Hierarchical Context Pruning: Optimizing Real-World Code Completion with Repository-Level Pretrained Code LLMs

Lei Zhang[1,2,3], **Yunshui Li**[1,2], **Jiaming Li**[1,2], **Xiaobo Xia**[4,5], **Jiaxi Yang**[1,2]
Run Luo[1,2], **Minzheng Wang**[2,7], **Longze Chen**[1,2], **Junhao Liu**[6], **Qiang Qu**[1], **Min Yang**[1,3*]

[1]Shenzhen Key Laboratory for High Performance Data Mining, Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences
[2]University of Chinese Academy of Sciences
[3]Key Laboratory of Intelligent Education Technology and Application of Zhejiang Province, Zhejiang Normal University
[4]School of Computing, National University of Singapore
[5]University of Science and Technology of China
[6]University of California, Irvine
[7]MAIS, Institute of Automation, Chinese Academy of Sciences
{lei.zhang2, min.yang}@siat.ac.cn

Abstract

Some of the latest released Code Large Language Models (Code LLMs) have been trained on repository-level code data, enabling them to perceive repository structures and utilize cross-file code information. This capability allows us to directly concatenate the content of repository code files in prompts to achieve repository-level code completion. However, in real development scenarios, directly concatenating all code repository files in a prompt can easily exceed the context window of Code LLMs, leading to a significant decline in completion performance. Additionally, overly long prompts can increase completion latency, negatively impacting the user experience. In this study, we conducted extensive experiments, including completion error analysis, topology dependency analysis, and cross-file content analysis, to investigate the factors affecting repository-level code completion. Based on the conclusions drawn from these preliminary experiments, we proposed a strategy called **Hierarchical Context Pruning (HCP)** to construct high-quality completion prompts. We applied the **HCP** to six Code LLMs and evaluated them on the CrossCodeEval dataset. The experimental results showed that, compared to previous methods, the prompts constructed using our **HCP** strategy achieved higher completion accuracy on five out of six Code LLMs. Additionally, the **HCP** managed to keep the prompt length around 8k tokens (whereas the full repository code is approximately 50k tokens), significantly improving completion throughput. Our code and data will be publicly available.

Code — https://github.com/Hambaobao/HCP-Coder
Extended version — https://arxiv.org/abs/2406.18294

Introduction

Code completion tools powered by Code Large Language Models (Code LLMs) (Chen et al. 2021; Nijkamp et al. 2023b; Li et al. 2023; Fried et al. 2023; Allal et al. 2023), such as *GitHub Copilot*, have become integral to daily development workflows, significantly boosting developer productivity. As research on Code LLMs continues to advance (Bavarian et al. 2022; Sun et al. 2024), a new generation of models (Guo et al. 2024; Lozhkov et al. 2024; Team et al. 2024) trained on repository-level code data (Repo-Code LLMs) has emerged. These models address the shortcomings of earlier file-level models, which struggled to comprehend repository structures and integrate code across multiple files during completion tasks. However, in real-world development scenarios, directly concatenating an entire code repository often exceeds the context window size of these Repo-Code LLMs, resulting in significant performance degradation and increased inference latency. Thus, effectively harnessing the capabilities of Repo-Code LLMs to integrate cross-file information and construct high-quality completion prompts within the model's context window remains an open challenge for further research.

In this study, we first evaluated six Repo-Code LLMs on the CrossCodeEval (Ding et al. 2023) benchmark by constructing completion prompts through randomly concatenating all repository code. We conducted a detailed analysis and categorization of the erroneous completions produced by these models. The analysis revealed that at least 30% of the errors were caused by cross-file information issues, highlighting the importance of constructing high-quality repository-level completion prompts. Next, considering the causal architecture characteristics of Code LLMs, we performed a topological sort of code files based on their dependency relationships and conducted topological dependency analysis experiments. The results showed that maintaining the topological dependency order of code files within the prompt significantly improved the accuracy of code completions. Finally, we analyzed cross-file code content through experiments and found that global context information in cross-file code can be completely removed without affecting completion accuracy. Even when the specific implementations of all functions and class methods in cross-file code

*Min Yang is the corresponding author.

are removed, there is no significant reduction in completion accuracy.

Based on the results of these preliminary experiments, we proposed a strategy named **Hierarchical Context Pruning (HCP)** to construct high-quality completion prompts. The **HCP** models the code repository at the function level, retaining the topological dependencies between files while eliminating a large amount of irrelevant code content. In our experiments, the **HCP** successfully reduced the input from over 50,000 tokens to approximately 8,000 tokens, and significantly enhanced the accuracy of completions.

In summary, our contributions are threefold:

- **Pioneering Studies Based on Repository-Aware Code LLMs:** To the best of our knowledge, our research is pioneering in exploring how to develop high-quality completion prompts by leveraging the repository structural awareness of Repo-Code LLMs.
- **Thorough Preliminary Experiments:** In this study, we first conducted extensive preliminary experiments, including baseline evaluation, completion error analysis, topological dependency analysis, and cross-file content analysis. The conclusions from these preliminary experiments collectively form the rationale behind the HCP.
- **Effective Method and Significant Improvements:** Our proposed HCP strategy achieved better results on five out of six Code LLMs compared to previous baseline methods. These experimental results demonstrate the effectiveness of the HCP strategy. We will make our code and data publicly available.

Experiments Setup

Dataset & Evaluation Metrics

To assess the code completion performance of Code LLMs in real development scenarios, we utilized CrossCodeEval (Ding et al. 2023) as the evaluation dataset. The CrossCodeEval (Ding et al. 2023) benchmark provides test cases that require the use of cross-file code information for completion. Without loss of generality, in this study, we have chosen `Python` language as the primary language for our research.

We used the original data from CrossCodeEval, retaining the original repository structure. For each test case, we first identified the file for completion and the cursor's position (the line and column where the completion occurs). We then removed the code after the cursor in that line to form authentic completion test cases. Ultimately, we obtained 2,655 real-world completion tests. Following the CrossCodeEval evaluation protocol, we evaluated the completion results using two metrics: *Exact Match* (EM) and *Edit Similarity* (ES). The results based on the Identifier Match (Identifier EM/Identifier F1) metrics can be found in the appendix.

Models & Prompt Templates

We selected three series of Repo-Code LLMs for investigation: DeepSeek-Coder (Guo et al. 2024), Starcoder-2 (Lozhkov et al. 2024), and CodeGemma (Team et al. 2024). The specific prompt templates used by these Repo-Code LLMs are presented in the appendix of the extended version.

Hardware & Hyperparameters

We conduct all the expiriments on NVIDIA A100 GPUs. We employ greedy decoding strategy for all the models, and set `max_new_tokens` to 32. The `model_max_length` of DeepseekCoder, Starcoder2 and CodeGemma is set to $16,352$, $16,352$ and $8,160$, respectively. All the prompts longer than the `model_max_length` are truncated from the left.

Preliminary Studies

Baseline Evaluation

Infile Only We initially evaluated the model's completion ability using only information from the current file, with results presented in Table 1 under the *Infile-Only* row. The completion results are less than satisfactory. Even the best-performing model achieved an accuracy of only about 30%.

Random-All Additionally, based on the pre-training data format of these Code LLMs, we constructed completion prompts by randomly concatenating all code files from the repository (truncating from the left to fit within the model's context window). The evaluation results are shown in Table 1 under the *Random-All* row. The experimental results indicate that including information from other files in the prompt significantly improved completion accuracy (with the exception of DScoder-1.3B). The performance of the DScoder-1.3B model progressively declines when the prompt includes multiple files, as can be seen in the experimental results in Appendix. We suspect this is likely due to the DScoder-1.3B model not being trained with a multi-file data format, rather than a reduction in the number of parameters.

RAG-Based Methods We also evaluated other retrieval-augmented generation (RAG) methods, including RAG-BM25 (using BM25 as the relevance metric), RAG-OpenAI (using OpenAI text embedding similarity as the relevance metric), ReAcc (Lu et al. 2022), DraCo (Cheng, Wu, and Hu 2024), and Repofuse (Liang et al. 2024a). The experimental results show that, for these Repo-Code LLMs, the completion accuracy of prompts constructed by simply concatenating repository code files randomly is comparable to that of these RAG-based methods. This highlights the potential of Repo-Code LLMs in repository-level code completion tasks.

Completion Error Analysis

To further investigate the issues of repository-level pre-trained Code LLMs in real-world completion tasks, we sampled 200 error examples from each model's *Random-All* evaluation results for error analysis. Ultimately, we categorized the issues present in these models into eight classes: *Parameter Value Error*, *Non-existent Method Call*, *Improper Method Invocation*, *Missing Method Invocation*, *Redundant Content Generation*, *Partial Content Missing*, *Incorrect Content Generation*, and *Exact Match Error*. We shows the error distribution statistics for six Repo-Code LLMs in the appendix of the extended version. We also provide examples of each type of error along with corresponding error analysis in the appendix.

XF-Context	Baseline Evaluation											
	DScoder-1.3B		DScoder-6.7B		Starcoder2-3B		Starcoder2-7B		CodeGemma-2B		CodeGemma-7B	
	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES
Infile-Only	16.72	56.58	28.14	68.36	21.92	61.49	22.98	63.58	20.64	56.26	30.58	70.36
Random-All	6.18	46.19	33.94	70.98	28.32	**66.87**	31.45	**69.09**	**26.93**	**62.13**	36.69	**74.42**
RAG-BM25	17.28	58.18	32.65	71.78	24.45	63.84	26.26	65.32	22.89	57.73	32.89	70.81
RAG-OpenAI	17.56	57.65	31.93	71.13	25.67	64.39	26.72	65.90	22.74	57.12	36.06	74.19
ReACC	16.08	55.51	32.36	70.60	24.79	63.59	26.54	64.39	22.80	56.56	35.67	73.47
DraCo	17.07	56.20	**37.06**	71.30	**29.83**	65.50	**31.71**	67.77	20.23	54.33	**40.79**	73.36
RepoFuse	**22.59**	**68.85**	27.92	**73.09**	-	-	-	-	-	-	-	-

Table 1: The completion results of the baseline methods. **EM** denotes Exact Match, and **ES** denotes Edit Similarity.

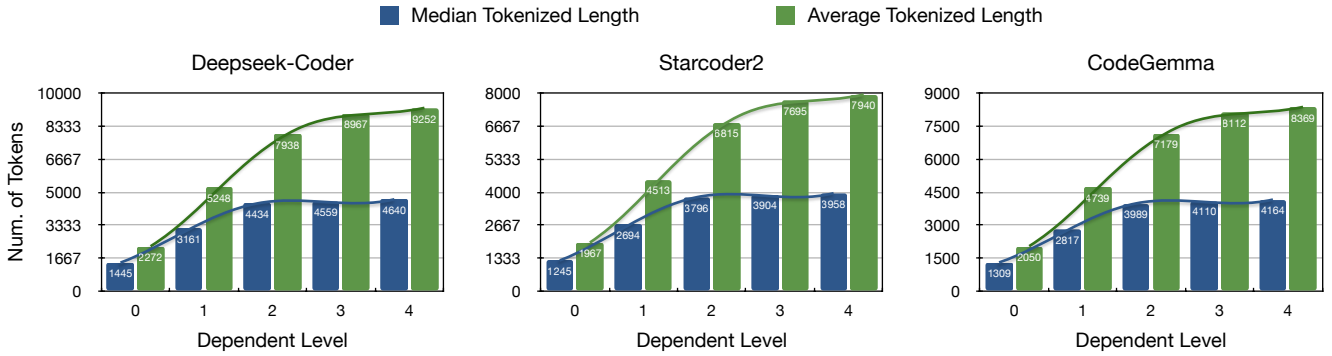

Figure 1: The distribution of tokenized prompt lengths in the CrossCodeEval benchmark. The x-axis represents the dependent level, and the y-axis represents the number of tokens. ■ denotes the median value of the tokenized prompt length. ■ denotes the average value of the tokenized prompt length.

Topological Dependency Analysis

Definition 1. (Dependency Level) Let F denote a set of files in a code repository, and let $f \in F$ represent a specific file. We define the dependency levels as follows:

$$\begin{aligned} I(f) &= \{g \mid g \text{ is imported by } f\} \\ D_0(f) &= \{f\} \\ D_{i+1}(f) &= D_i(f) \cup I(D_i(f)) \end{aligned} \tag{1}$$

We first identified the file requiring completion, then extracted all the import statements from the file with *Tree-Sitter*[1], and used a breadth-first search (BFS) method to progressively add dependent files.

Figure 1 illustrates the growth in the number of dependent files (calculated by the length of the tokenized prompt) as the number of dependency layers increases. We used median and average as statistical measures and found that in the vast majority of cases, the number of dependent files for a single file increases slowly after reaching four layers of dependencies. This suggests that using four layers of dependencies is sufficient to cover most scenarios. We further define:

$$D_\infty(f) = D_4(f) \cup \{F \setminus D_4(f)\} \tag{2}$$

to represent the prompt including all files in the repository.

In Table 2, the D-level rows show the results of completion using cross-file information with different dependency levels. The results indicate that although the maximum dependency depth of most files reaches 4 levels, only the information provided by $D_1(f)$ files is the most useful. Furthermore, the effectiveness of using $D_\infty(f)$ surpasses that of Random-All, indicating that besides $D_1(f)$ files, there are many other useful files within the repository.

Cross-File Content Analysis

Definition 2. (Pruning Level) We define the pruning levels into three categories:

- **P-Level 0**: No pruning is applied to the file content.
- **P-Level 1**: All global context content is removed from the file.
- **P-Level 2**: All global context content, function bodies and class method bodies are removed from the file.

Table 3 presents the results of completion using cross-file information with different pruning levels. We can see that the results of *P-level:1* outperform those of *P-level:0*, indicating that the Global Context information from cross-file content has minimal impact on the completion of the current

[1]https://tree-sitter.github.io/tree-sitter

XF-Context	Topological Dependency Analysis											
	DScoder-1.3B		DScoder-6.7B		Starcoder2-3B		Starcoder2-7B		CodeGemma-2B		CodeGemma-7B	
	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES
D-Level: 1	**15.44**	**55.03**	33.03	70.77	26.18	64.15	28.51	66.91	24.37	58.79	34.65	73.01
D-Level: 2	13.63	53.45	33.56	70.74	26.70	64.58	29.45	67.03	25.31	59.27	35.67	73.26
D-Level: 3	13.26	53.17	33.07	70.51	26.82	64.56	29.23	67.01	25.35	59.30	35.93	73.34
D-Level: 4	13.37	53.20	33.22	70.57	26.59	64.46	29.53	67.07	25.54	59.42	36.12	73.54
D-Level: ∞	5.76	46.22	**35.29**	**71.51**	**30.43**	**67.34**	**33.03**	**69.57**	**29.08**	**62.91**	**39.32**	**75.35**

Table 2: Comparison of completion results using different context dependency levels across six models. All the prompts is truncated to the max context window of the Code LLMs from the left. ∞ denotes the prompt including all files in the repository.

XF-Context	Cross-File Content Analysis											
	DScoder-1.3B		DScoder-6.7B		Starcoder2-3B		Starcoder2-7B		CodeGemma-2B		CodeGemma-7B	
	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES
P-Level: 0	6.18	46.19	33.94	70.98	28.32	66.87	31.45	69.09	26.93	62.13	36.69	74.42
P-Level: 1	6.55	46.58	36.20	71.90	**30.73**	**67.97**	**34.43**	**70.65**	**29.30**	**63.46**	**39.55**	**75.70**
P-Level: 2	**9.83**	**49.63**	34.73	70.89	30.02	66.41	31.26	68.24	27.34	61.13	38.31	74.32
+ *D-level:1*	9.45	49.44	**36.87**	**72.14**	29.91	66.96	32.62	69.11	28.93	62.03	39.17	75.16
+ *D-level:2*	8.70	48.61	36.38	71.66	29.64	66.99	32.96	69.13	28.44	61.76	39.06	74.91

Table 3: The results of completion using cross-file information with different pruning levels. + *D-level:x* denotes the model uses the cross-file information with dependency level x.

file. Additionally, the results of *P-level:2* are only slightly worse than those of $D_\infty(f)$, and when combined with the information from $D_1(f)$, they are almost equivalent to the results of $D_\infty(f)$. This suggests that the specific implementations of most cross-file functions have minimal impact on the completion of the current file, and retaining only the function header information is sufficient.

Hierarchical Context Pruning

Based on the analysis results concerning the dependencies and content of the files, we attempt to construct a hierarchical context prompt based on the importance and relevance of the repository content. This approach aims to enhance the accuracy of code completion models while effectively reducing the length of the context. Figure 2 shows the specific process for constructing a hierarchical context prompt.

Fine-grained Repository Modeling

In order to precisely control the content within the code repository, we employ *Tree-Sitter* to parse the files within the repository. We model the content using three types of nodes:

- **Function Node**: Represents a function or a class method within a code file.
- **Class Node**: Represents a class in a code file, consisting of the class's name, attributes, and Function Nodes.
- **File Node**: Represents a code file, comprising Nodes that represent the functions and classes within the file, along with global context information.

Hierarchical Context

As shown in the top right of Figure 2, following the settings in *Topological Dependency Analysis*, we conduct a dependency analysis on the files in the repository. We perform a topological sort based on the dependency relationships, centering around the file currently being completed. According to the experimental results in *Topological Dependency Analysis*, only files at dependency level 1 significantly enhance completion accuracy. Therefore, we select files designated as $D_1(f)$ to serve as dependency files. Ultimately, the files in the repository are categorized into three types: *current file*, *dependency files*, and *other files*. We will apply different strategies to optimize each type of file.

Current File. For the current file, any content within the file may be needed during completion, so we retain all content of the file and convert it into the Fill-in-the-middle (FIM) format.

Dependency Files. According to the experimental results in *Cross-File Content Analysis*, removing the global context across files does not affect the accuracy of completions. Therefore, for dependency files, we remove all global context from these files.

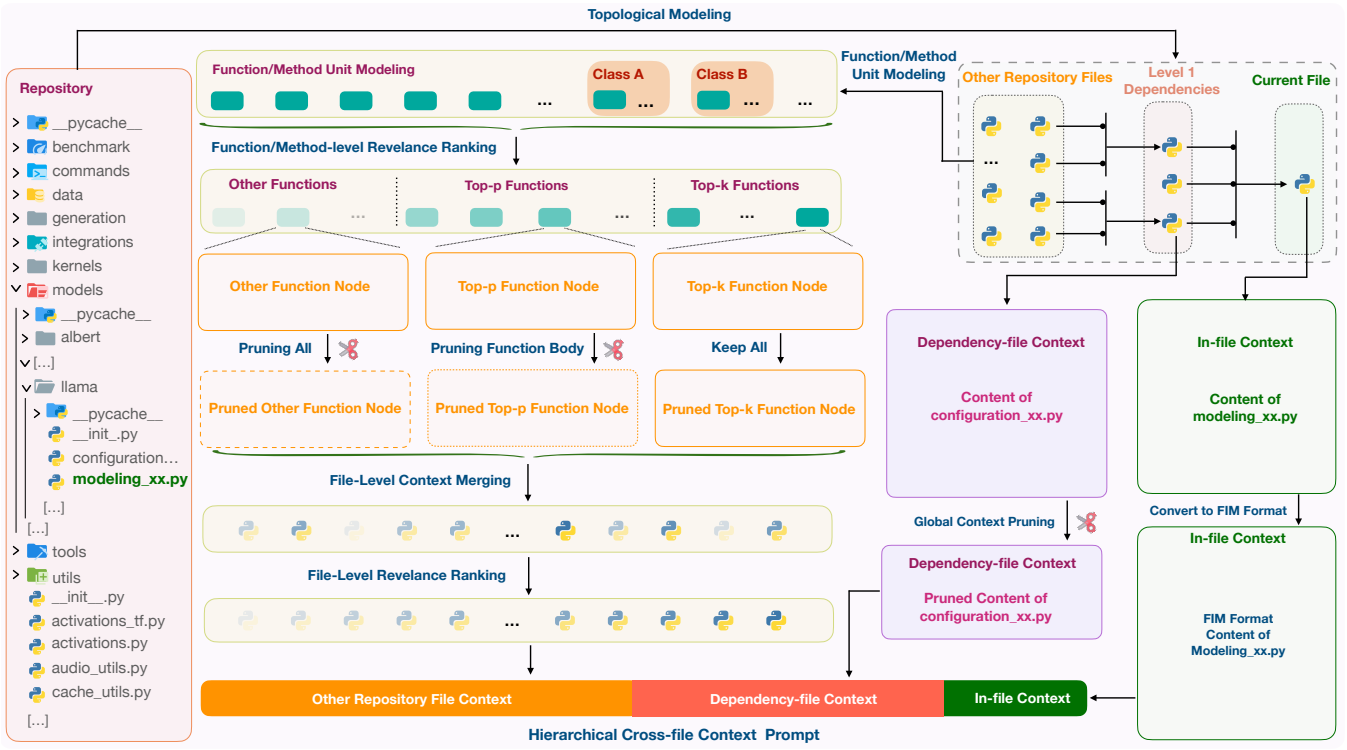

Figure 2: The framework of hierarchical context pruning for improving the performance of code large language models in real-world code completion tasks.

Other Files. We refer to files other than the current file and its direct dependency files, namely $\{F \setminus D_1(f)\} \setminus f\}$, collectively as other files. For the content in *other files*, we remove all global context, and then we employ **function-level** sampling and pruning methods to optimize the content of these files.

Function-level Sampling

In this study, we used OpenAI's text-embedding API[2] to embed each function (or class method) and query code snippet in the repository. We then used the pre-computed similarity of embeddings between the query and candidate functions (or class methods) as an indicator of relevance. We select the code from the current line of completion and the 10 lines before and after it as a query to find functions and class methods most relevant to the current completion content.

We implemented two sampling strategies (**top-k** and **top-p**) and designed distinct content pruning strategies for the functions (or class methods) sampled under each strategy, see *Function-level Pruning*.

Function-level Pruning

According to the experimental results in *Cross-File Content Analysis*, the global context from all non-current files and most of the function bodies (or class method bodies) within the code repository can be pruned. Appropriately pruning low-relevance content can significantly reduce the length of the prompt input to the model.

Let G denote the set of all functions and class methods in the repository, F_k represent the functions sampled using the top-k strategy, and F_p represent the functions sampled using the top-p strategy:

$$\begin{aligned} G_k &= \{g \mid g \in \text{Top}_k(G)\} \\ G_p &= \{g \mid g \in \text{Top}_p(G)\} \end{aligned} \tag{3}$$

where $G_k \subseteq G_p$. Content from functions and class methods not within the set $G_k \cup G_p$ was completely pruned.

Top-k Context Pruning. For functions (or class methods) within the set G_k, we retained their entire content.

Top-p Context Pruning. For functions (or class methods) in the set G_p but not in G_k, we prune their implementations and retained only their function headers (or class method headers).

File-level Relevance Ranking

Relevance Weighting Each function or class method in the repository is assigned a similarity score. We then apply different relevance weights to functions sampled using various sampling strategies, with highly relevant functions and class methods receiving higher weights. In this experiment,

[2]openai-text-embedding-ada-002

XF-Context	Hierarchical Context Pruning (Top-p: 1.0)											
	DScoder-1.3B		DScoder-6.7B		Starcoder2-3B		Starcoder2-7B		CodeGemma-2B		CodeGemma-7B	
	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES
Previous SOTA	**22.59**	**68.85**	37.06	73.09	29.83	66.87	31.71	69.09	26.93	62.13	40.79	74.42
Top-k: 0	9.45	49.44	36.87	72.14	29.91	66.96	32.62	69.11	28.93	62.03	39.17	75.16
Top-k: 5	9.64	49.78	39.74	73.90	32.68	69.05	35.76	71.41	31.26	63.74	42.44	76.95
Top-k: 10	9.91	49.85	**40.30**	**74.56**	**34.15**	**69.37**	**36.47**	**71.50**	**31.82**	**64.34**	**42.63**	**77.35**

Table 4: The results of completion using hierarchical context pruning with different top-k values.

XF-Context	Hierarchical Context Pruning (Top-k: 5)											
	DScoder-1.3B		DScoder-6.7B		Starcoder2-3B		Starcoder2-7B		CodeGemma-2B		CodeGemma-7B	
	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES
Previous SOTA	**22.59**	**68.85**	37.06	73.09	29.83	66.87	31.71	69.09	26.93	62.13	40.79	74.42
Top-p: 0.1	14.27	53.94	37.85	73.11	32.99	**68.75**	34.16	70.43	29.19	62.09	40.98	**76.26**
Top-p: 0.2	13.52	53.20	38.04	73.13	**33.15**	68.59	34.84	70.40	29.72	62.32	40.94	76.25
Top-p: 0.3	12.88	52.60	**38.49**	**73.19**	32.84	68.31	**35.22**	**70.64**	**30.13**	**62.77**	**41.21**	76.20

Table 5: The results of completion using hierarchical context pruning with different top-p values.

our specific setup is as follows:

$$W(g) = \begin{cases} 1.0, & \forall g \in G_k \\ 0.5, & \forall g \in G_p \setminus G_k \\ 0.0, & \forall g \in G \setminus (G_k \cup G_p) \end{cases} \tag{4}$$

where G_k and G_p represent the functions with the highest relevance scores sampled using the top-k and top-p strategies, respectively.

Class-level Relevance The similarity of a class is defined as the weighted sum of its class methods:

$$S(C) = \sum_{g \in C} W(g) * S(g) \tag{5}$$

where, C represents the class, g represents the class method, and $S(g)$ represents the similarity score of the class method.

File-level Relevance The similarity of a file is defined as the weighted sum of its functions and classes:

$$S(f) = \sum_{g \in \mathcal{G}} W(g) * S(g) + \sum_{c \in \mathcal{C}} S(c) \tag{6}$$

where, $\mathcal{G}$ and $\mathcal{C}$ represent the set of functions and classes in the file, respectively.

Finally, we sort the files at the file-level according to the relevance score to determine their relative positions in the prompt.

Experimental Results

Top-k and Top-p Analysis We initially fixed top-p at 1.0 and tested the impact of different top-k values on completion accuracy. Table 4 presents some of the experimental results, and we provides a more comprehensive results in the appendix of the extended version. We observed that increasing the top-k value beyond 5 did not result in significant improvements in accuracy. Therefore, we conclude that a top-k value of 5 is sufficient.

We further fixed the top-k value at 5 and tested the impact of varying top-p values (ranging from 0.1 to 0.9) on completion accuracy. Partial experimental results are presented in Table 5, with more comprehensive results available in the appendix of the extended version. Our observations indicate that increasing the top-p value enhances completion accuracy; however, beyond a top-p value of 0.3, the improvement in accuracy slows considerably. Thus, we consider 0.3 to be a reasonable value.

Comparison with Random-All Figure 3 visually compares the Hierarchical Context Pruning (HCP) strategy (top-k=5, top-p=0.3) with the method of randomly concatenating all repository code files across three dimensions: completion accuracy, throughput rate, and input length. The visualization shows that, compared to random concatenation, **HCP** significantly reduces input length (enhancing throughput) while improving the model's completion accuracy.

Related Work

Code Large Language Models

General Code LLMs Building on the success of large language models, researchers have explored further pre-training these models on code data to enhance their performance on code-related tasks (Chen et al. 2021; Austin et al. 2021; Cassano et al. 2022). This has led to the development and release of several powerful code-focused large language models, such as the CodeX series (Chen et al.

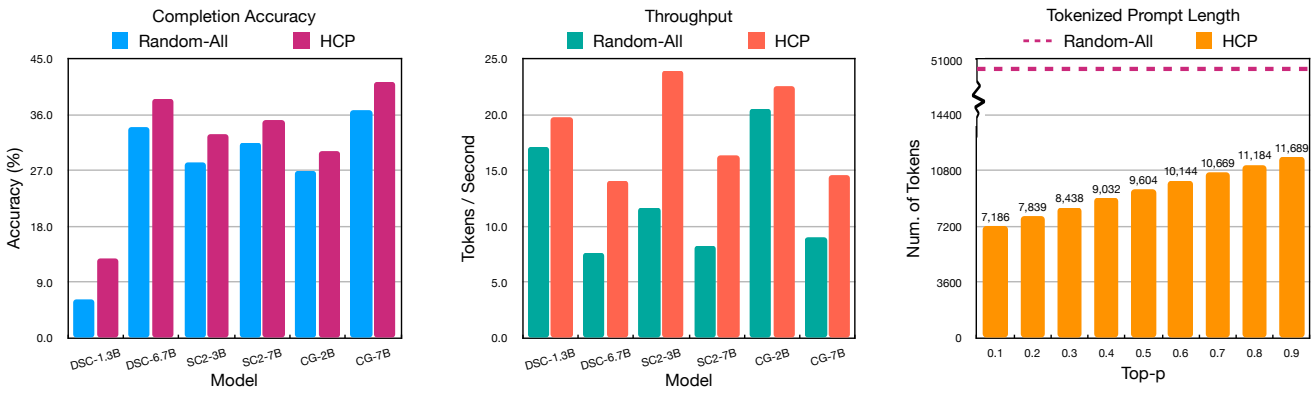

Figure 3: **left**: Comparison of completion results using random-all and the hierarchical context pruning across six models. **middle**: Comparison of throughput using random-all and the hierarchical context pruning across six models. **right**: Comparison of prompt length using random-all and the hierarchical context pruning of different top-p values (top-k=5).

2021), CodeGen series (Nijkamp et al. 2023c), CodeT5+ (Wang et al. 2023), and Lemur (Xu et al. 2023). These models have demonstrated strong capabilities in handling code-related tasks.

Infilling Code LLMs Infilling scenarios make up the majority of code completion tasks in the real world. Bavarian et al. (2022) demonstrated that pre-training Code LLMs with a certain proportion of fill-in-the-middle format code data enables these models to fill in missing code based on the surrounding context, without compromising their original left-to-right generation capabilities. Building on these findings, many subsequent Code LLMs, such as Incoder (Fried et al. 2023), SantaCoder (Allal et al. 2023), CodeGen2 (Nijkamp et al. 2023a), CodeLLama (Rozière et al. 2024), Starcoder (Li et al. 2023; Lozhkov et al. 2024), DeepSeek-Coder (Guo et al. 2024) and StableCoder (Pinnaparaju et al. 2024), have been developed with enhanced infilling capabilities.

Repository-Level Code Benchmarks

To better assess the repository-level code completion capabilities of Code LLMs, researchers have developed several comprehensive benchmarks. CrossCodeEval (Ding et al. 2023) introduced a multilingual cross-file code completion benchmark that underscores the importance of understanding cross-file context in real-world software development environments. RepoBench (Liu, Xu, and McAuley 2023) is a new benchmark specifically designed for evaluating code auto-completion systems at the repository level. It includes three interconnected evaluation tasks, code snippet retrieval, code completion, and end-to-end processing, emphasizing the critical role of multi-file context. Recently, additional repository-level code completion benchmarks have emerged, such as CoderEval (Zhang et al. 2024) and EvoCodeBench (Li et al. 2024).

Repo-level Code Completion

Due to the lack of repository structure awareness in earlier Code LLMs, most research efforts (Shrivastava, Larochelle, and Tarlow 2023), such as ReAcc (Lu et al. 2022), RepoCoder (Zhang et al. 2023), RepoHyper (Phan et al. 2024) and RepoFuse (Liang et al. 2024b), adopted a retrieval-augmented generation approach. This approach involves retrieving relevant code snippets from the repository and concatenating them as comments at the beginning of the current file to leverage cross-file code information. Recent work has further considered additional factors to improve the repository-level completion accuracy of Code LLMs. For example, CoCoGen (Bi et al. 2024) incorporated compiler feedback, while DraCo (Cheng, Wu, and Hu 2024) took data flow factors into account. However, most of these efforts have not utilized the Fill-in-the-Middle capability of Code LLMs. Additionally, since Repo-Code LLMs are relatively new, these works did not take advantage of the new capabilities offered by Repo-Code LLMs in their design. To the best of our knowledge, our research is pioneering in exploring how to develop high-quality completion prompts by leveraging the repository-awareness of Repo-Code LLMs.

Conclusion

In this study, we conducted extensive experiments on six newly released Code LLMs that were trained on repository-level code data, and proposed an effective method for constructing high-quality repository-level code completion prompts. Our preliminary experimental results showed that maintaining the topological dependency order of code files in the prompt can improve completion accuracy, while removing global cross-file context and specific function implementations does not significantly reduce accuracy. Based on these findings, we proposed the **Hierarchical Context Pruning (HCP)** method to construct efficient repository-level code completion prompts. Compared to previous state-of-the-art methods, our HCP approach achieved better results on five out of six Code LLMs and effectively controlled input length, demonstrating the effectiveness of our method.

Acknowledgments

This work was supported by Open Research Fund of Key Laboratory of Intelligent Education Technology and Application of Zhejiang Province, Zhejiang Normal University (Grant No. jykf21002w), National Natural Science Foundation of China (62376262), MoE Key Laboratory of Brain-inspired Intelligent Perception and Cognition, University of Science and Technology of China (Grant No. 2421002).

References

Allal, L. B.; Li, R.; Kocetkov, D.; Mou, C.; Akiki, C.; Ferrandis, C. M.; Muennighoff, N.; Mishra, M.; and et al., A. G. 2023. SantaCoder: don't reach for the stars! arXiv:2301.03988.

Austin, J.; Odena, A.; Nye, M.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.; Terry, M.; Le, Q.; and Sutton, C. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732.

Bavarian, M.; Jun, H.; Tezak, N.; Schulman, J.; McLeavey, C.; Tworek, J.; and Chen, M. 2022. Efficient Training of Language Models to Fill in the Middle. arXiv:2207.14255.

Bi, Z.; Wan, Y.; Wang, Z.; Zhang, H.; Guan, B.; Lu, F.; Zhang, Z.; Sui, Y.; Shi, X.; and Jin, H. 2024. Iterative Refinement of Project-Level Code Context for Precise Code Generation with Compiler Feedback. arXiv:2403.16792.

Cassano, F.; Gouwar, J.; Nguyen, D.; Nguyen, S.; Phipps-Costin, L.; Pinckney, D.; Yee, M.-H.; Zi, Y.; Anderson, C. J.; Feldman, M. Q.; Guha, A.; Greenberg, M.; and Jangda, A. 2022. MultiPL-E: A Scalable and Extensible Approach to Benchmarking Neural Code Generation. arXiv:2208.08227.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H. P.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; and et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374.

Cheng, W.; Wu, Y.; and Hu, W. 2024. Dataflow-Guided Retrieval Augmentation for Repository-Level Code Completion. arXiv:2405.19782.

Ding, Y.; Wang, Z.; Ahmad, W. U.; Ding, H.; Tan, M.; Jain, N.; Ramanathan, M. K.; Nallapati, R.; Bhatia, P.; Roth, D.; and Xiang, B. 2023. CrossCodeEval: A Diverse and Multilingual Benchmark for Cross-File Code Completion. arXiv:2310.11248.

Fried, D.; Aghajanyan, A.; Lin, J.; Wang, S.; Wallace, E.; Shi, F.; Zhong, R.; tau Yih, W.; Zettlemoyer, L.; and Lewis, M. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. arXiv:2204.05999.

Guo, D.; Zhu, Q.; Yang, D.; Xie, Z.; Dong, K.; Zhang, W.; Chen, G.; Bi, X.; Wu, Y.; Li, Y. K.; Luo, F.; Xiong, Y.; and Liang, W. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. arXiv:2401.14196.

Li, J.; Li, G.; Zhang, X.; Dong, Y.; and Jin, Z. 2024. EvoCodeBench: An Evolving Code Generation Benchmark Aligned with Real-World Code Repositories. arXiv:2404.00599.

Li, R.; Allal, L. B.; Zi, Y.; Muennighoff, N.; Kocetkov, D.; Mou, C.; Marone, M.; Akiki, C.; Li, J.; Chim, J.; and et al. 2023. StarCoder: may the source be with you! arXiv:2305.06161.

Liang, M.; Xie, X.; Zhang, G.; Zheng, X.; Di, P.; wei jiang; Chen, H.; Wang, C.; and Fan, G. 2024a. REPOFUSE: Repository-Level Code Completion with Fused Dual Context. arXiv:2402.14323.

Liang, M.; Xie, X.; Zhang, G.; Zheng, X.; Di, P.; wei jiang; Chen, H.; Wang, C.; and Fan, G. 2024b. REPOFUSE: Repository-Level Code Completion with Fused Dual Context. arXiv:2402.14323.

Liu, T.; Xu, C.; and McAuley, J. 2023. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems. arXiv:2306.03091.

Lozhkov, A.; Li, R.; Allal, L. B.; Cassano, F.; Lamy-Poirier, J.; Tazi, N.; Tang, A.; Pykhtar, D.; Liu, J.; Wei, Y.; Liu, T.; Tian, M.; Kocetkov, D.; and et al., A. Z. 2024. StarCoder 2 and The Stack v2: The Next Generation. arXiv:2402.19173.

Lu, S.; Duan, N.; Han, H.; Guo, D.; won Hwang, S.; and Svyatkovskiy, A. 2022. ReACC: A Retrieval-Augmented Code Completion Framework. arXiv:2203.07722.

Nijkamp, E.; Hayashi, H.; Xiong, C.; Savarese, S.; and Zhou, Y. 2023a. CodeGen2: Lessons for Training LLMs on Programming and Natural Languages. arXiv:2305.02309.

Nijkamp, E.; Pang, B.; Hayashi, H.; Tu, L.; Wang, H.; Zhou, Y.; Savarese, S.; and Xiong, C. 2023b. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. arXiv:2203.13474.

Nijkamp, E.; Pang, B.; Hayashi, H.; Tu, L.; Wang, H.; Zhou, Y.; Savarese, S.; and Xiong, C. 2023c. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. arXiv:2203.13474.

Phan, H. N.; Phan, H. N.; Nguyen, T. N.; and Bui, N. D. Q. 2024. RepoHyper: Better Context Retrieval Is All You Need for Repository-Level Code Completion. arXiv:2403.06095.

Pinnaparaju, N.; Adithyan, R.; Phung, D.; Tow, J.; Baicoianu, J.; Datta, A.; Zhuravinskyi, M.; Mahan, D.; Bellagente, M.; Riquelme, C.; and Cooper, N. 2024. Stable Code Technical Report. arXiv:2404.01226.

Rozière, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X. E.; Adi, Y.; Liu, J.; Sauvestre, R.; Remez, T.; Rapin, J.; and et al., A. K. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950.

Shrivastava, D.; Larochelle, H.; and Tarlow, D. 2023. Repository-Level Prompt Generation for Large Language Models of Code. arXiv:2206.12839.

Sun, Q.; Chen, Z.; Xu, F.; Cheng, K.; Ma, C.; Yin, Z.; Wang, J.; Han, C.; Zhu, R.; Yuan, S.; Guo, Q.; Qiu, X.; Yin, P.; Li, X.; Yuan, F.; Kong, L.; Li, X.; and Wu, Z. 2024. A Survey of Neural Code Intelligence: Paradigms, Advances and Beyond. arXiv:2403.14734.

Team, C.; Hartman, A. J.; Hu, A.; Choquette-Choo, C. A.; Zhao, H.; Fine, J.; Hui, J.; Shen, J.; Kelley, J.; Howland, J.; Bansal, K.; Vilnis, L.; Wirth, M.; Nguyen, N.; Michel, P.; Choy, P.; Joshi, P.; Kumar, R.; Hashmi, S.; Agrawal, S.; Zuo,

S.; Warkentin, T.; and Gong, Z. 2024. CodeGemma: Open Code Models Based on Gemma.

Wang, Y.; Le, H.; Gotmare, A. D.; Bui, N. D. Q.; Li, J.; and Hoi, S. C. H. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. arXiv:2305.07922.

Xu, Y.; Su, H.; Xing, C.; Mi, B.; Liu, Q.; Shi, W.; Hui, B.; Zhou, F.; Liu, Y.; Xie, T.; Cheng, Z.; Zhao, S.; Kong, L.; Wang, B.; Xiong, C.; and Yu, T. 2023. Lemur: Harmonizing Natural Language and Code for Language Agents. arXiv:2310.06830.

Zhang, F.; Chen, B.; Zhang, Y.; Keung, J.; Liu, J.; Zan, D.; Mao, Y.; Lou, J.-G.; and Chen, W. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. arXiv:2303.12570.

Zhang, Y.; Zhang, W.; Ran, D.; Zhu, Q.; Dou, C.; Hao, D.; Xie, T.; and Zhang, L. 2024. Learning-based Widget Matching for Migrating GUI Test Cases. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, ICSE '24. ACM.