RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems

Tianyang Liu Canwen Xu Julian McAuley
University of California, San Diego
{til040, cxu, jmcauley}@ucsd.edu

Abstract

Large Language Models (LLMs) have greatly advanced code auto-completion systems, with a potential for substantial productivity enhancements for developers. However, current benchmarks mainly focus on single-file tasks, leaving an assessment gap for more complex, real-world, multi-file programming scenarios. To fill this gap, we introduce RepoBench, a new benchmark specifically designed for evaluating repository-level code auto-completion systems. RepoBench supports both Python and Java and consists of three interconnected evaluation tasks: RepoBench-R (Retrieval), RepoBench-C (Code Completion), and RepoBench-P (Pipeline). Each task respectively measures the system's ability to retrieve the most relevant code snippets from other files as cross-file context, predict the next line of code with cross-file and in-file context, and handle complex tasks that require a combination of both retrieval and next-line prediction. RepoBench aims to facilitate a more complete comparison of performance and encouraging continuous improvement in auto-completion systems.

1 Introduction

Large language models (LLMs) [8, 10, 39, 28] have been instrumental in paving new avenues for innovative applications across diverse domains, with programming being a notably attractive and promising domain [9, 41, 4, 43]. In particular, the rise and application of code auto-completion systems like GitHub's Copilot [1], driven by OpenAI's Codex [9], have the potential to substantially changed the manner in which we interact with code. These changes facilitate coding for beginners and improve efficiency of the coding process for experienced developers.

A variety of code auto-completion models [9, 18, 15, 27, 23, 2] have emerged in recent years, each boasting unique capabilities and performance characteristics. This emergence of models emphasizes the increasing importance of AI in the realm of programming, leading to a more diversified and competitive landscape. However, current evaluation datasets and benchmarks [26, 34, 3] predominantly focus on completion tasks within the scope of a single file. This focus fails to reflect the complexity and intricacies of real-world programming scenarios, where developers frequently work on multi-file projects, often navigating through and understanding code spanning several repositories.

Recognizing the need for a more comprehensive evaluation, we introduce RepoBench, a new benchmark for evaluating the effectiveness of repository-level code auto-completion systems. Specifically, RepoBench offers three distinct evaluation sub-tasks, each emphasizing a unique aspect of a fully functioning code auto-completion system: (1) **The Retrieval Task (RepoBench-R)**, which tests the system's ability to retrieve the most relevant code snippets, thereby providing the necessary context for the prediction of the next line of code. (2) **The Code Completion Task (RepoBench-C)**, where the task is to predict the next line of code given a pre-defined context. The context can involve

[1] https://github.com/features/copilot

content from different files (cross-file context) and within the file (in-file context) with a moderate length setting that can fit most models. (3) **The End-to-End Pipeline Task (RepoBench-P)**, which is designed to simulate the complete process of a code auto-completion system like GitHub Copilot - first retrieving relevant snippets and then completing the code by predicting the next line. In this scenario, the system may encounter a large set of potential snippets for retrieval, resulting in longer and broader contexts, which leads to the need for the system to optimize the efficient selection of numerous candidates to facilitate code completion while ensuring that the extensive context remains within the system's processing capabilities.

To summarize, the primary contributions of our work are as follows:

- We present RepoBench, a benchmark tailored for evaluating repository-level code auto-completion systems. This benchmark comprises three interconnected tasks: RepoBench-R for code retrieval, RepoBench-C for code completion, and RepoBench-P, which integrates both aspects to reflect the entire workflow of an auto-completion system, offering a balanced assessment.
- We conduct a series of experiments on RepoBench, analyzing the efficacy of various retrieval methods and code completion models of different magnitudes, and the assessment of their combined performance in a full pipeline, providing some insights for future research and development. Our results underscore the significance of code models that can manage extended contexts and maintain generalizability in real-world coding environments.

2 Related Work

LLMs for Code Completion Code completion, also referred to as auto-completion or intelligent code completion, is an essential feature provided by many modern Integrated Development Environments (IDEs) and code editors. It aids programmers in writing code more efficiently by predicting and automatically completing the next line or multiple next lines. The inception of Language Models (LMs) in code completion can be traced back to the usage of n-gram based LMs [40, 20], RNN models [44], and probabilistic grammar-models [7, 35, 19], which laid the foundation for the subsequent introduction of more advanced LMs in this field. With the advent of transformer-based models [42, 11, 31, 32, 8], decoder-only models trained on large-scale code datasets have been proposed to foster the advancements in code completion. For instance, GPT-C [37] and CodeGPT [26] following the underlying architecture of GPT-style models are pre-trained on vast amounts of code. UniXCoder [18] and CugLM [24] incorporates multi-task learning strategies, and leverages code structures to enhance pretraining. More recent LLMs, including Codex [9], PolyCoder [46], CodeGen [27], In-Coder [15], CodeGeeX [49], SantaCoder [2], and StarCoder [23], employ billions of parameters and excel in code generation tasks, benefiting from large-scale, high-quality code corpora. The scope of code completion has expanded with works like RLPG [36], CoCoMIC [12], and RepoCoder [48], emphasizing the integration of in-file and cross-file contexts and the importance of specialized benchmarks for evaluating repository-level code autocompletion systems.

Code Completion Datasets The task of code completion serves as a foundation for programming language models and plays a pivotal role in intelligent code completion systems. While public benchmarks like CodeXGLUE [26] with datasets *PY150* [34] and *Github Java Corpus* [3] play a key role in evaluating models within single-file contexts, they may not fully encapsulate the intricacies of real-world coding projects which often entail cross-file interactions. To address this, Ding et al. [12] proposed CoCoMIC, a model for cross-file completion and a code completion dataset with retrieved cross-file context. Different from the CoCoMIC data, our benchmark extends beyond code completion and includes evaluation of retrieval and pipeline construction, thus can better capture the complexity of such cross-file code completion systems. RepoEval by Zhang et al. [48] serves as a project-oriented benchmark, focusing on 16 selected Python repositories to simulate real-world coding environments. However, its limitation arises from being integrated into the training data of StarCoder. RepoBench not only spans a wider range of repositories across Python and Java, but also offers a segmented evaluation into retrieval, completion, and end-to-end tasks.

Transitioning from file-based to repository-level code completion not only offers a more realistic representation of practical coding scenarios but also serves as a platform for evaluating the transfer learning capabilities of language models, as most models are not initially pre-trained with cross-file contexts included. This shift also introduces the challenge of handling longer prompts, a situation less common in single-file contexts, and a known limitation of many Transformer-based models. Recent research on long-range transformers [47] has shown promise in handling long sequences, with notable contributions from initial works like LongFormer [5] and Reformer [21], as well as more recent advancements like CoLT5 [1], UnlimiFormer [6], and Claude-100k [30], which has demonstrated their potential in effectively processing and generating code with much more cross-file context included.

3 The RepoBench Dataset

RepoBench is benchmark for auto-code completion, combining two preprocessed sources that serve different roles in the benchmarking process.

3.1 Data Sources

Github-Code Dataset: The first source of RepoBench is the `github-code` dataset[2], which consists of a vast collection of code files sourced from GitHub repositories under open-source licenses with a data cutoff date of *March 16, 2022*. Specifically, we aggregate files based on their repository name as the github-code dataset is originally stored at the file-level. Given that the code in this dataset has been widely utilized for training various models [23, 27], we primarily use this dataset for constructing our training data. The use of this data for training specifically addresses the adoption of patterns that concatenate cross-file context and in-file context for next-line prediction. Fine-tuning on this dataset is optional, as sufficiently robust models may already exhibit this generalizability.

Newly Crawled GitHub Data: To mitigate impacts regarding data leakage and memorization, we augment the dataset by incoporating the most recent, non-forked GitHub repositories that are permitted under their respective licenses. Specifically, we use GitHub's official API to crawl Python and Java repositories created after *February 9, 2023*, which aligns with the newest knowledge cutoff date of The Stack [22], and before *August 3, 2023*. This newly-crawled data serves exclusively as our test set for evaluation.

3.2 Data Processing

The data processing procedure for this study involves multiple steps. For the training data sourced from `github-code`, repositories with a number of Python or Java files between 32 and 128 are selected. This range is chosen to ensure an adequate cross-file dependency while avoiding excessive complexity and keeping the data volume within a reasonable range. While for the newly crawled test data, we do not set file number constraints to ensure a thorough evaluation. To identify cross-file dependencies and their usage, we use tree-sitter[3] to parse each file. This parsing is primarily directed at import statements, enabling us to identify all cross-file modules and the lines utilizing these modules (termed cross-file lines). Further, we track the corresponding code snippets that define these imported modules.

After processing the data, our dataset comprises 10,345 Python and 14,956 Java historical repositories, serving as training data and are available for optional fine-tuning. Additionally, we have 1,075 Python and 594 Java new repositories from GitHub designated as test data for evaluation.

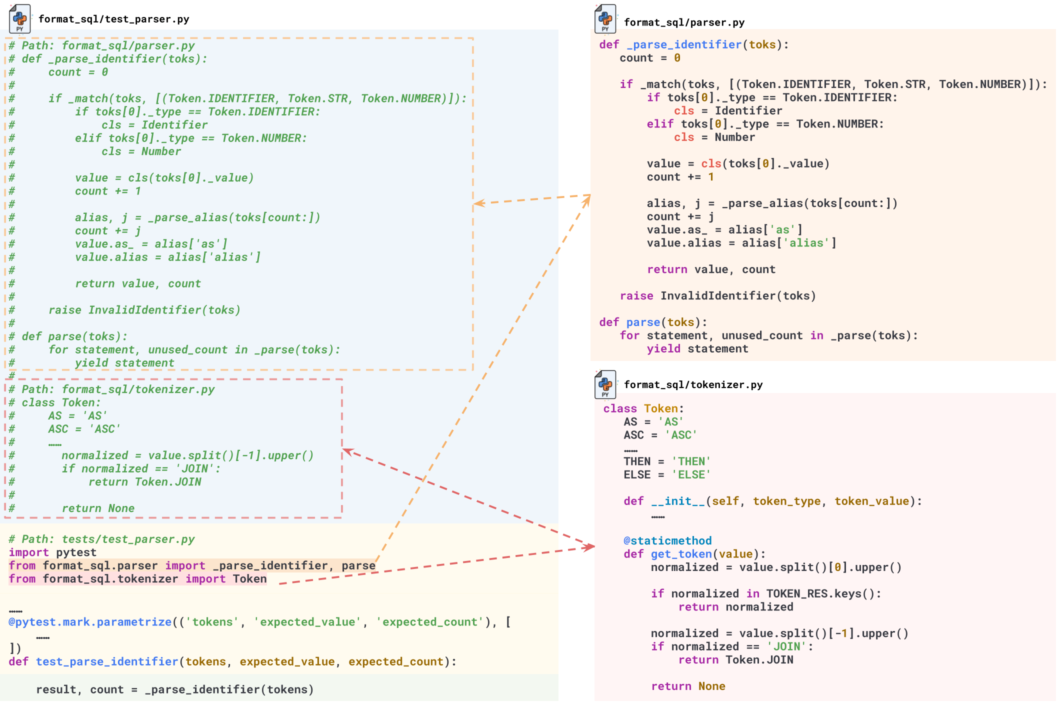
3.3 Task Construction

Task Settings To effectively evaluate next-line prediction in auto-completion systems, we define three settings:

- **Cross-File-First (XF-F):** This is the most challenging setting, where we mask the first appearance of a cross-file line within a file. In this setting, there is no prior usage of the module in the in-file context to aid the prediction, thereby requiring the system to handle long-range cross-file context for better accuracy.

[2] `https://huggingface.co/datasets/codeparrot/github-code`

[3] `https://tree-sitter.github.io/tree-sitter/`



format_sql/test_parser.py

```python
# Path: format_sql/parser.py
# def _parse_identifier(toks):
#     count = 0
#
#     if _match(toks, [(Token.IDENTIFIER, Token.STR, Token.NUMBER)]):
#         if toks[0]._type == Token.IDENTIFIER:
#             cls = Identifier
#         elif toks[0]._type == Token.NUMBER:
#             cls = Number
#
#         value = cls(toks[0]._value)
#         count += 1
#
#         alias, j = _parse_alias(toks[count:])
#         count += j
#         value.as_ = alias['as']
#         value.alias = alias['alias']
#
#         return value, count
#
#     raise InvalidIdentifier(toks)
#
# def parse(toks):
#     for statement, unused_count in _parse(toks):
#         yield statement
#
# Path: format_sql/tokenizer.py
# class Token:
#     AS = 'AS'
#     ASC = 'ASC'
#     ……
#       normalized = value.split()[-1].upper()
#       if normalized == 'JOIN':
#           return Token.JOIN
#
#       return None

# Path: tests/test_parser.py
import pytest
from format_sql.parser import _parse_identifier, parse
from format_sql.tokenizer import Token


……
@pytest.mark.parametrize(('tokens', 'expected_value', 'expected_count'), [
    ……
])
def test_parse_identifier(tokens, expected_value, expected_count):

    result, count = _parse_identifier(tokens)
```

format_sql/parser.py

```python
def _parse_identifier(toks):
   count = 0

   if _match(toks, [(Token.IDENTIFIER, Token.STR, Token.NUMBER)]):
       if toks[0]._type == Token.IDENTIFIER:
           cls = Identifier
       elif toks[0]._type == Token.NUMBER:
           cls = Number

       value = cls(toks[0]._value)
       count += 1

       alias, j = _parse_alias(toks[count:])
       count += j
       value.as_ = alias['as']
       value.alias = alias['alias']

       return value, count

   raise InvalidIdentifier(toks)

def parse(toks):
   for statement, unused_count in _parse(toks):
       yield statement
```

format_sql/tokenizer.py

```python
class Token:
   AS = 'AS'
   ASC = 'ASC'
   ……
   THEN = 'THEN'
   ELSE = 'ELSE'

   def __init__(self, token_type, token_value):
       ……

   @staticmethod
   def get_token(value):
       normalized = value.split()[0].upper()

       if normalized in TOKEN_RES.keys():
           return normalized

       normalized = value.split()[-1].upper()
       if normalized == 'JOIN':
           return Token.JOIN

       return None
```

Figure 1: Construction of a prompt for repository-level cross-file code completion. The commented cross-file context (path + snippet), parsed from import statements using `tree-sitter`, is concated with the in-file context (path + import statements + preceding lines), which cropped to a maximum of 30 lines in RepoBench to form the input prompt, with the objective is to predict the next line. Note that for clarity, certain lines of code are omitted in this figure, which is an abbreviated and simplified version derived from a real example. Refer to Appendix A for a detailed ablation study on prompt construction.

- **Cross-File-Random (XF-R):** In this setting, we mask a random and non-first occurrence of a cross-file line. Unlike the XF-F setting, the prior in-file usage of the module may serve as a hint for the prediction.

- **In-File (IF):** In this setting, we mask an in-file line that does not involve any cross-file modules. This setting serves as a robustness test to ensure that the incorporation of cross-file context does not greatly affect the accuracy of predictions.

Note that RepoBench-R (Retrieval) is designed with only XF-F and XF-R settings, as IF does not involve retrieval and thus cannot be evaluated in this task, while both RepoBench-C (Code Completion) and RepoBench-P (Pipeline) involve all three settings: XF-F, XF-R, and IF.

RepoBench-R RepoBench-R targets the retrieval component of a repository-level auto-completion system, focusing on extracting the most relevant code snippet from a project repository for next-line code prediction.

In RepoBench-R, every snippet parsed from import statements is treated as a potential candidate for next-line prediction, where only one 'gold snippet' is the optimal context for prediction. This task considers scenarios with 5 or more candidate snippets, and specifically, we categorize them into two subsets: those with 5-9 candidates as the *easy* subset, and those with 10 or more candidates as the *hard* subset. As demonstrated in Table 1 (top), both the *easy* and *hard* subsets contain 12,000 samples for the XF-F setting, whereas for the XF-R setting, each subset consists of 6,000 samples. We also provide training data for optional usage, further details can be also located in Table 1 (bottom). For evaluative purposes, the Accuracy@k (acc@k) metric is employed to assess retrieval performance. The *easy* subset is evaluated using acc@1 and acc@3, while the *hard* subset is examined through acc@1, acc@3, and acc@5 metrics.

Table 1: *(Top)* Test data overview for RepoBench across Python and Java for 3 different tasks; *(Bottom)* Training data for RepoBench across Python and Java.

Lang.	Task	Subset	XF-F	XF-R	IF	Mean Candidates	Mean Tokens
Python	RepoBench-R	Easy	12,000	6,000	-	6.7	-
		Hard	12,000	6,000	-	17.8	-
	RepoBench-C	2k	12,000	5,000	7,000	-	1,035
		8k	18,000	7,500	10,500	-	3,967
	RepoBench-P		10,867	4,652	6,399	24	44,028
Java	RepoBench-R	Easy	12,000	6,000	-	6.8	-
		Hard	12,000	6,000	-	25.5	-
	RepoBench-C	2k	12,000	5,000	7,000	-	1,093
		8k	18,000	7,500	10,500	-	4,179
	RepoBench-P		10,599	4,459	6,196	26	139,406

Language	Task	XF-F	XF-R	IF
Python	Code Retrieval	175,199	86,180	-
	Code Completion	349,023	179,137	214,825
java	Code Retrieval	340,121	216,642	-
	Code Completion	683,890	447,464	709,218

RepoBench-C RepoBench-C simply focuses on the prediction of the next line of code, given a set of in-file context (including several preceding lines and import statements), and cross-file context.

In RepoBench-C, as shown in Figure 1 the prompt is created by combining all the parsed snippets as cross-file contexts and an in-file context. The in-file context includes import statements and several preceding lines of code with a maximum limit of 30 lines. To address the varying context length in existing models, RepoBench-C is divided into two subsets: RepoBench-C-2k and RepoBench-C-8k. RepoBench-C-2k, designed for models with a token limit of 2,048, holds prompts that do not exceed 1,925 tokens. Concurrently, RepoBench-C-8k is architected with a higher threshold, encompassing up to 7,685 tokens, apt for models with an 8,192 token limit (e.g., StarCoder [23]) or 8,000 token limit (e.g., Codex [9]).

RepoBench-C is designed primarily for 0-shot learning, in order to examine the model's ability to handle long-range contexts. Despite this, we also provide a large amount of training data to allow fine-tuning, thereby enhancing the transfer capabilities of relatively smaller models, and for the test set, we allocate more data under XF-F settings compared with XF-R and IF settings. Details of this data are provided in Table 1. For evaluation metrics, we follow the previous work [26] to use Exact Match (EM) and Edit Similarity (ES) [38] to evaluate the accuracy of the predicted code line.

RepoBench-P RepoBench-P evaluates the model's performance by combining RepoBench-R and RepoBench-C: retrieval of relevant snippets and next-line code prediction, presenting a challenging pipeline task. This task mirrors complex real-world scenarios that a practical auto-completion system would face, assessing the model's comprehensive performance and flexibility.

In RepoBench-P, each setting (XF-F, XF-R, and IF) requires the model to first identify the most pertinent snippets and then employ these snippets as cross-file context in conjunction with the in-file context to predict the subsequent line. Contrary to specifying a maximum token limit, we define a minimum token threshold: 12,000 for Python and 24,000 for Java, and the gold snippet retrieval process requires a minimum of 10 candidates. Due to the substantial amount of data resulting from these constraints, we opt to down-sample to ensure parity between Java and Python datasets. Details of this data are provided in Table 1. For evaluating the predicted next line, we also use the Exact Match (EM) and Edit Similarity (ES) metrics, in line with the RepoBench-C setting.

Table 2: Baseline results of RepoBench-R on Python and Java retrieval tasks for *Easy* and *Hard* subset. The models we use are `codebert-base` for CodeBERT, `unixcoder-base` for UniXcoder.

	Retrieval	Model	Easy				Hard					
			XF-F		XF-R		XF-F			XF-R		
			acc@1	acc@3	acc@1	acc@3	acc@1	acc@3	acc@5	acc@1	acc@3	acc@5
Python	Random		15.68	47.01	15.61	46.87	6.44	19.28	32.09	6.42	19.36	32.19
	Lexical	Jaccard	20.82	53.27	24.28	54.72	10.01	25.88	39.88	11.38	26.02	40.28
		Edit	17.91	50.61	20.25	51.73	7.68	21.62	36.14	8.13	22.08	37.18
	Semantic	CodeBERT	16.47	48.23	17.87	48.35	6.56	19.97	33.34	7.03	19.73	32.47
		UniXcoder	**25.94**	**59.69**	**29.40**	**61.88**	**17.70**	**39.02**	**53.54**	**20.05**	**41.02**	**54.92**
Java	Random		15.33	45.98	15.43	46.13	5.59	16.88	28.15	5.64	16.91	28.18
	Lexical	Jaccard	15.33	46.73	19.08	52.00	7.15	19.92	32.12	9.22	22.65	34.17
		Edit	14.67	44.99	16.23	47.78	5.80	17.55	28.73	6.67	17.80	28.62
	Semantic	CodeBERT	15.38	46.77	16.27	45.98	6.07	17.42	28.94	5.92	17.73	28.53
		UniXcoder	**17.34**	**50.52**	**24.15**	**57.83**	**10.79**	**26.42**	**39.73**	**15.10**	**33.38**	**46.17**

4 Experiments

4.1 RepoBench-R

The primary objective of the retrieval task in RepoBench-R is to identify the most relevant code snippets to predict the next line given an in-file context. The process generally involves cropping certain lines of the in-file code before the predicted line, followed by the calculation of the degree of relevance (we use the term 'similarity' uniformly) between the cropped code and each candidate snippet. Formally, the general method for retrieval can be mathematically formulated as follows:

$$\underset{i\in\{1,\ldots,n\}}{\arg\overset{k}{\max}}\; f(C[-m:], S_i)$$

where C denotes the in-file code, S_i refers to the i-th candidate snippet, n is the total number of candidate snippets, m is the number of lines from the in-file code retained, k represents the top k candidates to be retrieved, and f is the function computing the similarity (or other scores) between the cropped in-file code and the candidate snippets.

Baseline In our baseline approach, three strategies are employed for the retrieval task: (1) **Random Retrieval** involves retrieving code snippets in a random manner, serving as a lower-bound benchmark against which we can compare the effectiveness of the other retrieval methods. To ensure the stability and reliability of our results, this random process is repeated 100 times and the outcomes are averaged. (2) **Lexical Retrieval** uses Jaccard Similarity and Edit Similarity to assess the relevance between the cropped code from the in-file context and the candidate code snippets. (3) **Semantic Retrieval** applies encoder models, including CodeBERT [14] based on BERT [11] and UnixCoder [18] based on UniLM [13] (we use the `Encoder-only` mode) to obtain the code embeddings. Cosine Similarity is employed to measure the semantic similarity between the cropped code and the candidate snippets. In baseline, we crop $m = 3$ lines from the in-file code as specified in the general method ($C[-m:]$), indicating the last three lines before the line to predict (Refer to the Appendix B for the ablation study on the number of lines kept). All computations for determining the similarity score are executed at the token level.

Results Table 2 presents a detailed comparison of different retrieval strategies in RepoBench-R. Upon analysis of the results, several key observations emerge: (1) **Superior Performance of UniXcoder:** The results highlight that UniXcoder [18], with its unique approach of multi-modal data representation learning and combined use of multi-modal contrastive learning (MCL) [17] and cross-modal generation tasks (CMG), consistently outperforms other methods. This demonstrates its advanced ability in capturing the semantic meaning of diverse code fragments, solidifying its standing as a robust model for such tasks. (2) **Jaccard Similarity Outperforms Edit Similarity:** The analysis indicates that Lexical Retrieval using Jaccard Similarity generally surpasses Edit Similarity. This suggests that the shared tokens between code fragments play a more important role than their precise arrangement in enhancing the retrieval performance. It emphasizes the importance of common terms and their semantic context in code retrieval tasks. (3) **Python Retrieval Shows Higher Accuracy**

Than Java: The language-specific results show that Python tasks typically show higher accuracy than Java across all retrieval methods. This discrepancy might be attributed to Python's simpler syntax and less verbose nature, potentially reducing the variability of similar code snippets. Additionally, the common Python practice of defining function arguments in close proximity to their corresponding function calls could provide valuable context that aids retrieval. Conversely, Java's extensive use of classes might complicate the retrieval task. These findings hint at the potential of refining retrieval strategies to better accommodate language-specific characteristics, particularly for languages with complex structures like Java.

4.2 RepoBench-C

The code completion task, RepoBench-C, aims to predict the next line of code based on a given in-file context (C_{in}), consisting of import statements and preceding lines before the target line, as well as a cross-file context (C_x), comprising snippets from other files parsed by import statements. This task commonly uses autoregressive language models trained on code for prediction. The formal expression of this task can be illustrated as follows:

$$P(Y) = \prod_{i=1}^{n} P(y_i | y_{<i}, C_x, C_{in}) \tag{1}$$

where $P(Y)$ is the joint probability of all tokens in the predicted sequence Y. The variable y_i denotes the i^{th} token in sequence Y, while $y_{<i}$ symbolizes the sequence of all preceding tokens. C_x and C_{in} represent the cross-file context and the in-file context, respectively. This product notation represents the autoregressive assumption that each token y_i is conditionally dependent on all preceding tokens $y_{<i}$ and the given contexts C_x and C_{in}.

Baseline To establish a performance baseline, our benchmark compares 3 prominent models: (1) **Codex** [9] (i.e. `code-davinci-002`), developed by OpenAI, is recognized for its code generation capabilities, stands as the current SOTA model in the field, and serves as the base model for Copilot. (2) **CodeGen** [27] is a family of autoregressive language models for program synthesis, under 3 pre-training data variants (nl, multi, mono) and 4 model size variants (350M, 2B, 6B, 16B). Concretely, `CodeGen-NL` is trained on the Pile [16] including English text and programming language data, while `CodeGen-Multi` extends its capabilities by incorporating code data from multiple programming languages, and `CodeGen-Mono` enhances it further by specializing in Python code. (3) **StarCoder** [23] comprises 15.5B parameter models trained across over 80 programming languages, including a base version (`StarCoderBase`) and a Python-specialized version (`StarCoder`). To ensure uniformity in our evaluations, we adopt the following model-language pairings: `CodeGen-Mono` and `StarCoder` for Python, and `CodeGen-Multi` and `StarCoderBase` for Java. Fine-tuning experiments are also performed on CodeGen-350M and CodeGen-2B using the training data specified in Table 3 for the 2k version of RepoBench-C.

Results Table 3 presents the comprehensive results of RepoBench-C. Our findings on the two RepoBench-C subsets provide several insights: (1) **Comparable Performance in Python for RepoBench-C-2k:** Codex, CodeGen, and StarCoder exhibit almost indistinguishable performance in Python. This outcome may result from a combination of factors, including model size and the knowledge cutoff of the models. (2) **Pronounced Performance Differences in Java for RepoBench-C-2k:** The evaluation on Java showcases a marked differentiation in model performance: Codex notably stands out as the superior model, followed by StarCoder, while CodeGen largely lags behind. (3) **Substantial Performance Gap for RepoBench-C-8k** For the evaluation on the much longer 8k set, a more significant divergence in performance is observed between StarCoder and Codex, with Java witnessing greatly pronounced differences. This observed trend may be indicative of underlying out-of-distribution challenges, i.e., a varying average performance of StarCoder across different lengths. For an in-depth discussion on this topic, please refer to Appendix D.

4.3 RepoBench-P

RepoBench-P combines the retrieval and code completion tasks to form a pipeline, where the goal is to first retrieve the most relevant code snippets given an in-file context and then predict the optimal next line of code based on the retrieved snippets and the in-file context. This pipeline approach aims to leverage the strengths of both tasks to enhance code assistance systems' capabilities. The formal

Table 3: RepoBench-C performance of CodeGen models with varying parameter sizes (350M to 16B), including original and fine-tuned (FT) versions, StarCoder and Codex (code-davinci-002), across Python and Java, evaluated using Exact Match (EM), Edit Similarity (ES). 'All' represents the average performance over the mixture of all test data, weighted by the size of each test setting.

	Model	Params.	XF-F		XF-R		IF		All	
			EM	ES	EM	ES	EM	ES	EM	ES
Python	CodeGen	350M	15.14	60.18	27.72	68.91	25.24	67.80	20.71	64.22
	CodeGen+FT	350M	16.19	62.82	27.88	69.64	23.81	66.83	20.85	65.41
	CodeGen	2.7B	22.09	64.94	34.48	72.67	31.27	70.93	27.35	68.30
	CodeGen+FT	2.7B	22.38	67.20	33.72	73.69	28.94	69.90	26.66	69.34
	CodeGen	6.1B	26.98	67.95	38.30	74.58	34.96	72.59	31.67	70.68
	StarCoder	15.5B	28.06	69.64	37.28	73.69	33.86	72.37	31.67	71.28
	CodeGen	16.1B	**28.86**	68.68	38.48	73.71	**37.60**	**73.78**	**33.41**	71.22
	Codex	175B	26.67	**70.29**	**39.40**	**76.39**	33.47	72.54	31.31	**72.22**
Java	CodeGen	350M	12.89	59.64	24.20	66.31	33.33	68.60	21.21	63.64
	CodeGen+FT	350M	16.12	65.10	25.88	72.08	33.91	71.57	23.34	68.44
	CodeGen	2.7B	20.77	66.65	30.06	69.96	39.97	72.93	28.31	69.17
	CodeGen+FT	2.7B	20.64	68.64	31.24	74.76	38.99	74.65	28.20	71.67
	CodeGen	6.1B	22.34	68.37	31.88	71.13	40.40	72.96	29.59	70.28
	StarCoder	15.5B	30.17	75.10	39.46	78.25	48.14	79.39	37.35	77.01
	CodeGen	16.1B	23.42	68.53	33.08	70.99	40.61	72.85	30.45	70.30
	Codex	175B	**35.16**	**78.05**	**45.58**	**82.09**	**53.09**	**82.20**	**43.14**	**80.22**

	Model	Params.	XF-F		XF-R		IF		All	
			EM	ES	EM	ES	EM	ES	EM	ES
Python	StarCoder	15.5B	23.00	66.29	33.01	72.85	29.00	68.79	26.84	68.39
	Codex	175B	**28.02**	**69.89**	**40.51**	**76.66**	**33.19**	**71.94**	**32.13**	**71.90**
Java	StarCoder	15.5B	20.10	66.14	30.39	70.87	41.23	73.29	28.41	69.21
	Codex	175B	**33.07**	**75.63**	**42.76**	**79.97**	**51.68**	**80.60**	**40.52**	**77.98**

expression of RepoBench-P can be represented as follows:

$$P(Y) = \prod_{i=1}^{n} P(y_i | y_{<i}, S_1, \ldots, S_k, C_{in}) \tag{2}$$

where $P(Y)$ denotes the joint probability of all tokens in the predicted sequence Y. y_i represents the i-th token in sequence Y, while $y_{<i}$ symbolizes the sequence of all preceding tokens. $S_1, \ldots, S_k$ refer to the retrieved code snippets, and C_{in} represents the in-file context. This product notation signifies the autoregressive assumption that each token y_i is conditionally dependent on all preceding tokens $y< i$, the given in-file context C_{in}, and the retrieved snippets $S_1, \ldots, S_k$.

Baseline To establish a performance baseline for the end-to-end task RepoBench-P, we test Codex (code-davinci-002) as the base model. For reference, the performance and analysis of StarCoder as a base model are also provided in Appendix E. We reserve 1,600 tokens for the in-file context, with a cropping limit of 60 preceding lines. Any unused tokens from this allocation are then filled by the cross-file context, up to a total prompt size of 6,400 tokens.

For the retrieval component, we delve into several strategies for retrieving relevant snippets from the cross-file context: (1) **Gold-Only**: In cross-file completions, the cross-file context includes just the 'gold snippet'. For in-file completions, the context is left empty. (2) **Gold-Filled**: The cross-file context integrates the 'gold snippet' alongside with other randomly fetched snippets until the 6,400-token capacity is filled. Inspired by the work of [25], we employ two variant strategies for the placement of the 'gold snippet': *Gold-Filled-Head*, where the 'gold snippet' is positioned at the beginning; and *Gold-Filled-Tail*, where it is positioned at the tail-end. (3) **UniXcoder**: Using UniXcoder as cross-file context retriever, snippets are obtained based on their relevance to the cropped preceding three in-file lines while adhering to a 6,400-token limit for the input length. The includes the *UniXcoder-H2L (High-to-Low)* variant, ranking snippets from the most to least relevant, and the *UniXcoder-L2H (Low-to-High)* approach, ranking in the reverse order. (5) **Random**: Cross-file

Table 4: Comparison of various retrieval strategies on the RepoBench-P for Python and Java using Codex [9] (code-davinci-002). Each strategy is evaluated in terms of Exact Match (EM) and Edit Similarity (ES) metrics for XF-F, XF-R, and IF settings. 'All' represents the average performance over the mixture of all test data, weighted by the size of each test setting. Strategies (*Gold-Only* and *Gold-Filled*), marked with an asterisk (*), include gold snippets for benchmarking purposes and serve only as references; they do not embody oracle capabilities.

	Retrieval Method	XF-F		XF-R		IF		All	
		EM	ES	EM	ES	EM	ES	EM	ES
Python	Gold-Only*	30.59	70.43	40.65	74.45	41.10	78.32	35.79	73.59
	Gold-Filled-Head*	31.07	70.48	39.77	74.42	41.87	78.56	36.07	73.68
	Gold-Filled-Tail*	31.35	70.73	40.56	74.37	41.21	78.50	36.18	73.77
	UniXcoder-H2L	30.99	70.68	**40.71**	**74.74**	**43.19**	79.18	36.61	74.02
	UniXcoder-L2H	**32.12**	**71.36**	40.59	74.48	43.07	**79.22**	**37.11**	**74.32**
	Random	28.06	68.95	38.75	73.81	41.16	78.34	34.15	72.72
	Baseline	26.75	68.16	37.60	73.30	40.79	78.26	33.15	72.20
Java	Gold-Only*	32.48	70.39	43.51	76.49	55.91	81.65	41.62	74.95
	Gold-Filled-Head*	32.48	70.29	43.44	76.36	55.84	81.68	41.59	74.88
	Gold-Filled-Tail*	32.37	70.30	43.48	76.63	55.66	81.55	41.49	74.91
	UniXcoder-H2L	32.46	70.16	42.79	76.14	**56.71**	81.86	41.70	74.83
	UniXcoder-L2H	**32.69**	**70.33**	**43.08**	**76.21**	56.57	**81.89**	**41.83**	**74.93**
	Random	31.34	69.81	42.08	75.73	55.94	81.67	40.77	74.51
	Baseline	30.73	69.48	41.44	75.42	56.18	81.70	40.40	74.29

context snippets are totally randomly selected without considering their relevance until the token limit is reached. Due to the constraints imposed by the codex rate limit, we are unable to perform multiple runs for the random retrieval, which is necessary to mitigate the inherent randomness; consequently, the results presented should be considered indicative and not conclusive. (6) **Baseline**: In this strategy, a token limit of 6,400 is solely allocated for the in-file context, abstaining from using any cross-file context during completion. It serves as a fundamental point of comparison, highlighting the model's performance when exclusively dependent on the in-file context.

Results Table 4 presents a comparison of various retrieval strategies using Codex in RepoBench-P. From this comparison, we present the following insights: (1) **Inclusion of Cross-file Contexts Improves Performance:** Integrating more cross-file contexts enhances performance, irrespective of retrieval quality. Even randomly selected contexts significantly boost performance, potentially by fostering contextual understanding, enabling the model to draw from a broader code repository. (2) **Effective Retrieval Enhances Performance:** Retrievers deploying specific models or methods like UniXcoder model, outperform random retrieval systems. Notably, this improvement is not confined to cross-file line prediction (XF-F and XF-R) but is also observed in in-file next-line prediction (IF), highlighting the value of retrieving code related to current code in the same repository as cross-file contexts, even if the succeeding line does not encompass cross-file modules. (3) **Placement Order of Retrieved Code Snippets Matters:** The positioning of related code snippets influences code completion effectiveness. Positioning higher similar code snippets adjacent to or in close proximity to the line requiring completion tends to improve code completion performance.

5 Conclusion

In this paper, RepoBench is introduced as a benchmark designed for evaluating repository-level code auto-completion systems, conmprising three distinct yet interrelated tasks: RepoBench-R for code retrieval, RepoBench-C for code completion, and RepoBench-P for testing the complete auto-completion pipeline. These tasks collectively provide a diverse evaluation environment for the Python and Java programming languages. The paper underscores through evaluation experiments the need for models capable of handling longer and more complex contexts, akin to those encountered in real-world programming scenarios. RepoBench aims to serve as a benchmark that contributes to ongoing innovation in code intelligence.

References

[1] Joshua Ainslie, Tao Lei, Michiel de Jong, Santiago Ontañón, Siddhartha Brahma, Yury Zemlyanskiy, David Uthus, Mandy Guo, James Lee-Thorp, Yi Tay, Yun-Hsuan Sung, and Sumit Sanghai. Colt5: Faster long-range transformers with conditional computation, 2023.

[2] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. Santacoder: don't reach for the stars!, 2023. URL `https://arxiv.org/abs/2301.03988`.

[3] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pp. 207–216, 2013. doi: 10.1109/MSR.2013.6624029.

[4] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *ArXiv preprint*, abs/2108.07732, 2021. URL `https://arxiv.org/abs/2108.07732`.

[5] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *ArXiv preprint*, abs/2004.05150, 2020. URL `https://arxiv.org/abs/2004.05150`.

[6] Amanda Bertsch, Uri Alon, Graham Neubig, and Matthew R. Gormley. Unlimiformer: Long-range transformers with unlimited length input, 2023.

[7] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. PHOG: probabilistic model for code. In Maria-Florina Balcan and Kilian Q. Weinberger (eds.), *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pp. 2933–2942. JMLR.org, 2016. URL `http://proceedings.mlr.press/v48/bielik16.html`.

[8] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL `https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html`.

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *ArXiv preprint*, abs/2107.03374, 2021. URL `https://arxiv.org/abs/2107.03374`.

[10] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm:

Scaling language modeling with pathways. *ArXiv preprint*, abs/2204.02311, 2022. URL `https://arxiv.org/abs/2204.02311`.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, Minneapolis, Minnesota, 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL `https://aclanthology.org/N19-1423`.

[12] 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, 2022. URL `https://arxiv.org/abs/2212.10007`.

[13] Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. Unified language model pre-training for natural language understanding and generation. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 13042–13054, 2019. URL `https://proceedings.neurips.cc/paper/2019/hash/c20bb2d9a50d5ac1f713f8b34d9aac5a-Abstract.html`.

[14] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, Online, 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL `https://aclanthology.org/2020.findings-emnlp.139`.

[15] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis, 2022. URL `https://arxiv.org/abs/2204.05999`.

[16] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The Pile: An 800gb dataset of diverse text for language modeling. *ArXiv preprint*, abs/2101.00027, 2021. URL `https://arxiv.org/abs/2101.00027`.

[17] Tianyu Gao, Xingcheng Yao, and Danqi Chen. SimCSE: Simple contrastive learning of sentence embeddings. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 6894–6910, Online and Punta Cana, Dominican Republic, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.552. URL `https://aclanthology.org/2021.emnlp-main.552`.

[18] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7212–7225, Dublin, Ireland, 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.499. URL `https://aclanthology.org/2022.acl-long.499`.

[19] Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 763–773, 2017.

[20] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016.

[21] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL `https://openreview.net/forum?id=rkgNKkHtvB`.

[22] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The stack: 3 tb of permissively licensed source code. *Preprint*, 2022.

[23] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you!, 2023.

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

[25] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts, 2023.

[26] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *ArXiv preprint*, abs/2102.04664, 2021. URL `https://arxiv.org/abs/2102.04664`.

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

[28] OpenAI. Gpt-4 technical report, 2023.

[29] OpenNMT. Ctranslate2: A c++ and python library for efficient inference with transformer models. `https://github.com/OpenNMT/CTranslate2`, 2023.

[30] Anthropic PBC. Introducing 100k context windows. `https://www.anthropic.com/index/100k-context-windows`, 2023. Accessed: 2023-05-27.

[31] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[32] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[33] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models, 2020.

[34] Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. *SIGPLAN Not.*, 51(10):731–747, 2016. ISSN 0362-1340. doi: 10.1145/3022671.2984041. URL `https://doi.org/10.1145/3022671.2984041`.

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

[36] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Repository-level prompt generation for large language models of code, 2022. URL `https://arxiv.org/abs/2206.12839`.

[37] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer, 2020.

[38] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer, 2020.

[39] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *ArXiv preprint*, abs/2302.13971, 2023. URL `https://arxiv.org/abs/2302.13971`.

[40] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 269–280, 2014.

[41] Tim van Dam, Maliheh Izadi, and Arie van Deursen. Enriching source code with contextual data for code completion models: An empirical study, 2023.

[42] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 5998–6008, 2017. URL `https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html`.

[43] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. Codet5+: Open code large language models for code understanding and generation, 2023.

[44] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 334–345. IEEE, 2015.

[45] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Huggingface's transformers: State-of-the-art natural language processing, 2020.

[46] Frank F Xu, Uri Alon, Graham Neubig, and Vincent J Hellendoorn. A systematic evaluation of large language models of code. *ArXiv preprint*, abs/2202.13169, 2022. URL `https://arxiv.org/abs/2202.13169`.

[47] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontañón, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: Transformers for longer sequences. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL `https://proceedings.neurips.cc/paper/2020/hash/c8512d142a2d849725f31a9a7a361ab9-Abstract.html`.

[48] Fengji Zhang, Bei Chen, Yue Zhang, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation, 2023.

[49] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x, 2023.

A Ablation Study for Prompt Construction

In this appendix, we present a pilot study that focuses on constructing appropriate prompts for cross-file code completion. Our study is conducted using Codex, specifically, `code-davinci-002`, in an ablation setting where we systematically vary the prompt design. The following elements constitute the building blocks of our prompts:

1. **In-File Context (IFC)**: In-file contexts are the preceding n lines before the line we want to predict. We investigate two cases - **Short IFC (IFC-Short)** which crops a maximum of the preceding 30 lines, and **Long IFC (IFC-Long)** which crops a maximum of the preceding 120 lines.
2. **Import Statements (IS)**: To avoid losing the import statements due to cropping, we construct the prompt by concatenating all the IS before the IFC.
3. **Cross-File Context (XFC)**: Cross-file contexts are commented code snippets from other files parsed from import statements. We attach them at the beginning of the prompt.

Table 5: Ablation study comparing different combinations of Cross-File Context (XFC), Import Statements (IS), and In-File Context (IFC) with both short (IFC-Short) and long (IFC-Long) variants, using Codex [9]. The All score of Exact Match (EM) and Edit Similarity (ES) are calculated by averaging the three settings. The results are based on examples that are randomly sampled from the training set of RepoBench-C, with 5,000 examples for each of the three settings (XF-F, XF-R, IF) for Python and Java separately.

	Prompt Construction	XF-F		XF-R		IF		All	
		EM	ES	EM	ES	EM	ES	EM	ES
Python	IFC-Short	7.30	58.38	36.28	76.70	36.34	75.68	26.64	70.25
	IFC-Long	7.96	59.37	43.44	80.16	38.56	76.62	29.99	72.05
	IS+IFC-Short	22.28	69.30	38.90	78.03	38.46	77.25	33.21	74.86
	IS+IFC-Long	23.62	69.99	**44.82**	**80.44**	40.80	78.16	36.42	76.20
	XFC+IFC-Short	19.64	66.44	41.62	78.86	39.12	77.04	33.32	74.11
	XFC+IS+IFC-Short	**28.38**	**72.46**	43.30	79.68	**41.24**	**78.29**	**37.64**	**77.06**
Java	IFC-Short	7.00	57.41	33.30	74.57	57.76	80.94	32.69	70.97
	IFC-Long	7.32	57.95	37.12	76.74	59.74	81.76	34.73	72.15
	IS+IFC-Short	22.72	71.72	37.92	77.73	60.84	83.29	40.49	77.58
	IS+IFC-Long	23.36	72.34	40.68	79.01	62.98	84.48	42.34	78.61
	XFC+IFC-Short	23.58	68.02	41.86	79.25	60.42	82.29	41.95	76.52
	XFC+IS+IFC-Short	**31.80**	**75.03**	**44.62**	**81.21**	**63.44**	**84.57**	**46.62**	**80.27**

We consider 6 different combinations, as shown in Table 5. Our pilot study yields several notable results regarding cross-file code completion, as summarized below:

1. The integration of both ISC and XFC shows the best overall results and significantly enhances cross-file code completion performance, even though there may be duplicated information between XFC and ISC.
2. Including ISC and XFC improves not only cross-file completion but also in-file completion. This improved performance is observed even when the included snippets do not specifically target in-file completion.
3. For XF-R settings, where the module that the next line will use is possibly used in the IFX, which may provide hints for next-line prediction, the inclusion of a longer in-file context (IF-Long) appears to be beneficial for Python. This suggests that extended context within the same file can potentially help prediction if it is not the first usage. However, the combination of IFC with IS and XFC (XFC+IS+IFC) yields nearly comparable results, which highlights the role of cross-file context and import statements.

Thus, as shown in Figure 1, in our dataset, we leverage the prompt construction by adopting the XFC+IS+IFC-S strategy, incorporating both cross-file context, import statements and short in-file context as the input for LLMs.

B Ablation Study of Kept Lines for Retrieval

In this appendix, an ablation study is conducted to ascertain the optimal number of preceding lines to be retained during the retrieval of pertinent code snippets for the prediction line. The study evaluates four distinct retrieval methodologies, namely, two Lexical Retrievers (*Jaccard Similarity* and *Edit Similarity*) and two Semantic Retrievers (*CodeBERT* and *UniXcoder*). The experimental evaluation considers keeping 3, 5, 10, 20, 30, 60, and 120 lines for the retrieval process. For consistency, the model sizes selected for this ablation study align with the configurations delineated in Section 4.1.

The experiments are executed on subsets of the training data, encompassing 8,000 XF-F samples and 4,000 XF-R samples, each categorically divided into easy and hard subsets. The accompanying tables delineate the performance metrics for each retrieval method: *Jaccard Similarity* (Table 6), *Edit Similarity* (Table 7), *CodeBERT* (Table 8), and *UniXcoder* (Table 9). The ablation analysis aims to elucidate the influence of varying the number of retained lines on the performance of code retrieval in the code generation task. The empirical results suggest a general trend: as the number of retained lines increases, the performance of most retrievers tends to diminish, with the optimal performance typically achieved when retaining either 3 or 5 lines.

Table 6: Performance of Jaccard Similarity as retrieval method for different numbers of kept lines.

	Lines	Easy Level				Hard Level					
		XF-F		XF-R		XF-F			XF-R		
		acc@1	acc@3	acc@1	acc@3	acc@1	acc@3	acc@5	acc@1	acc@3	acc@5
Python	*Rand*	15.72	47.16	15.79	47.08	6.67	20.00	33.41	6.79	20.28	33.67
	3	**18.10**	**51.38**	24.40	55.73	11.24	30.04	**45.23**	15.43	35.25	49.53
	5	17.94	50.72	**24.68**	56.27	**11.51**	**30.66**	44.96	16.70	36.12	49.38
	10	17.65	49.83	24.60	**56.60**	11.29	29.07	44.44	**17.35**	**37.03**	**50.60**
	20	16.79	48.01	22.43	55.55	10.40	27.15	41.56	15.65	34.42	48.35
	30	16.18	47.93	22.07	54.10	9.09	25.10	39.64	14.25	32.25	46.23
	60	15.61	46.29	19.38	52.25	8.03	23.03	36.93	12.30	28.88	42.83
	120	15.02	45.67	18.32	50.88	7.03	21.43	35.04	10.45	26.27	40.30
Java	*Rand*	15.81	47.52	15.82	47.40	6.92	20.79	34.68	6.95	20.90	34.90
	3	**16.98**	**49.89**	**21.32**	52.90	**9.14**	**25.97**	**41.39**	12.15	29.70	43.45
	5	16.53	49.06	21.22	**54.30**	8.92	25.45	40.34	**12.25**	**30.05**	43.35
	10	15.82	46.54	21.25	54.20	8.06	24.20	38.42	12.07	29.12	**43.50**
	20	14.32	45.65	19.50	53.67	6.54	21.91	36.52	11.28	28.52	43.12
	30	14.05	45.02	18.88	52.38	6.31	20.95	35.80	11.12	28.35	42.95
	60	13.55	43.94	17.72	50.92	6.05	19.30	34.19	9.57	26.65	41.17
	120	13.21	43.14	17.20	49.45	5.51	18.38	33.10	9.00	25.50	39.73

Table 7: Performance of Edit Similarity as retrieval method for different numbers of kept lines.

	Lines	Easy Level				Hard Level					
		XF-F		XF-R		XF-F			XF-R		
		acc@1	acc@3	acc@1	acc@3	acc@1	acc@3	acc@5	acc@1	acc@3	acc@5
Python	*Rand*	15.72	47.16	15.79	47.08	6.67	20.00	33.41	6.79	20.28	33.67
	3	**16.56**	**48.74**	**20.38**	51.05	**9.54**	**25.60**	**39.77**	**11.97**	28.10	41.83
	5	16.53	48.59	20.08	50.75	9.40	25.06	39.51	11.58	**28.95**	**42.45**
	10	16.20	47.09	19.78	51.20	8.54	23.79	38.20	10.47	27.27	42.50
	20	15.41	45.81	18.75	**52.12**	7.98	22.35	36.54	9.98	26.05	41.58
	30	15.64	46.11	18.85	51.05	6.95	21.45	35.93	9.40	24.85	39.88
	60	14.49	44.73	18.65	49.30	6.78	20.12	33.66	8.03	23.05	36.83
	120	14.03	44.64	16.98	48.12	6.35	19.78	33.52	8.15	21.88	35.20
Java	*Rand*	15.81	47.52	15.82	47.40	6.92	20.79	34.68	6.95	20.90	34.90
	3	**16.65**	**49.14**	16.07	48.43	7.21	**22.44**	**36.88**	**8.20**	**22.73**	**36.62**
	5	16.04	48.33	16.28	48.25	7.12	22.15	36.12	**8.20**	21.82	35.60
	10	15.64	46.90	**16.40**	**49.05**	6.64	20.77	34.98	7.95	21.50	35.38
	20	15.78	46.60	16.10	47.90	6.81	20.23	34.25	7.15	21.93	35.15
	30	15.15	46.30	15.68	47.48	6.73	20.21	34.11	7.45	21.43	34.67
	60	15.01	45.91	14.80	46.30	6.10	19.38	33.17	6.48	19.68	33.85
	120	14.97	45.96	14.62	46.27	6.19	19.19	32.17	6.48	20.23	33.62

Table 8: Performance of CodeBERT [14] (`codebert-base`) as the retriever for different numbers of kept lines.

	Lines	Easy Level				Hard Level					
		XF-F		XF-R		XF-F			XF-R		
		acc@1	acc@3	acc@1	acc@3	acc@1	acc@3	acc@5	acc@1	acc@3	acc@5
Python	*Rand*	15.72	47.16	15.79	47.08	6.67	20.00	33.41	6.79	20.28	33.67
	3	**15.81**	46.36	**15.90**	45.95	6.83	20.66	**34.52**	**7.20**	19.85	33.60
	5	15.36	46.48	15.40	46.02	**7.19**	**21.01**	34.39	6.42	19.82	33.23
	10	15.41	46.17	15.55	45.85	6.86	20.21	34.17	5.92	19.82	32.75
	20	15.39	46.35	15.62	46.48	6.85	19.86	33.39	7.05	**21.62**	33.90
	30	14.91	46.65	15.62	**47.10**	6.84	20.11	33.19	6.90	20.47	**33.95**
	60	14.72	46.73	14.95	47.05	6.53	19.51	32.96	6.30	19.50	32.60
	120	14.71	**46.74**	15.02	46.98	6.62	19.35	32.80	6.60	20.42	33.05
Java	*Rand*	15.81	47.52	15.82	47.40	6.92	20.79	34.68	6.95	20.90	34.90
	3	16.21	48.35	15.90	47.67	7.38	**22.26**	36.99	6.55	20.20	34.40
	5	**16.26**	**48.73**	16.68	47.93	**7.58**	21.79	36.16	6.50	20.42	34.88
	10	15.35	48.19	16.48	48.25	7.00	21.50	35.62	**7.20**	20.62	34.40
	20	15.79	47.95	**17.35**	48.30	7.15	20.94	34.39	7.15	21.00	**36.38**
	30	15.78	47.58	17.25	**48.40**	6.61	20.86	34.81	6.73	21.00	35.68
	60	15.05	47.70	16.43	48.10	6.21	20.86	34.51	6.88	**21.77**	35.95
	120	14.80	47.21	16.28	48.33	6.15	20.31	34.04	6.73	21.02	35.48

Table 9: Performance of UniXcoder [18] (`unixcoder-base`) as the retriever for different numbers of kept lines.

	Lines	Easy Level				Hard Level					
		XF-F		XF-R		XF-F			XF-R		
		acc@1	acc@3	acc@1	acc@3	acc@1	acc@3	acc@5	acc@1	acc@3	acc@5
Python	*Rand*	15.72	47.16	15.79	47.08	6.67	20.00	33.41	6.79	20.28	33.67
	3	**27.02**	**60.14**	29.23	61.95	**19.86**	**41.40**	55.74	22.50	43.12	56.93
	5	25.99	59.96	**29.62**	**62.78**	19.38	41.30	**55.75**	**22.55**	**44.98**	**58.83**
	10	23.34	57.53	27.80	61.02	16.89	39.29	53.73	20.93	42.83	56.53
	20	20.20	54.47	24.25	56.62	12.83	32.10	47.00	16.38	36.27	50.62
	30	18.68	51.95	21.48	54.05	10.82	27.56	42.12	12.95	31.32	46.25
	60	17.05	48.65	19.07	51.02	8.03	22.82	36.61	10.30	26.75	40.27
	120	16.09	47.38	18.12	48.83	7.12	20.51	33.73	8.33	22.65	34.67
Java	*Rand*	15.81	47.52	15.82	47.40	6.92	20.79	34.68	6.95	20.90	34.90
	3	**20.44**	**54.25**	**27.25**	61.48	**13.79**	**35.73**	**51.54**	**21.05**	43.38	57.80
	5	19.24	53.01	26.47	**61.65**	13.18	33.98	50.18	20.15	**43.40**	**58.17**
	10	16.16	49.58	25.55	60.12	11.06	30.18	46.16	17.70	40.65	56.62
	20	14.69	46.64	21.93	57.98	8.96	26.20	41.90	14.80	35.75	51.45
	30	13.81	45.40	20.25	56.15	7.96	24.27	39.40	13.40	33.12	48.62
	60	12.85	43.66	17.47	51.25	6.40	20.72	34.79	9.35	26.57	42.33
	120	12.31	42.50	16.50	48.62	5.76	18.79	32.67	7.90	22.53	37.50

C Experiment Settings

All models (except codex and fine-tuned models) use CTranslate2 [29] for inference [4] and the model weights are sourced from Huggingface [45].The Codex model was inferenced via OpenAI's API with authorized usage rights, maintaining a rate limit of 20 queries or 40,000 tokens per minute despite its deprecation as of March 2023. The `code-davinci-002` model continues to be accessible free of charge upon application [5]. The results for Codex were obtained through queries conducted from January 2023 to September 2023. During inference for new token generation, all models are set with

[4] Due to limited experimental resources and the extensive scale of our experiments, we rely on quantized models and libraries known for fast inference speeds. This reliance potentially introduces discrepancies in our results from the orginal models due to quantization effects or bugs within these libraries.

[5] `https://platform.openai.com/docs/guides/code`

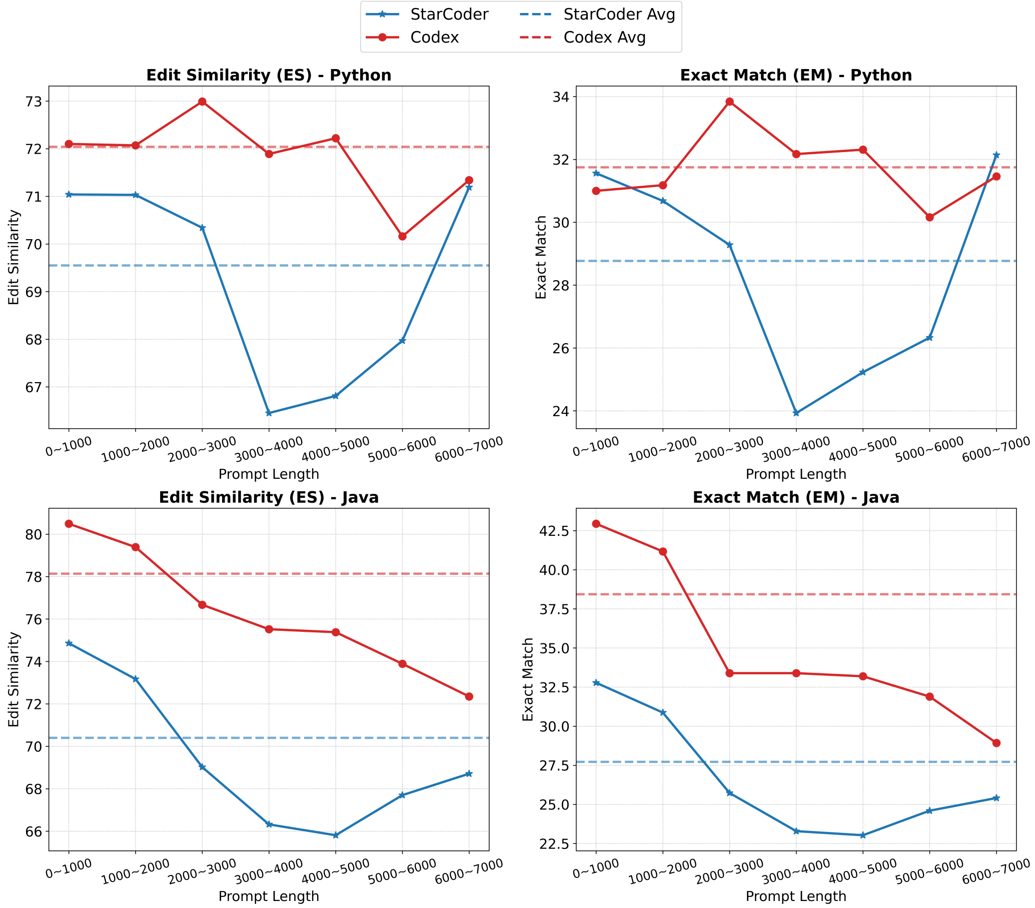


Figure 2: Performance comparison between Starcoder and Codex with respect to Edit Similarity (ES) and Exact Match (EM) metrics across various prompt lengths for Python (top row) and Java (bottom row). The left and right columns showcase the ES and EM scores, respectively. Note that suboptimal performance does not necessarily signify model limitations at specific prompt lengths, which might also be indicative of the inherent complexity or difficulty associated with the data at these lengths.

a temperature of 0.2, generating 64 tokens per next-line prediction, with the first non-comment line truncated as the output.

For fine-tuning, we sampled a total of 8,000 XF-F, 4,000 XF-R, and 4,000 IF data from the training set of RepoBench-C. Additionally, 200 data points were sampled for validation. The CodeGen model with 350M parameters was fine-tuned on 2 Nvidia A3090 GPUs, and the 2B model was fine-tuned on 4 Nvidia A3090 GPUs, both using Deepspeed [33]. Training was conducted with a learning rate of 1e-5 and a batch size of 32, applying early stopping based on the sampled validation set performance with a maximum of 3 epochs.

D Performance Analysis of StarCoder vs Codex

In order to illuminate the differential performance exhibited by StarCoder and Codex over varying context lengths, an extensive analysis was undertaken on the average performance metrics obtained from a total of 120,000 examples on RepoBench-C, for both Python and Java languages (with 60,000 examples for each), including both 2K and 8K lengths. Figure 2 show the their performances across different context lengths.

The analysis unveils a notable phenomenon pertinent to Python examples. Specifically, StarCoder tends to underperform with medium-length contexts, presenting a performance dip that is not observable in the Codex counterpart. The consistency in Codex's performance across different

Table 10: Comparison of various retrieval strategies on the RepoBench-P for Python and Java using `StarCoder` [23] (`StarCoderBase` for Java). Each strategy is evaluated in terms of Exact Match (EM) and Edit Similarity (ES) metrics for XF-F, XF-R, and IF settings. 'All' represents the average performance over the mixture of all test data, weighted by the size of each test setting. Strategies (*Gold-Only* and *Gold-Filled*), marked with an asterisk (*), include gold snippets for benchmarking purposes and serve only as references; they do not embody oracle capabilities.

	Retrieval Method	XF-F		XF-R		IF		All	
		EM	ES	EM	ES	EM	ES	EM	ES
Python	Gold-Only*	32.27	71.36	40.59	74.42	46.68	79.93	38.24	74.51
	Gold-Filled-Head*	31.10	70.75	40.52	73.78	43.79	79.18	36.80	73.85
	Gold-Filled-Tail*	31.20	70.81	40.00	73.67	43.40	79.03	36.63	73.82
	UniXcoder-H2L	30.99	70.74	40.35	73.94	43.60	79.11	36.66	73.86
	UniXcoder-L2H	31.40	70.70	**40.89**	73.72	43.85	79.30	37.05	73.85
	Random	29.17	69.73	38.80	73.13	43.48	78.98	35.39	73.15
	Baseline	**32.19**	**71.05**	40.80	**74.52**	**44.93**	**79.31**	**37.74**	**74.20**
Java	Gold-Only*	16.95	59.48	26.69	64.77	43.35	75.69	26.69	65.32
	Gold-Filled-Head*	17.67	60.05	27.07	65.18	37.17	69.27	25.33	63.81
	Gold-Filled-Tail*	17.32	59.72	26.93	65.27	38.17	70.16	25.41	63.93
	UniXcoder-H2L	17.72	59.78	26.31	64.25	37.83	69.35	25.38	63.51
	UniXcoder-L2H	17.86	59.87	26.53	64.84	38.41	69.85	25.67	63.82
	Random	16.24	59.01	25.72	63.99	36.81	68.92	24.23	62.94
	Baseline	**18.75**	**62.27**	**29.24**	**69.05**	**42.67**	**74.83**	**27.92**	**67.35**

context lengths suggests a more robust generalization capability, potentially attributed to its exposure to a diverse range of training data in terms of code length. In the case of Java examples, however, both StarCoder (employing the `StarCoderBase` model as delineated in the paper) and Codex demonstrate a declining performance trend as the context length increases. This universal trend across the two models reveals the potential relationship between prompt length and intrinsic difficulty, which means that that prompts of varying lengths might inherently differ in their levels of complexity and challenge, thereby impacting the performance as shown.

It is crucial and hard to interpret these results with caution, as the observed trends may be indicative of the models' exposure to training data with varied length distributions and the inherent complexity in prompts of different lengths. Future work should explore these aspects in depth to gain a more nuanced understanding of the underlying factors influencing the models' performance across diverse coding tasks and context lengths.

E StarCoder Performance on RepoBench-P

As previously noted, the performance of StarCoder is inconsistent across various metrics, rendering it an unreliable base model for evaluating the impact of different retrieval strategies on code completion tasks. Hence, we examine and discussion the results here for reference purposes.

The performance of StarCoder, as outlined in Table 10, elucidates some intriguing aspects. One noteworthy observation is that StarCoder often performs optimally when there is no retrieval of relevant code (the *Baseline* retrieval method). We present the following two ideas and speculations regarding this phenomenon. Firstly, StarCoder demonstrates limited generalizability at the repository-level code completion, which means it struggles to effectively adapt the pattern of repository-level code completion and gain insights from extensive cross-file snippets. Secondly, as conjectured in the last section, the input sequence length seems to play a crucial role in the model performance. This speculation is empirically supported by the performance under the in-file (*IF*) setting, where *Gold-Only* and *Baseline* exhibit discernible differences. Since there are no gold snippets for IF tasks, the only distinction between *Gold-Only* and *Baseline* in this context lies in the allocated token limit for in-file context – 1,600 tokens for *Gold-Only* and 6,400 for the *Baseline*. However, this increased token allocation does not consistently translate to improved performance, which is inconsistent with both cognition and the desired results observed by Codex.