ON PRETRAINING FOR PROJECT-LEVEL CODE COMPLETION

Maksim Sapronov, Evgeniy Glukhov
JetBrains Research
{name.last_name}@jetbrains.com

ABSTRACT

Repository-level pretraining is commonly used to enable large language models for code to leverage codebase-wide context. This enhances their ability to generate accurate and context-aware code completions. In this work, we investigate how different repository-processing strategies affect in-context learning in OpenCoder, a 1.5B-parameter model. We extend its context window from 4,096 to 16,384 tokens by training on additional 1B tokens of curated repository-level data. Despite relying on a smaller dataset than competing models (which often use hundreds of billions of tokens), our model achieves comparable performance on the Long Code Arena benchmark. We find that various repository-processing techniques yield similarly strong results, with the primary gain coming from adapting to a new rotary positional embedding (RoPE) scaling parameter. Finally, we show that a simpler file-level training approach at the original sequence length remains highly effective, opening up repository-level code completion research to settings with more constrained data and compute resources.

1 INTRODUCTION AND MOTIVATION

Large Language Models (LLMs) trained on source code, commonly known as Code LLMs, have demonstrated impressive capabilities on a variety of code-related tasks (Lu et al., 2021; Jimenez et al., 2023; Hou et al., 2024; Jiang et al., 2024). Traditionally, these models are pretrained on individual files, effectively capturing local context but often missing broader, project-level information. To address this limitation, several recent works have incorporated a *repository-level pretraining* phase, *i.e.,* a stage of pretraining during which the model gets training examples from entire repositories to learn context spanning multiple files, shared dependencies, and cohesive development patterns. For example, models such as DeepSeek Coder, Starcoder 2, Qwen2.5 Coder and CodeGemma (Guo et al., 2024; Lozhkov et al., 2024; Hui et al., 2024; CodeGemma Team et al., 2024) incorporate repository-level pretraining to extend their context windows and capture cross-file relationships. Beyond repository-level pretraining, other techniques have been investigated (Guo et al., 2023; Zhang et al., 2023; 2024).

While repository-level pretraining enhances long-context capabilities, it also introduces significant challenges. Firstly, it requires huge amounts of data, *e.g.,* Qwen2.5 Coder's repository-level pretraining leverages approximately 300B tokens of repository data. Secondly, long sequences can strain computational resources due to the quadratic complexity of traditional transformer architecture. Recent advances in efficient attention mechanisms (for example, Flash Attention and Ring Attention (Dao, 2023; Liu et al., 2023a)) have enabled training with context lengths typically in the tens of thousands of tokens, and even millions of tokens for smaller models. However, effective utilization of repository-level information remains challenging both for training and inference (Ding et al., 2022; Liu et al., 2023b; Ding et al., 2024; Pei et al., 2023).

In this work, we focus on a single line repository-level code completion and study context extension pretraining for various repository preprocessing approaches. Following the Long Code Arena benchmark (Bogomolov et al., 2024) terminology, we evaluate the impact of different *context composers*, *i.e.,* processors that transform repository files into context strings. Our approach builds on the OpenCoder base model (Huang et al., 2024), originally configured with a 4,096 (4K) context window, by training on repository-level input sequences of up to 16,384 (16K) tokens. This extension results in significantly improved performance on 16K token sequences compared to the initial configuration.

We assess our methods using the Project-Level Code Completion task from the Long Code Arena benchmark, which effectively estimates a model's ability to handle cross-file dependencies in realistic settings. By isolating the impact of repository-level pretraining and comparing different context composer strategies, our study provides practical insights for enhancing long-context code completion performance.

The main contributions of this paper are:

1. We boost the project-level code completion performance of OpenCoder 1.5B to state-of-the-art levels using only 1B tokens of training data;
2. Our experiments reveal that the choice of context composer during pretraining has only a marginal impact on final model quality, with performance scores ranging from 45.2 to 48.8 (out of 100) on the chosen metric.

2 Experiment Design

We start this section with a description of the data sourcing and preparation steps, then explain our training setup, and finally we detail our evaluation strategy. In addition, we discuss the role of context composers — repository processing functions, and their distinct modes for the training and evaluation phases.

2.1 Training Data

To collect the training data, we follow the approach from the Long Code Arena benchmark, see Bogomolov et al. (2024) for more details. Starting with open-source GitHub repositories in Python and then traverse the Git commit history for each repository to extract repository data. Filtering process is described in B.1

The repository data for each commit consists of two elements: (1) *repository snapshot* — a context source with contents of all code and text files before the commit; (2) *completion files* — list of files to perform completion on with contents of all `.py` files added in that commit.

The resulting *raw repositories dataset* contains 1,640 repositories, 160,801 commits, and 361,052 completion files. The total number of characters in completion files is 1.7B, and in repository snapshot files — 4.8T.

To get a *context string* from the repository data, we apply a *context composer* to a repository snapshot. A *context composer* is a repository processor that (1) sorts a subset of files (or file chunks) from the repository snapshot by relevance (based on specified criteria), (2) retrieves the most relevant ones that fit within the context window, and (3) concatenates them into a single string with the most relevant file at the end.

For each context composer in the list provided in Appendix A.1, we prepare the *composed dataset* from the raw repositories dataset which consists of two columns: (1) *completion file* — one file from the completion files; (2) *composed context* — string with the result of the context composer.

Of the various context composers used in our experiments, we highlight the following two for clarity and conciseness.

1. **File-level** — Produces an empty context.
2. **Path Distance `.py`** — The context is built solely from .py files, sorted in descending order by their path distance from the completion file. For files with the same distance, a secondary sort uses the Intersection over Union (IoU) score of their matching lines.

Note that rows from the raw repositories dataset can produce multiple rows of the composed dataset with one row for each completion file.

2.2 Training

For each context composer, we pretrain OpenCoder 1.5B model (Huang et al., 2024) on the corresponding composed dataset with a context window size of 16,384 tokens.

In our *training mode* for the context composer, we aim to include as many files as possible in the context string. To achieve this, we truncate both the context string and the completion file, ensuring that the context-to-completion token ratio is at least 3 : 1. For more details, see Appendix C.2.

To extend the model context window size, we change RoPE's base frequency θ from 10,000 to 500,000 following the ABF approach (Xiong et al., 2023); our focus is on this method, although alternative approaches exist (Chen et al., 2023; Peng et al., 2023; Zhong et al., 2024; Liu et al., 2023c).

To evaluate models after training on approximately 1 billion tokens, and in accordance with our training hyperparameters (see Appendix C.1), we save the model's weights at the 512th optimization step, referring to this saved state as a *checkpoint*.

2.3 Evaluation

Our evaluation setup is based on the *large context* dataset, which is a part of the Project-level code completion task from the Long Code Arena dataset (LCA-large) (Bogomolov et al., 2024). The task is to write the next line of code based on the file prefix and the repository snapshot, with the evaluation metric being Exact Match (percentage of correct answers). Additionally, each line has one of six categories that corresponds to various scenarios of project cross-file dependencies. We use categories *infile* and *inproject*, *i.e.,* a completion line that contains an API declared in the completion file or in repository snapshot files. These two categories indicate in-context learning capabilities the best out of six, since they contain more project-specific information.

We evaluate each checkpoint on *infile* and *inproject* categories for two different context composers in the *evaluation mode*: (1) **FL-4K**: File-Level composer with maximum sequence length 4K tokens, and (2) **PD-16K**: Path Distance `.py` composer with maximum sequence length 16K tokens. Moreover, we calculate **RCB**: repository-context boost, *i.e.,* the difference between scores for the PD-16K and FL-4K composers.

3 Results

In the following subsections, we present our main results: first, we achieve state-of-the-art quality on LCA-large with much less extensive repository-level pretraining; second, we demonstrate the impact of the context composer choice on the result of repository-level pretraining. Additionally, we provide a more detailed comparative study of repository-level pretraining in the Appendix.

3.1 Benchmarking Against State-of-the-Art

To estimate the effectiveness of our trained models, we compare them to DeepSeek Coder 1.3B, OpenCoder 1.5B with no repository-level pretraining, and Qwen2.5-Coder 0.5B and 1.5B. These models serve as strong baselines, representing state-of-the-art performance in similar parameter ranges. Results are shown in Table 1.

We started with OpenCoder model which is pretty good on file-level code completion among the similar size models and got a significant gain by file-level pretraining on just 1B tokens [1]. This approach serves as a guideline for scenarios with limited data and low GPU resources, since we do not need repositories, and do not actually need long context for training. We can even achieve Qwen2.5-Coder performance level with 1B tokens of curated repository-level data.

3.2 Impact of Context Composer Choice

Findings in the previous subsection leave an open question if there is even better composer for repository-level pretraining. To answer this question, we evaluate all studied composers and present

[1] While the model had access to 1B tokens, it was actually used just 72M tokens for training.

Table 1: Comparison of existing models on LCA-large for the line categories: *inproject* and *infile*. FL-4K and PD-16K report Exact Match scores for File-level and Path Distance `.py` evaluation composers. RCB represents the repository-context boost score.

	inproject			**infile**		
Model	FL-4K	PD-16K	RCB	FL-4K	PD-16K	RCB
Qwen2.5-Coder 0.5B	22.6	44.2	+21.6	27.5	43.2	+15.7
DeepSeek Coder 1.3B	25.1	42.3	+17.2	30.3	43.8	+13.5
OpenCoder 1.5B	<u>26.4</u>	0.0	–26.4	32.6	0.0	–32.6
Qwen2.5-Coder 1.5B	**27.2**	<u>48.5</u>	<u>+21.3</u>	**34.3**	**49.7**	**+15.4**
Ours (OpenCoder 1.5B)						
File-level pretr.	25.9	45.2	+19.3	33.0	44.6	+11.6
Path Distance `.py` pretr.	26.2	**48.8**	**+22.6**	<u>33.1</u>	<u>47.6</u>	<u>+14.5</u>

Table 2: Results of evaluating checkpoints after repository-level pretraining. Evaluation dataset is LCA-large for the line categories: *inproject* and *infile*. FL-4K and PD-16K report Exact Match scores for File-level and Path Distance `.py` evaluation composers.

Pretraining Composer	**inproject**		**infile**	
	FL-4K	PD-16K	FL-4K	PD-16K
Base model (no training)	26.4	0.0	32.6	0.0
File-level	25.9	45.2	33.0	44.6
Path Distance `.py`	26.2	48.8	33.1	47.6
Other Pretraining Composers	25.5 – 26.5	46.8 – 48.7	32.3 – 33.3	45.6 – 47.8

condensed results for in Table 2 and extended results in Table 3. Our experiments demonstrate the performance variations across repository level pretrainings with different context composers.

We observe that file-level composer pretraining results in +19.3 repository-context boost, with other pretraining strategies getting repository-context boost within a +20.3 to +22.9 range. Combining with comparable values of the Exact Match, we validate that adapting to the longer context window, *i.e.,* new RoPE's base frequency, rather than the specific sequence composition, is the primary factor in repository-level pretraining, with context composers contributing only marginally for suggested approaches.

4 CONCLUSION

In this paper, we address the challenge of project-level code completion by evaluating pretraining for code LLM on various data extracted from repository. Our extensive experiments demonstrate that even relatively small training dataset and simple context composer (*e.g.,* file-level or path distance) is enough to get a model comparable to the latest state-of-the-art code LLMs. This insight reduces the complexity of repository-level pretraining, which effectively minimizes the technical complexities and encourages to broadly research the topic.

Although our findings are promising, they have certain limitations. Our experiments are limited to the OpenCoder model, and it remains unclear whether they generalize to other LLMs. A key direction for future work is to apply our approach on a broader range of Code LLMs. However, recent Code LLMs were released after the repository-level pretraining stage, which may introduce inconsistencies in evaluation.

ACKNOWLEDGMENTS

We appreciate the contributions of Alexander Bezzubov, Egor Bogomolov, Timofey Bryksin and Yaroslav Golubev from JetBrains, whose guidance greatly enhanced this research.

REFERENCES

Egor Bogomolov, Aleksandra Eliseeva, Timur Galimzyanov, Evgeniy Glukhov, Anton Shapkin, Maria Tigina, Yaroslav Golubev, Alexander Kovrigin, Arie van Deursen, Maliheh Izadi, and Timofey Bryksin. Long code arena: a set of benchmarks for long-context code models. *arXiv preprint arXiv:2406.11612*, 2024.

Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*, 2023.

CodeGemma Team, Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A Choquette-Choo, Jingyue Shen, Joe Kelley, et al. CodeGemma: open code models based on gemma. *arXiv preprint arXiv:2406.11409*, 2024.

Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. CoCoMIC: Code completion by jointly modeling in-file and cross-file context. *arXiv preprint arXiv:2212.10007*, 2022.

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

Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian McAuley. LongCoder: A long-range pre-trained language model for code completion. *arXiv preprint arXiv:2306.14893*, 2023.

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

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79, 2024.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J Yang, JH Liu, Chenchen Zhang, Linzheng Chai, et al. OpenCoder: The open cookbook for top-tier code large language models. *arXiv preprint arXiv:2411.04905*, 2024.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. Qwen2.5-Coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? *arXiv preprint arXiv:2310.06770*, 2023.

Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. *arXiv preprint arXiv:2310.01889*, 2023a.

Tianyang Liu, Canwen Xu, and Julian McAuley. RepoBench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*, 2023b.

Xiaoran Liu, Hang Yan, Shuo Zhang, Chenxin An, Xipeng Qiu, and Dahua Lin. Scaling laws of rope-based extrapolation. *arXiv preprint arXiv:2310.05209*, 2023c.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. StarCoder 2 and The Stack v2: The Next Generation. *arXiv preprint arXiv:2402.19173*, 2024.

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

Hengzhi Pei, Jinman Zhao, Leonard Lausen, Sheng Zha, and George Karypis. Better context makes better code language models: A case study on function call argument completion. *arXiv preprint arXiv:2306.00381*, 2023.

Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. YaRN: efficient context window extension of large language models. *arXiv preprint arXiv:2309.00071*, 2023.

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

Wenhan Xiong, Jingyu Liu, Igor Molybog, Hejia Zhang, Prajjwal Bhargava, Rui Hou, Louis Martin, Rashi Rungta, Karthik Abinav Sankararaman, Barlas Oguz, et al. Effective long-context scaling of foundation models. *arXiv preprint arXiv:2309.16039*, 2023.

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

Kechi Zhang, Ge Li, Huangzhao Zhang, and Zhi Jin. HiRoPE: Length extrapolation for code models using hierarchical position. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13615–13627, 2024.

Meizhi Zhong, Chen Zhang, Yikun Lei, Xikai Liu, Yan Gao, Yao Hu, Kehai Chen, and Min Zhang. Understanding the RoPE extensions of long-context LLMs: An attention perspective. *arXiv preprint arXiv:2406.13282*, 2024.

A Context Composers

A.1 Complete List

All composers follow two standard preprocessing steps, filtering out empty files and normalizing all line separators to Line Feed (LF). With these shared characteristics, the full list of context composers ensures comprehensive coverage for research exploration.

1. **File-level** — Produces an empty context.
2. **Path Distance `.py`** — Constructs the context using only files with the `.py` extension. The selected files are sorted in descending order based on their path distance from the completion file. If multiple files share the same path distance, a secondary sorting step is applied using the Intersection over Union (IoU) metric, computed over lines shared with the completion file. The IoU metric is calculated on lines with leading and trailing whitespace characters removed, considering only those lines that are at least five characters long after the whitespace removal.
3. **Lines IoU `.py`** — Similar to the Path Distance `.py` method but does not apply the primary sorting step based on path distance. Instead, files are directly ranked using the IoU metric.
4. **Code Chunks** — Removes all docstrings, comments, and import statements from the context produced by Path Distance `.py`.
5. **Half-memory `.py`** — Starts with the context produced by Path Distance `.py`. Each line is independently removed with a dropout probability of 0.5, maintaining the overall saturation of the context window.
6. **Declarations `.py`** — Builds upon Path Distance `.py` by filtering out all non-declarative elements, retaining only function and class declarations.
7. **Text Chunks `.py`** — Uses Path Distance `.py` as the base method. All code is removed from the context, leaving docstrings and comments only.
8. **Text files** — Constructs the context using files with the extensions `.json`, `.yaml`, `.yml`, `.sh`, `.md`, `.txt`, and `.rst`. The selected files are grouped in ascending order of relevance: [`.json`], [`.yaml`, `.yml`], [`.sh`], [`.md`, `.txt`, `.rst`]. Within each group, a secondary sorting step is performed in descending order based on path distance from the completion file.
9. **Random files** — Constructs the context by randomly ordering all files from the repository snapshot.
10. **Random `.py`** — Selects only files with the `.py` extension and orders them randomly.
11. **Mixed context**[2] — The context for each data point is constructed by randomly selecting one of the following composers: File-level, Path Distance `.py`, Half-memory `.py`, Declarations `.py`, Text files, Random files, or Duplication.

Furthermore, we propose four additional context composers, which are omitted from the results discussion as they do not reflect realistic scenarios.

1. **Random tokens** — Constructs the context using a randomly sampled sequence of non-special tokens, each selected independently and with equal probability.
2. **Duplication** — Constructs the context by concatenating the content of the completion file repeatedly until the maximum context window size is reached.
3. **Leak** — Starts with the context produced by Path Distance `.py`. The completion file is randomly split into five segments at newline characters, which then disjointedly replace context lines at random positions, approximately preserving the original token count.
4. **Masked Leak** — Starts with the context produced by Path Distance `.py`. The completion file is divided into segments, each consisting of five lines with one overlapping line at the beginning and one at the end. These segments independently and disjointedly replace context lines at random positions. Additionally, each token in the context has a 0.15 probability of being replaced with a different non-special token.

[2]Duplication composer is disabled in evaluation mode

For each composer we also consider two modifications:

- *reversed* — we retrieve files that fit into the context window with the composer and reverse their order, so the most relevant one is in the beginning of the context string;
- *irrelevant* — we reverse the order of files obtained from the composer and therefore retrieve most irrelevant files.

A.2 Input Formatting

All composers employ a uniform strategy for input formatting. The files processed by a composer undergo a predefined formatting pattern 1, which uses a special token from the OpenCoder's vocabulary. Subsequently, the processed files are concatenated into a single string, referred to as the composed context.

```
<file_sep># {file_name}\n{file_content}
```

Figure 1: File Representation

A similar transformation is applied independently to the completion file.

B Training Dataset

B.1 Filtering

To avoid training on test data, we exclude repositories used in the Long Code Arena's Project-level code completion task. In addition, to ensure data relevance and quality, we apply the following filtering criteria. First, all commits made prior to 2010 are excluded. Second, completion files with lengths outside the closed interval $[800, 25000]$ characters are removed. Third, to eliminate redundancy, a simple deduplication strategy is employed on completion files based on the file name and the name of the repository to which they belong. Finally, up to 1000 of the most recently updated unique completion files are selected from each repository. The remaining repository snapshot is retained without additional processing.

C Training Details

C.1 Hyperparameters

The optimization process was conducted using the AdamW optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.999$, and a weight decay of 0.01. A batch size of 128 was employed, with a micro-batch size of 1 to accommodate hardware constraints. To ensure stable training, gradient clipping was applied with a maximum gradient Euclidean norm of 2. The learning rate was managed using a cosine decay scheduler with a linear warm-up phase, where the maximum learning rate was set to 5×10^{-5}. The warm-up phase lasted for 256 iterations, after which the learning rate followed a cosine decay schedule for 3244 additional iterations, reaching a minimum value of 5×10^{-8}.

C.2 Training Mode of Context Composers

For training, we obtain an input sequence from each row of the composed dataset by independently tokenizing the context string and the completion file. This process ensures that the completion sequence does not exceed 4,096 tokens and that the total length of the concatenated input remains within 16,384 tokens. To enforce these constraints, we apply truncation from the left for the context and from the right for the completion. Given that most composed contexts exhibit high token saturation, we maintain a context-to-completion token ratio exceeding $3 : 1$.

D Evaluation Details

D.1 Evaluation Mode of Context Composers

For evaluation, we obtain an input sequence for each row of the composed dataset by tokenizing the concatenation of the context string and the completion file. We then apply truncation from the left.

Compared to the training mode (see Appendix C.2), we do not fix the maximum sequence length. Instead, we treat it as a parameter that can be adjusted based on the evaluation requirements. For example, in Appendix F, we demonstrate the dependency between checkpoint quality and maximum sequence length.

We use the following four evaluation composers in our tables:

- **FL-4K**: File-Level composer with maximum sequence length 4K tokens. We use it to estimate how hard the task is without any repository-context, and as a reference point for calculating gains.
- **PD-4K**: Path Distance composer with maximum sequence length 4K tokens. We use it to estimate model's in-context learning capabilities with initial input sequence length (4K tokens).
- **PD-16K**: Path Distance composer with maximum sequence length 16K tokens. We use it to estimate model's in-context learning capabilities with new input sequence length (16K tokens), and this is the main composer to compare repository-level pretraining with different context composers.
- **Or-16K**: original pretraining composer in evaluation mode with maximum sequence length 16K tokens. We use it to identify the most promising composer overall. This composer applies only to our checkpoints.

E Comprehensive Compilation of Evaluation Results

We present results of our experiments in Table 3. They can be used as baselines for further research.

We additionally include results for the base model with RoPE's base frequencies (θ) being 10,000 and 500,000, results for pretraining with file-level composer for the same values of θ. These results demonstrate that RoPE adjustments impact model quality, and that the model with initial base frequency performs on zero-level for long contexts even after finetuning.

When using FL-4K composer, the model successfully recovers its quality after RoPE adjustments, suggesting that file-level data alone is sufficient to restore performance. The initial model shows strong in-context learning capabilities for the PD-4K composer, with it outperforming file-level inference. This advantage persists after repository-level pretraining, indicating that training on the collected data effectively retains model's ability to utilize relevant context for shorter context size.

For the PD-16K composer, the initial model, without RoPE adaptation, fails completely, but RoPE scaling alone improves Exact Match scores. Further pretraining yields gains of +19 for file-level training and +22 for the best composer in *inproject* category, with all final scores being slightly higher than file-level pretraining performance. This suggests that adapting to the longer context window, rather than the specific sequence composition, is the primary factor in repository-level code completion, with context composers contributing only marginally for suggested approaches (+3 points for *inproject* category).

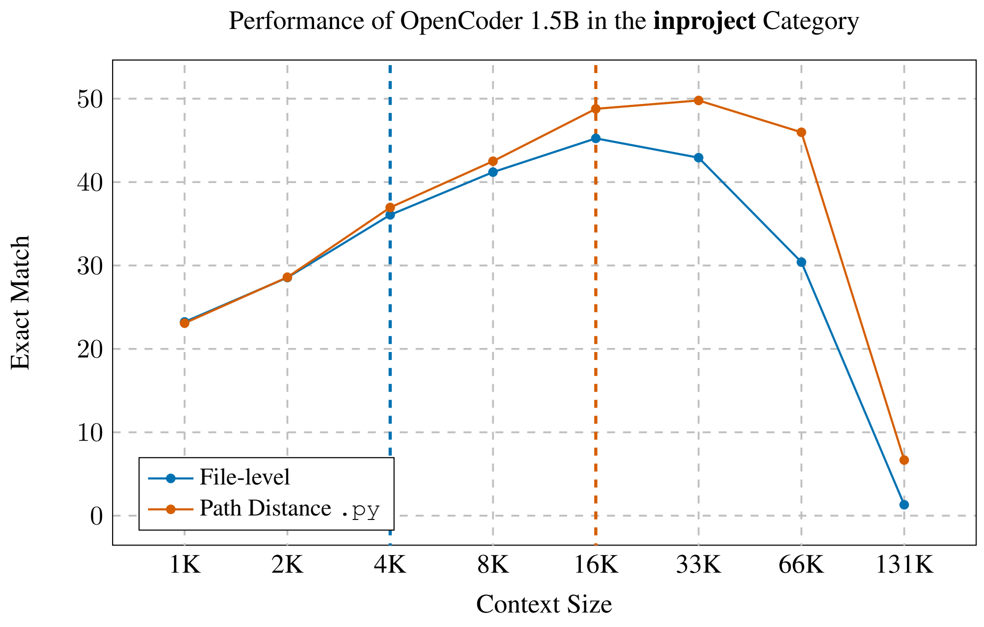
Overall, our findings emphasize that RoPE adaptation is the dominant factor in long-context performance gains, while sequence composition plays a secondary role. Future work should explore more effective retrieval-based strategies to maximize repository-level context utilization.

F Performance Scaling Beyond Training Context Window

The repository-level pretraining with File-level composer and Path Distance composer for maximum sequence lengths of 4K and 16K. However, pretrained checkpoints extrapolate beyond these lengths up to 16K and 32K as snown on Figure 2.

Table 3: Results of evaluating all checkpoints after repository-level pretraining on all evaluation composers. Evaluation dataset is LCA-large for the line categories: *inproject* and *infile*. Highlighted column is the main column for in-context learning capabilities comparison.

Pretraining Composer	**inproject**				**infile**			
	FL-4K	PD-4K	PD-16K	Or-16K	FL-4K	PD-4K	PD-16K	Or-16K
Base model (no training)								
$\theta = 10{,}000$	26.4	36.6	0.0	—	32.6	38.2	0.0	—
$\theta = 500{,}000$	13.5	16.6	9.8	—	15.5	12.9	4.5	—
File-level 4K								
$\theta = 10{,}000$	26.2	36.4	0.0	26.2	32.7	38.1	0.0	32.7
$\theta = 500{,}000$	25.9	36.1	45.2	25.9	33.0	38.1	44.6	33.0
Path Distance `.py`	26.2	37.0	48.8	48.8	33.1	38.7	47.6	48.8
reversed	26.1	36.9	48.3	43.2	32.9	38.8	47.5	44.0
irrelevant	25.8	36.5	47.9	26.7	32.5	38.1	46.7	33.4
Lines IoU `.py`	25.7	36.3	48.7	51.8	33.2	38.4	47.7	50.1
reversed	26.1	36.8	48.4	43.5	33.2	38.9	47.4	44.6
irrelevant	25.8	36.4	47.5	26.7	32.7	38.4	46.6	33.4
Code Chunks `.py`	25.9	36.5	47.9	47.8	32.8	38.2	47.5	47.9
reversed	26.1	36.5	47.8	41.3	32.8	38.3	47.4	43.0
irrelevant	25.8	36.5	47.7	26.9	32.3	37.9	46.3	33.2
Half-memory `.py`	25.7	36.0	47.4	38.6	32.9	38.4	46.6	38.7
reversed	25.7	36.2	47.3	35.0	32.9	38.2	46.5	36.9
irrelevant	25.8	36.0	47.0	27.5	32.4	37.7	46.5	33.0
Declarations `.py`	25.9	36.5	46.8	28.2	32.6	38.1	46.1	34.4
reversed	25.7	36.5	46.9	28.1	32.7	38.1	45.7	34.2
irrelevant	26.2	36.3	47.2	28.2	32.4	38.4	45.6	33.9
Text Chunks `.py`	26.1	36.6	47.5	26.9	33.0	38.5	46.9	33.2
reversed	26.0	36.1	47.4	26.8	32.9	38.5	46.2	33.5
irrelevant	25.9	36.4	47.2	26.8	32.7	38.5	46.2	33.8
Text files	25.9	36.2	47.1	26.9	33.0	38.6	46.4	33.5
reversed	26.0	36.5	46.9	26.7	33.2	38.4	46.2	33.1
irrelevant	26.0	36.5	47.1	27.0	32.7	38.2	46.3	33.7
Random files	26.2	37.0	48.1	29.8	32.8	38.3	47.3	34.2
Random `.py`	25.9	36.8	48.4	31.9	32.8	38.1	47.0	35.3
Mixed context	26.2	36.7	48.5	31.0	32.6	38.2	47.5	36.4
Random tokens	26.0	36.2	44.5	26.0	32.6	37.9	45.1	33.1
Duplication	19.6	28.8	34.7	96.7	24.5	27.0	28.1	95.0
Leak	24.9	34.8	46.1	82.9	30.8	35.5	43.6	81.6
reversed	24.3	34.7	45.6	83.8	30.8	35.3	42.8	81.0
irrelevant	24.5	34.6	45.6	82.2	31.3	35.6	43.2	79.7
Masked Leak	25.2	35.4	46.4	65.5	31.6	36.9	45.0	63.5



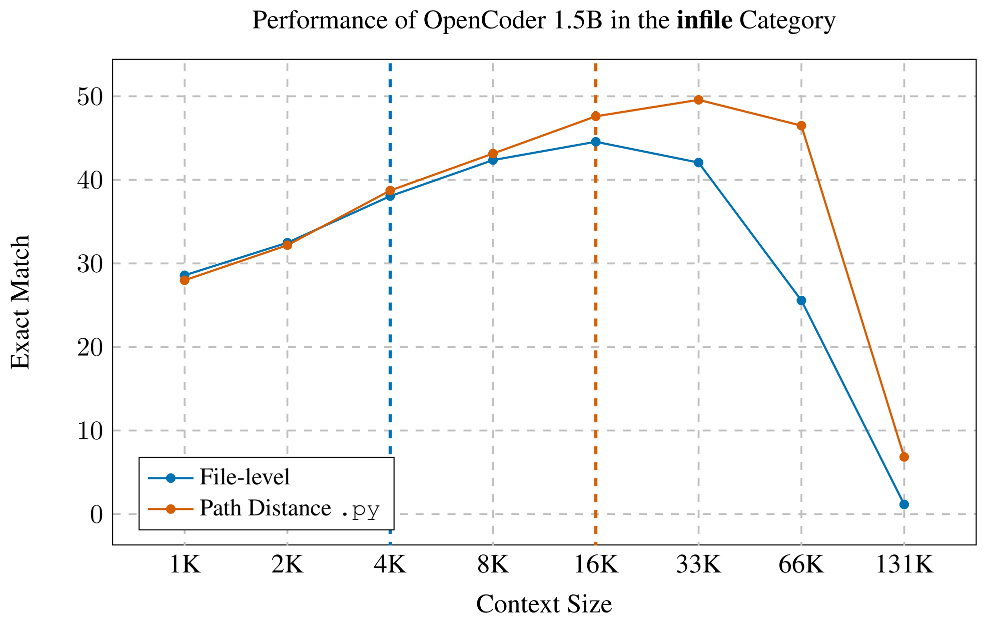


Figure 2: Performance comparison of File-level and Path Distance `.py` approaches across different context sizes for OpenCoder 1.5B model. The plots show the Exact Match accuracy for both inproject (top) and infile (bottom) categories. The dashed vertical lines represent the context length used during repository-level pretraining.

Similar behavior was observed in Roziere et al. (2023) and it needs additional research.

G Masked Loss and Full Loss Results

Some context composers create out-of-distribution sequences (*e.g.,* Declarations `.py`). We avoid distribution shift by masking the loss, *i.e.,* use only gradients from completion file tokens for training. In case of any composer that includes unprocessed code files in the context, we lose tokens for training. However, the results for each composer in Table 4 are comparable for masked loss and full loss pretraining, with the only exception being the Duplication composer.

Table 4: Comparison of checkpoints pretrained with masked loss and full loss.

Pretraining Composer	**inproject**				**infile**			
	FL-4K	PD-4K	PD-16K	Or-16K	FL-4K	PD-4K	PD-16K	Or-16K
Path Distance `.py`								
Masked loss	26.2	37.0	48.8	48.8	33.1	38.7	47.6	47.6
Full loss	26.3	36.5	48.4	48.4	33.1	38.6	47.8	47.8
Path Distance `.py`, *reversed*								
Masked loss	26.1	36.9	48.3	43.2	32.9	38.8	47.5	44.0
Full loss	26.2	36.8	48.4	43.1	33.1	38.6	47.2	44.1
Path Distance `.py`, *irrelevant*								
Masked loss	25.8	36.5	47.9	26.7	32.5	38.1	46.7	33.4
Full loss	25.5	36.5	48.0	26.5	33.0	38.4	46.8	33.6
Lines IoU `.py`								
Masked loss	25.7	36.3	48.7	51.8	33.2	38.4	47.7	50.1
Full loss	25.9	36.6	48.4	51.2	32.9	38.8	47.5	49.6
Lines IoU `.py`, *reversed*								
Masked loss	26.1	36.8	48.4	43.5	33.2	38.9	47.4	44.6
Full loss	26.0	36.8	48.3	43.6	33.1	38.9	47.4	44.8
Lines IoU `.py`, *irrelevant*								
Masked loss	25.8	36.4	47.5	26.7	32.7	38.4	46.6	33.4
Full loss	26.2	36.8	48.2	26.6	33.3	38.7	47.2	33.3
Code Chunks `.py`								
Masked loss	25.9	36.5	47.9	47.8	32.8	38.2	47.5	47.9
Full loss	26.5	36.8	48.7	47.7	33.2	38.6	47.8	47.8
Code Chunks `.py`, *reversed*								
Masked loss	26.1	36.5	47.8	41.3	32.8	38.3	47.4	43.0
Full loss	26.4	37.0	48.7	41.5	33.3	38.8	47.8	43.4
Code Chunks `.py`, *irrelevant*								
Masked loss	25.8	36.5	47.7	26.9	32.3	37.9	46.3	33.2
Full loss	25.7	36.5	48.1	26.9	33.3	38.7	47.1	33.3
Random `.py`								
Masked loss	25.9	36.8	48.4	31.9	32.8	38.1	47.0	35.3
Full loss	26.2	36.8	48.3	32.2	33.1	38.5	47.6	36.0
Duplication								
Masked loss	19.6	28.8	34.7	96.7	24.5	27.0	28.1	95.0
Full loss	25.5	35.9	46.4	97.3	32.8	38.4	44.4	96.4