeEval dataset, the RepoLC fails to yield further improvements for the Deepseek-Coder and CodeLlama. Instead, it leads to adverse impacts on their performance. This observation suggests that despite the ineffectiveness of SCE in enhancing the performance of certain models, the information retrieved remains a critical factor influencing the overall performance of the models.

Model	Policy	RepoEval-Api		RepoEval-Line		CrossCodeEval-Python	
		EM	ES	EM	ES	EM	ES
Deepseek-Coder-6.7B	In-File	22.93	57.22	33.50	62.65	29.41	70.41
	RAG	27.13	58.24	37.68	63.32	33.47	73.14
	RepoLC	45.10	74.45	53.68	77.01	33.47	74.32
CodeGEN-2B	In-File	22.31	58.91	33.06	65.21	7.27	55.75
	RAG	33.56	65.11	44.81	72.83	13.13	61.51
	RepoLC	35.93	68.35	44.38	74.77	23.44	66.39
CodeGEN-6B	In-File	23.12	59.32	34.56	66.49	11.14	58.23
	RAG	35.12	65.91	46.18	72.86	18.01	64.16
	RepoLC	37.52	70.32	47.81	77.08	24.00	66.82
CodeLlama-7B	In-File	26.56	61.38	36.06	65.93	26.56	61.38
	RAG	32.68	71.45	44.06	68.86	32.68	63.95
	RepoLC	51.23	82.04	59.18	85.02	30.99	72.65
Llama3.2-3B-base	In-File	12.53	39.79	30.31	57.33	6.94	52.36
	RAG	14.81	39.93	36.37	60.32	8.10	53.76
	RepoLC	27.56	58.60	38.50	65.93	11.30	55.37

Table 1: Overall performance of 5 models across 3 methods on RepoEval and CrossCodeEval

Additionally, we observed that for the CodeGEN series models, the performance enhancement brought about by RepoLC on the RepoEval dataset is relatively modest, while the improvement on CrossCodeEval is notably more pronounced. In contrast, for the Llama series models, the transition from Infile to RAG and subsequently to RepoLC results in substantial performance gains on RepoEval, whereas the improvements on CrossCodeEval are more marginal. These discrepancies may stem from the distinct knowledge acquired by different models during their pretraining phases, suggesting that the introduction of higher-quality contextual information does not always guarantee enhanced performance.

5.2 Efficiency Evaluation

We employed the torch profiler to evaluate the CUDA Time (ms) and GFLOPs of three methods: In-File, RAG, and RepoLC across two datasets. The experiments were conducted on identical hardware, utilizing an NVIDIA H20 GPU and an Intel® Xeon® Platinum 8469C CPU. In these evaluations, CodeGEN-2B, operating in bfloat16 inference mode, served as the base LLM. The experimental results are presented in Table 2.

Although SCE introduces additional compression time, it reduces the context length of LLM inputs. When compared with the inference cost of the LLM, the cost of the small models used in SCE is negligible. Experimental results demonstrate that RepoLC accelerates the RAG inference process by reducing both CUDA time and GFLOPs. In fact, RepoLC even approaches the inference time of Infile, showcasing its efficiency.

Furthermore, while the improvements from RepoLC in API completion are limited, it exhibits significant efficiency advantages in line-level completion tasks, where inference is based on an autoregressive model. It is evident that, as the number of tokens to be generated increases, the enhancement brought about by RepoLC becomes more pronounced.

5.3 Compare with Other RAG Method

We experiment with RepoLC and other RAG methods on CodeGEN and CodeLlama. The results are in Table 3. Other methods include using the LLM itself for soft-prompt compression, the LLMLingua series for hard-prompt compression, as well as RepoLC and RAG + FT. The experimental results show that RepoLC significantly outperforms other compression methods, demonstrating the necessity of our training scheme.

In addition, when compare with RAG + FT, we find that by directly compressing code chunks to 32 tokens, even when reducing the context length, RepoLC does not lose much accuracy compared with other methods. This verifies the applicability of RepoLC in repository-level code completion.

	CUDA Time (ms)			GFLOPS		
	In-File	RAG	RepoLC	In-File	RAG	RepoLC
RepoEval-Api	5913	6035	5987	8741	11993	10190
RepoEval-Line	3543	4400	3729	6090	9229	7386
CrossCodeEval-Python	3879	5022	4124	6545	9726	8101

Table 2: A Comparative Analysis of CUDA Time and GFLOPS across Different Settings. The generator is based on CodeGEN-2B. The SCE is based on CodeBert.

		RepoEval-Api		RepoEval-Line		CrossCodeEval-Python	
		EM	ES	EM	ES	EM	ES
CodeGEN-2B	RAG + FT	37.03	68.70	52.12	78.64	22.92	67.46
	RAG	33.56	65.11	44.81	72.83	13.13	61.51
	Self-Softprompt	14.13	51.56	21.13	56.39	2.44	46.94
	LLMLingua	33.44	67.37	42.18	71.58	21.05	64.95
	LLMLingua2	32.50	66.21	41.06	70.94	20.67	64.54
	RepoLC	35.93	68.35	44.38	74.77	23.44	66.39
CodeLlama-7B	RAG + FT	52.43	82.91	63.56	86.37	31.90	74.11
	RAG	32.68	71.45	44.06	68.86	32.68	63.95
	Self-Softprompt	18.31	53.33	23.56	55.69	8.56	57.38
	LLMLingua	34.56	64.88	39.13	66.39	26.86	69.26
	LLMLingua2	34.43	65.23	40.18	68.10	27.31	69.32
	RepoLC	51.15	82.04	59.18	85.02	30.99	72.65

Table 3: Comparison of SCE and other RAG methods

The other soft-prompt method is pre-trained on other corpus and is also not a plug-and-play method to serve any LLM. Due to resource limitations, we are unable to reproduce their method on different code models. However, we also attempt their base generator, and the results are shown in the Appendix C.

5.4 Ablation Study

In Table 4, we analyze the impact of different training steps on the overall RepoLC and the different combinations of training objects. The results show that if stage 1 is skipped and the parameters of the LLM are directly adjusted, poor results will be obtained. This indicates that due to the limited benefits of fine-tuning LLMs, great results cannot be achieved based on poor SCE representations. If stage 2 is omitted, the LLM only performs code completion based on the model aligned in the first stage, which significantly degrades the model's performance. This occurs because the LLM is unaware of how to utilize soft-prompt for downstream tasks.

However, we also discover a more lightweight approach: adjusting only the SCE without tuning the LLM still leads to performance improvement. This suggests that modifying the SCE module alone can bring substantial improvements, without the need for additional LLM training.

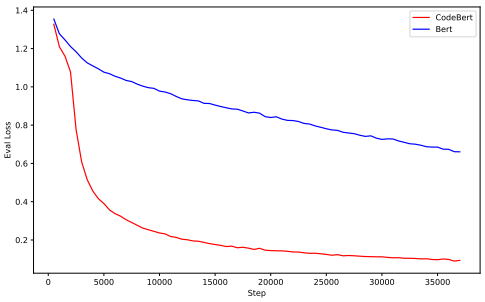
Finally, even after completing the semantic alignment in the first stage, further adjustments to both the SCE and LLM in downstream tasks do not result in additional performance gains. This is because, after the first stage of semantic alignment, the vector representation output by the SCE module has already reached a near optimal state. Continuing to adjust the SCE parameters would cause the model to lose its optimal solution.

5.5 Different Encoder Models

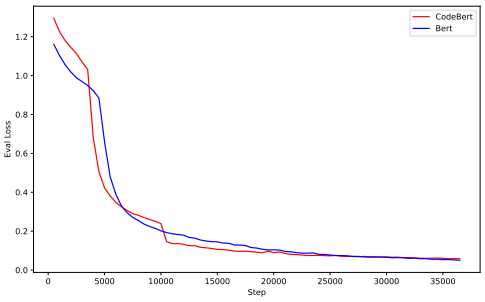
We conduct experiments using CodeBert and Bert as the base models for encoders, observing the performance of different generators. The percentage difference are shown in Figure 4, while the validation loss during the semantic alignment phase for CodeGEN-2B and CodeLlama is presented in Figure 3. The validation loss refers to the cross-entropy loss of the LLM calculated based on the first 500 samples of the entire aligned dataset.

Model	Policy	RepoEval-Api		RepoEval-Line		CrossCodeEval-Python	
		EM	ES	EM	ES	EM	ES
CodeGEN-2B	RepoLC	35.93	68.35	44.37	74.77	23.44	66.39
	W/o stage1	30.53	63.89	39.68	69.98	19.91	62.01
	W/o stage2	25.68	60.63	33.06	64.73	25.63	54.45
	Tuning SCE	33.06	66.56	40.31	70.39	22.12	65.82
	Tuning -SCE&LLM	34.31	68.21	43.12	73.98	22.46	65.53

Table 4: Ablation study and exploration of the SCE training strategy



(a) CodeGEN-2B



(b) CodeLlama

Figure 3: The figure shows the loss variations of diverse models during the first stage when distinct encoders are employed for semantic alignment. In fact, on most models, the alignment effect of CodeBert is better than that of Bert.

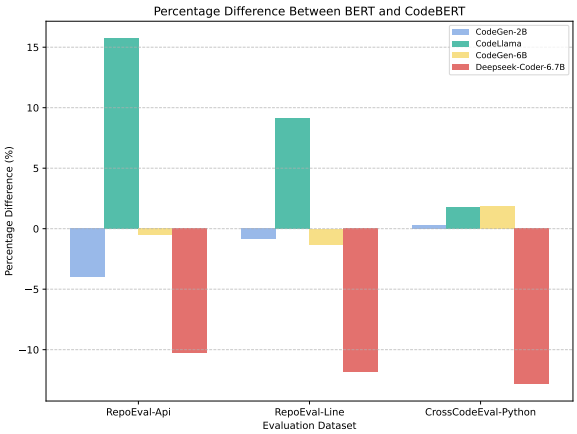


Figure 4: The performance comparison of various models across different datasets, using different encoders, shows that CodeLlama performs better when paired with Bert than with CodeBert. In most cases, however, Bert and CodeBert exhibit similar performance, and in some instances, CodeBert outperforms Bert.

During the semantic alignment phase, the majority of models adhere to a training performance analogous to that of CodeGEN-2B. CodeBert demonstrates the ability to effectively harmonize the semantic spaces of the two models. In contrast, Bert encounters challenges in achieving this alignment. The validation loss of Bert reaches a plateau after declining to a relatively elevated level. Nevertheless, during the subsequent task alignments, by capitalizing on its generalizability as a universal model, BERT showcases commendable performance. Consequently, its code completion outcomes are on par with those of CodeBert. Intriguingly, the code completion efficacy of CodeLlama, which is founded on Bert, has witnessed a substantial improvement in comparison to CodeBert. Significantly, in the initial phase of CodeLlama, the loss incurred when employing Bert is comparable to that of CodeBert, surpassing other models. In the second phase, the more adaptable Bert can be trained to generate superior representations, thereby enhancing the code completion performance of CodeLlama.

6 Conclusion

This paper introduces RepoLC, which utilizes a lightweight compressor for repository-level code completion. To enable the LLM to complete code using fewer tokens compressed by the SCE, a two-phase training strategy has been meticulously formulated. The experimental results on the CrossCodeEval and repoeval datasets have convincingly demonstrated that RepoLC exhibits promising performance and remarkable efficiency. Additionally, RepoLC is characterized by its good generalization ability and lightweight, making it a practical and effective solution in the realm of code completion.

7 Limitations

RepoLC depends on SCE to compress more advanced semantic features. Therefore, the performance of RepoLC may rely on the representational ability of SCE, which might also be determined by the generalization level of the training data. In our study, the semantic alignment phase is trained within the repository, while the task alignment is trained outside the repository. According to our experiments, RepoLC has demonstrated cross language generalization. However, for practical applications, we suggest making adjustments within a private repository to achieve better results.

SCE does not possess the generalizability to be directly transferred between generators. Different LLMs have distinct semantic spaces, and there are fundamental differences in their hidden dimensions. Technically, direct transfer is unfeasible. Nevertheless, generally speaking, the problem can be alleviated by simply retraining a projector to align different semantic spaces.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2023. Self-rag: Learning to retrieve, generate, and critique through self-reflection. *arXiv preprint arXiv:2310.11511*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Canyu Chen and Kai Shu. 2024. Combating misinformation in the age of llms: Opportunities and challenges. *AI Magazine*, 45(3):354–368.

Wei Cheng, Yuhan Wu, and Wei Hu. 2024a. Dataflow-guided retrieval augmentation for repository-level code completion. *arXiv preprint arXiv:2405.19782*.

Wei Cheng, Yuhan Wu, and Wei Hu. 2024b. Dataflow-guided retrieval augmentation for repository-level code completion. *arXiv preprint arXiv:2405.19782*.

Xin Cheng, Di Luo, Xiuying Chen, Lemao Liu, Dongyan Zhao, and Rui Yan. 2024c. Lift yourself up: Retrieval-augmented text generation with self-memory. *Advances in Neural Information Processing Systems*, 36.

Xin Cheng, Xun Wang, Xingxing Zhang, Tao Ge, Si-Qing Chen, Furu Wei, Huishuai Zhang, and Dongyan Zhao. 2024d. xrag: Extreme context compression for retrieval-augmented generation with one token. *arXiv preprint arXiv:2405.13792*.

Alexis Chevalier, Alexander Wettig, Anirudh Ajith, and Danqi Chen. 2023. Adapting language models to compress contexts. *arXiv preprint arXiv:2305.14788*.

Jiaxi Cui, Zongjian Li, Yang Yan, Bohua Chen, and Li Yuan. 2023. Chatlaw: Open-source legal large language model with integrated external knowledge bases. *CoRR*.

Jacob Devlin. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason E Weston. Chain-of-verification reduces hallucination in large language models. In *ICLR 2024 Workshop on Reliable and Responsible Foundation Models*.

Dejiao Zhang Murali Krishna Ramanathan Xiaofei Ma Di Wu, Wasi Uddin Ahmad. 2024. Repoformer: Selective retrieval for repository-level code completion.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. 2024. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, 36.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Wenqi Fan, Yujuan Ding, Liangbo Ning, Shijie Wang, Hengyun Li, Dawei Yin, Tat-Seng Chua, and Qing Li. 2024. A survey on rag meeting llms: Towards retrieval-augmented large language models. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 6491–6501.

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. *arXiv preprint arXiv:2002.08155*.

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

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

Tao Ge, Hu Jing, Lei Wang, Xun Wang, Si-Qing Chen, and Furu Wei. In-context autoencoder for context compression in a large language model. In *The Twelfth International Conference on Learning Representations*.

Linyuan Gong, Mostafa Elhoushi, and Alvin Cheung. 2024. Ast-t5: Structure-aware pretraining for code generation and understanding. *arXiv preprint arXiv:2401.03003*.

Zhanming Guan, Junlin Liu, Jierui Liu, Chao Peng, Dexin Liu, Ningyuan Sun, Bo Jiang, Wenchao Li, Jie Liu, and Hang Zhu. 2024. Contextmodule: Improving code completion via repository-level contextual information. *arXiv preprint arXiv:2412.08063*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Rujun Han, Yuhao Zhang, Peng Qi, Yumo Xu, Jenyuan Wang, Lan Liu, William Yang Wang, Bonan Min, and Vittorio Castelli. 2024. Rag-qa arena: Evaluating domain robustness for long-form retrieval augmented question answering. *arXiv preprint arXiv:2407.13998*.

Xiaoxin He, Yijun Tian, Yifei Sun, Nitesh V Chawla, Thomas Laurent, Yann LeCun, Xavier Bresson, and Bryan Hooi. 2024. G-retriever: Retrieval-augmented generation for textual graph understanding and question answering. *arXiv preprint arXiv:2402.07630*.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023a. Mistral 7b. *arXiv preprint arXiv:2310.06825*.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023b. LLMLingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13358–13376, Singapore. Association for Computational Linguistics.

Zhengbao Jiang, Frank F Xu, Luyu Gao, Zhiqing Sun, Qian Liu, Jane Dwivedi-Yu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023c. Active retrieval augmented generation. *arXiv preprint arXiv:2305.06983*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474.

Jian Li, Yue Wang, Michael R Lyu, and Irwin King. 2017a. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573*.

Jian Li, Yue Wang, Michael R Lyu, and Irwin King. 2017b. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573*.

Xiaoqian Li, Ercong Nie, and Sheng Liang. From classification to generation: Insights into crosslingual retrieval augmented icl. In *NeurIPS 2023 Workshop on Instruction Tuning and Instruction Following*.

Fang Liu, Ge Li, Bolin Wei, Xin Xia, Zhiyi Fu, and Zhi Jin. 2020. A self-attentional neural architecture for code completion with multi-task learning. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 37–47.

Junwei Liu, Yixuan Chen, Mingwei Liu, Xin Peng, and Yiling Lou. 2024a. Stall+: Boosting llm-based repository-level code completion with static analysis. *arXiv preprint arXiv:2406.10018*.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024b. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173.

Yinhan Liu. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 364.

Yubo Ma, Yixin Cao, Yong Ching Hong, and Aixin Sun. Large language model is not a good few-shot information extractor, but a good reranker for hard samples! In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A conversational paradigm for program synthesis. *arXiv preprint*.

Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Menglin Xia, Xufang Luo, Jue Zhang, Qingwei Lin, Victor Rühle, Yuqing Yang, Chin-Yew Lin, H. Vicky Zhao, Lili Qiu, and Dongmei Zhang. 2024. LLMLingua-2: Data distillation for efficient and faithful task-agnostic prompt compression. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 963–981, Bangkok, Thailand. Association for Computational Linguistics.

Huy N Phan, Hoang N Phan, Tien N Nguyen, and Nghi DQ Bui. 2024. Repohyper: Better context retrieval is all you need for repository-level code completion. *arXiv preprint arXiv:2403.06095*.

David Rau, Shuai Wang, Hervé Déjean, and Stéphane Clinchant. 2024. Context embeddings for efficient answer generation in rag. *arXiv preprint arXiv:2407.09252*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Disha Shrivastava, Denis Kocetkov, Harm de Vries, Dzmitry Bahdanau, and Torsten Scholak. 2023. Repofusion: Training code models to understand your repository. *arXiv preprint arXiv:2306.10998*.

Ze Tang, Jidong Ge, Shangqing Liu, Tingwei Zhu, Tongtong Xu, Liguo Huang, and Bin Luo. 2023. Domain adaptive code completion via language models and decoupled domain databases. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 421–433. IEEE.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Ante Wang, Linfeng Song, Ge Xu, and Jinsong Su. 2023. Domain adaptation for conversational query production with the rag model feedback. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 9129–9141.

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*.

Qinyun Wu, Chao Peng, Pengfei Gao, Ruida Hu, Haoyu Gan, Bo Jiang, Jinhe Tang, Zhiwen Deng, Zhanming Guan, Cuiyun Gao, et al. 2024. Repomastereval: Evaluating code completion via real-world repositories. *arXiv preprint arXiv:2408.03519*.

Shicheng Xu, Liang Pang, Mo Yu, Fandong Meng, Huawei Shen, Xueqi Cheng, and Jie Zhou. 2024. Unsupervised information refinement training of large language models for retrieval-augmented generation. *arXiv e-prints*, pages arXiv–2402.

Shi-Qi Yan, Jia-Chen Gu, Yun Zhu, and Zhen-Hua Ling. 2024. Corrective retrieval augmented generation. *arXiv preprint arXiv:2401.15884*.

Ori Yoran, Tomer Wolfson, Ori Ram, and Jonathan Berant. 2023. Making retrieval-augmented language models robust to irrelevant context. *arXiv preprint arXiv:2310.01558*.

Wenhao Yu, Hongming Zhang, Xiaoman Pan, Kaixin Ma, Hongwei Wang, and Dong Yu. 2023a. Chain-of-note: Enhancing robustness in retrieval-augmented language models. *arXiv preprint arXiv:2311.09210*.

Zichun Yu, Chenyan Xiong, Shi Yu, and Zhiyuan Liu. 2023b. Augmentation-adapted retriever improves generalization of language models as generic plug-in. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2421–2436.

Shenglai Zeng, Jiankun Zhang, Pengfei He, Yue Xing, Yiding Liu, Han Xu, Jie Ren, Shuaiqiang Wang, Dawei Yin, Yi Chang, et al. 2024. The good and the bad: Exploring privacy issues in retrieval-augmented generation (rag). *arXiv preprint arXiv:2402.16893*.

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. *arXiv preprint arXiv:2303.12570*.

Tianjun Zhang, Shishir G Patil, Naman Jain, Sheng Shen, Matei Zaharia, Ion Stoica, and Joseph E Gonzalez. 2024. Raft: Adapting language model to domain specific rag. *arXiv preprint arXiv:2403.10131*.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V Le, et al. Least-to-most prompting enables complex reasoning in large language models. In *The Eleventh International Conference on Learning Representations*.

Wei Zou, Runpeng Geng, Binghui Wang, and Jinyuan Jia. 2024. Poisonedrag: Knowledge poisoning attacks to retrieval-augmented generation of large language models. *arXiv preprint arXiv:2402.07867*.

A Implementation Details

In this work, the construction of the retriever and retrieval library is completely based on the retrieval results of Repoformer and RepoCoder. For the generator part, the transformers library with version 4.46.1 is adopted for loading and inference. Given the limitations of hardware conditions, data loading in both the training and inference processes is carried out in the BFloat16 format. In the first stage, a full-scale tuning of the parameters of SCE is performed. Specifically, methods such as average pooling and pad (padding) are employed to map matrices of any dimension to the specified dimension.

Regarding the generation parameters, the most basic settings are adopted, and only the max_new_tokens parameter is set to the length of the code to be completed plus 10. For the second stage, the LoRA is utilized to fine-tune LLMs. The PEFT (Parameter Efficient Fine Tuning) library with version 0.12.0 is used to load the LoRA configuration. Among them, the rank of the matrix is set to 16, the scaling factor of LoRA is set to 32, the LoRA dropout is set to 0.1, and the remaining parameters adopt the default settings. The number of layers of the projection we set is three.

In the second phase's task alignment stage, the overall length is set to 2048. In particular, the left-context truncation is set to 1024, and the right-context is set to 1024 minus the product of the compressed length and top-K. In the RAG method, the settings are consistent with previous studies. Specifically, the right-context truncation is set to 512, the retrieved code chunks are truncated to 512, and the left-context truncation is 1024. The entire experiment is conducted in a 2 * H20 hardware environment.

B Robustness in Different Programming Languages

In our approach, both semantic alignment and task alignment are based on the Python programming language, and therefore, the main experiments are conducted using Python. We further investigate the effectiveness of models trained in Python when applied to other programming languages, with the experimental results presented in the Table 5. From the results, it is evident that SCE is an effective enhancement method with strong cross-language generalization capabilities, significantly improving the performance of code generation models, particularly for languages such as C# and TypeScript. While the introduction of RAG leads to some performance improvement, SCE provides superior results when dealing with complex languages and tasks. For the CodeGEN-2B model, although SCE enhances its performance on EM and ES, the overall improvement remains limited, especially in EM, where the performance increase is still relatively low. This suggests that CodeGEN-2B may not have fully mastered the generation of high-quality code during pretraining, despite the improvements brought by SCE. The characteristics of different languages, such as syntax complexity and standard libraries, may affect the model's performance. For instance, languages like TypeScript and Java, due to their unique type systems and syntax, may present additional challenges in generating accurate code.

C Baseline Method Details

The method details are as follows:

- LLMlinuga (Jiang et al., 2023b): It utilizes compact and well - trained language models such as GPT2 - small and LLaMA - 7B to identify and remove unnecessary tokens in prompts. This approach enables efficient inference using large language models (LLMs), achieving a compression ratio of up to 20 times while minimizing performance degradation.

- LLMlingua2 (Pan et al., 2024): It is a compact yet powerful and fast compression method. It is trained through data distillation from GPT-4 (Achiam et al., 2023) and uses a BERT-level encoder for token classification, demonstrating excellent performance in task-agnostic compression. It outperforms LLMLingua in handling out-of-domain data, with a performance improvement of 3 to 6 times.

- ICAE (Ge et al.) : In-Context Auto-Encoder is pre-trained on a large amount of text data using auto-encoding and language-modeling objectives. This enables it to generate memory slots that can accurately and comprehensively represent the original context. Subsequently, it is fine-tuned based on the instruction data to produce desirable responses to various prompts. The lightweight ICAE introduces approximately 1% additional parameters, effectively achieving 4× context compression based on Llama. It has the advantages of

CrossCodeEval		C#		TypeScript		JAVA	
		EM	ES	EM	ES	EM	ES
CodeLlama	In-File	20.47	66.69	11.02	66.66	24.82	68.81
	RAG	22.11	67.86	12.33	68.29	28.42	69.56
	SCE	46.26	84.72	34.11	78.71	35.20	77.43
CodeGEN-2B	In-File	0.96	39.45	1.63	48.40	6.40	55.98
	RAG	3.57	50.65	2.03	44.01	9.81	58.61
	SCE	11.20	57.21	12.91	53.71	13.97	59.77

Table 5: The effectiveness of migrating frameworks trained on the Python language to other programming languages.

improved latency and reduced GPU memory costs during the inference process.

- Self-Softprompt: This method is intuitively designed by us. LLM should be able to directly understand their own outputs. LLM use the self-attention mechanism for global modeling at each layer, while pooling only models the nearby semantics. Therefore, we explore whether LLMs can pool their own hidden states for compression.

Beyond the descriptions presented in the main body of the text, additional experiments have been incorporated into the Table 6. Concurrently, the experimental results for ICAE are depicted in the Figure 5. Owing to resource constraints, the replication of their methods across every model was unfeasible. As a consequence, our method was tested on their base-generator: Mistral-7B (Jiang et al., 2023a). Notwithstanding these limitations, it can be inferred that RepoLC continues to exhibit outstanding competitiveness.

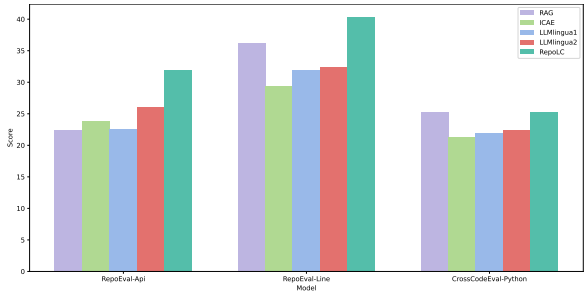


Figure 5: The experimental results are based on the generator of ICAE and the score is EM. In most dataset, the performance of RepoLC is leading. Moreover, we find that the ICAE method is even inferior to the hard-prompt method.

D Stage 1 Details

D.1 Case of Stage 1

In the first phase of training for SCE, we focus on the integration of SCE with LLMs to effectively compress and reconstruct code. The goal of this phase is to train the LLM to correctly regenerate the original code snippets from the semantic embeddings compressed by SCE. The MODEL INPUT section describes how the original code is compressed into semantic embeddings using SCE, which are then fed into the LLM to generate code. Specific examples illustrate the input format, which includes semantic embeddings ([<pad>* N]) and corresponding text prompts. However, the actual code represented by these embeddings is not described in detail.

The MODEL ANSWERS section presents two specific code generation results, demonstrating how the LLM is able to reconstruct detailed code snippets based on the compressed semantic embeddings. From this, it becomes evident that when training with the CodeGEN-2B and BERT combination, this pairing may not fully achieve the desired training outcomes, particularly in terms of the accuracy and completeness of the code reconstruction. In contrast, when using CodeBert as the encoder and CodeLlama as the generation model, the code is successfully reconstructed to be fully consistent with the ground truth. This indicates that CodeBert and CodeLlama are more effective in understanding and applying the semantic embeddings compressed by SCE, thus more accurately restoring the original code's logic and structure.

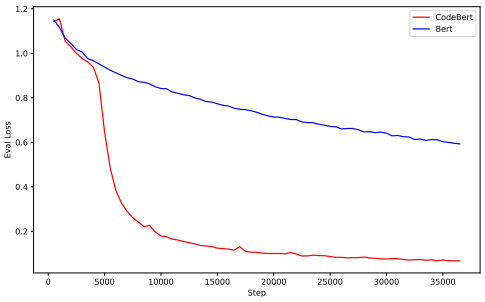
This disparity can be attributed to differences in model size and the variations in training data used during pretraining, which influence the semantic understanding and the ability to restore compressed information. These findings underscore the importance of carefully considering the characteristics

		RepoEval-Api		RepoEval-Line		CrossCodeEval-Python	
		EM	ES	EM	ES	EM	ES
Deepseek-Coder-6.7B	In-File	22.93	57.22	33.5	62.65	29.41	70.41
	RAG	27.10	58.24	37.68	63.32	33.47	73.14
	Self-Softprompt	34.56	62.93	39.88	65.11	31.55	71.71
	LLMLingua	35.25	66.83	40.94	67.78	30.24	71.12
	LLMLingua2	27.31	55.19	35.93	60.59	30.31	71.13
	RepoLC	45.13	74.45	53.68	77.01	33.47	74.32
Llama-3.2B	In-File	12.52	39.79	30.31	57.33	6.94	52.36
	RAG	14.81	39.93	36.37	60.32	8.10	53.76
	Self-Softprompt	20.75	46.77	30.25	57.17	12.02	53.52
	LLMLingua	15.25	37.33	32.36	57.40	11.18	53.11
	LLMLingua2	14.25	36.02	30.68	55.22	11.63	53.07
	RepoLC	27.56	58.60	38.5	65.93	11.30	55.37

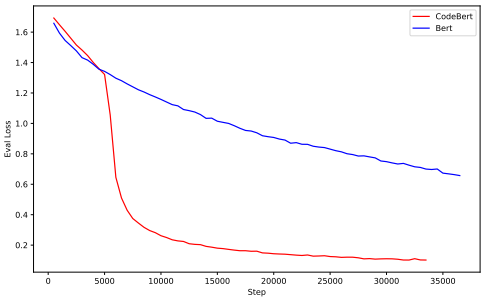
Table 6: Supplementary results of using different compression methods on the performance of other models.

and compatibility of different model combinations when configuring SCE, to ensure that the resulting model is capable of effectively leveraging the compressed code information for accurate and efficient code generation.

D.2 Supplementary Stage 1 Loss



(a) Deepseek-Coder-6B loss in stage 1



(b) llama3.2-3B loss in stage 1

Figure 6: Most models exhibit behavior similar to that of (a), with CodeBert demonstrating better training performance. However, CodeLlama is an exception. In the case of CodeLlama, the performance of CodeBert and Bert is very close.

In the main text, we discussed and experimented on the influence of different encoder models on the entire pipeline and stage 1. Here, we supplement with additional loss data of stage 1. It is the loss performance of Deepseek-Coder-6B and Llama3.2 in the first stage. It can be observed that CodeBert shows more stability during training, with a faster loss decline, indicating a better understanding of code data and more effective semantic compression and task alignment. In contrast, BERT performs poorly in stage 1 because of the lack of code fine-tuning, leading to a slower loss reduction in the semantic alignment process.

MODEL INPUT
The semantic embedding are [<pad>* N]. The text content of the embedding is:
MODEL ANSWERS
CodeGEN-2B+Bert
the below code fragment can be found in: example_chatbot.py model = ExLlamaChatbot(tokenizer, config) # print(model) # print(model.model) # print(model.tokenizer) # print(model.config) # print(model.model.config) # print(model.model.config.to_dict()) # print(model.model.config.to_json_string()) # print(model.model.config.to_json_file()) # print(model.model.config.to_yaml_string()) # print(model.model.config.to_yaml_file()) # print(model.model.config.to_dict())
CodeLlama+CodeBert & CodeGEN-2B+CodeBert & CodeLlama+Bert

the below code fragment can be found in:
example_chatbot.py
lora = ExLlamaLora(model, args.lora_config, args.lora)
if lora.bias_ignored:
print(f" !! Warning: Loading zero bias Lora zero ")
# Generator 	
generator = ExLlamaGenerator(model, tokenizer, cache)
generator.settings = ExLlamaGenerator.Settings()
generator.settings.temperature = args.temperature
generator.settings.top_k = args.top_k
generator.settings.top_p = args.top_p
generator.settings.min_p = args.min_p

GROUND TRUTH

the below code fragment can be found in:
example_chatbot.py
lora = ExLlamaLora(model, args.lora_config, args.lora)
if lora.bias_ignored:
print(f" !! Warning: Loading zero bias Lora zero ")
# Generator 	
generator = ExLlamaGenerator(model, tokenizer, cache)
generator.settings = ExLlamaGenerator.Settings()
generator.settings.temperature = args.temperature
generator.settings.top_k = args.top_k
generator.settings.top_p = args.top_p
generator.settings.min_p = args.min_p

Table 7: The case for stage1 in different generator.

Raw code chunks (263 tokens):

the below code fragment can be found in:
alt_generator.py
# stop_conditions: List of strings or integer token IDs that will end the sequence 	
# settings: ExLlamaAltGeneratorSettings 	
# encode_special_characters: Set to true to tokenize "</s>" etc. 	
def begin_stream(self, prompt: str, stop_conditions: list, max_new_tokens: int, gen_settings: Settings,
encode_special_characters = False):
assert isinstance(prompt, str), "ExLlamaAltGenerator does not support batched generation"
# Tokenize prompt and limit length to allow prompt and (max) new tokens within max sequence length 	
max_input_tokens = self.model.config.max_seq_len - max_new_tokens
self.remaining_tokens = max_new_tokens
input_ids = self.cached_tokenize(prompt, encode_special_characters)
applied_input_ids = input_ids[:, -max_input_tokens:]

LLMlingua1 compressed (140 tokens) :
the below code fragment be in#_# #___itions: of strings or integer token IDs that will the#: ExLlama#__: Set to toize ">" def___, prompt:,_,_:acters "ator support # and limit to allow prompt and (max tokens within max __._ -_tokens self.remaining_tokens = max_new_tokens input_ids = self.cached_tokenize(prompt, encode_special_characters) applied_input_ids = input_ids[:, -max_input_tokens:]
LLMlingua2 compressed (102 tokens):
code alt_generator.py stop_conditions token IDs sequence settings ExLlamaAltGeneratorSettings encode_special_characters tokenize def begin_stream stop_conditions max_new_tokens_settings encode_special_characters assert support batched generation Tokenize prompt limit length tokens sequence length max_input_tokens _tokens input_ids_tokenize applied_input_ids_tokens

Table 8: The case for LLMlingua-series to compress the retrieved code chunks.