Qeios

aiXcoder-7B: A Lightweight and Effective Large Language Model for Code Completion

Siyuan Jiang*
aiXcoder
Beijing, China
jiangsiyuan@aixcoder.com

Jia Li* ♂
Peking University
Beijing, China
lijia@stu.pku.edu.cn

He Zong
aiXcoder
Beijing, China
zonghe@aixcoder.com

Huanyu Liu, Hao Zhu
Peking University
Beijing, China
{huanyuliu, zhuhao}@stu.pku.edu.cn

Shukai Hu, Erlu Li, Jiazheng Ding, Yu Han, Wei Ning, Gen Wang
aiXcoder
Beijing, China
{hushukai, lierlu, dingjiazheng, hanyu, ningwei, wanggen}@aixcoder.com

Yihong Dong, Kechi Zhang, Ge Li
Peking University
Beijing, China
dongyh@stu.pku.edu.cn
{zhangkechi, lige}@pku.edu.cn

***Abstract*—Large Language Models (LLMs) have been widely used in code completion, and researchers are focusing on scaling up LLMs to improve their accuracy. However, larger LLMs will increase the response time of code completion and decrease the developers' productivity. In this paper, we propose a lightweight and effective LLM for code completion named aiXcoder-7B. Compared to existing LLMs, aiXcoder-7B achieves higher code completion accuracy while having smaller scales (*i.e.,* 7 billion parameters). We attribute the superiority of aiXcoder-7B to three key factors: ❶ Multi-objective training. We employ three training objectives, one of which is our proposed Structured Fill-In-the-Middle (SFIM). SFIM considers the syntax structures in code and effectively improves the performance of LLMs for code. ❷ Diverse data sampling strategies. They consider inter-file relationships and enhance the capability of LLMs in understanding cross-file contexts. ❸ Extensive high-quality data. We establish a rigorous data collection pipeline and consume a total of 1.2 trillion unique tokens for training aiXcoder-7B. This vast volume of data enables aiXcoder-7B to learn a broad distribution of code. We evaluate aiXcoder-7B in five popular code completion benchmarks and a new benchmark collected by this paper. The results show that aiXcoder-7B outperforms the latest six LLMs with similar sizes and even surpasses four larger LLMs (*e.g.,* StarCoder2-15B and CodeLlama-34B), positioning aiXcoder-7B as a lightweight and effective LLM for academia and industry. Finally, we summarize three valuable insights for helping practitioners train the next generations of LLMs for code. aiXcoder-7B has been open-souced and gained significant attention [1]. As of the submission date, aiXcoder-7B has received 2,193 GitHub Stars.**

***Index Terms*—Code Completion, Large Language Model**

I. Introduction

Large Language Models (LLMs) have been widely used in code completion [2]–[5], *i.e.,* predicting the subsequent code based on the previous context. For example, GitHub Copilot [6], an LLM-based code completion tool, is regularly utilized by developers from over 10,000 organizations. Nowadays, researchers often improve the accuracy of LLMs by scaling up LLMs, *e.g.,* CodeLlama-70B [2]. However, larger LLMs will increase the response time of code completion, which is a critical factor for developer experience and productivity. Thus, it is necessary to train lightweight LLMs that maintain high code completion accuracy while having smaller scales.

Recognizing the above research gap, we present aiXcoder-7B, a lightweight and powerful LLM for code completion. aiXcoder-7B contains 7 billion parameters, ensuring a high inference speed while achieving superior code completion accuracy. In our later experiments, **aiXcoder-7B outperforms the latest LLMs with similar sizes in six code completion benchmarks and even surpasses larger LLMs (*e.g.,* StarCoder2-15B and CodeLlama-34B).** aiXcoder-7B effectively balances model size and performance, providing a better foundational model for both academia and industry.

Compared to previous LLMs, we attribute the superiority of aiXcoder-7B to the following three key factors:

- **Multi-objective training.** Previous LLMs mainly use Next-Token Prediction (NTP) as the training objective, which only covers limited code completion scenarios. To address this limitation, **we propose multi-objective training, including NTP, Fill-In-the-Middle (FIM), and Structured Fill-In-the-Middle (SFIM).** NTP simulates the scenario where developers write a new file from top to bottom, and FIM models the scenario of developers modifying existing code. Because FIM mainly trains models to predict incomplete and irregular code snippets, we further propose SFIM. It parses the code into a syntax tree and mines a relatively complete code span based on a tree node. aiXcoder-7B is trained to predict the code span based on its surrounding context. The three objectives help aiXcoder-7B learn a comprehensive code completion ability across a wider range of code completion scenarios. Details of the multi-objective training are in Section III-C.
- **A diverse data sampling algorithm.** A code repository often contains multiple code files. Previous studies [2], [4], [5] typically randomly sample files for training, failing to leverage the relationships and contextual information between files. **We propose four new sampling strategies: sampling based on file content similarity, sampling based on file**

* Siyuan Jiang and Jia Li contribute equally and are co-first authors.

path similarity, sampling based on inter-file dependency, and random sampling. The first three strategies simulate common cross-file code completion scenarios, such as code completion augmented by similar code and cross-file API completion, helping aiXcoder-7B better understand and utilize dependencies across files. The fourth strategy, random sampling, is to simulate other potential code completion scenarios. Through these diverse sampling strategies, we enhance aiXcoder-7B's understanding capability of cross-file contexts within a repository. Details of our data sampling algorithm are in Section III-B.

- **Extensive high-quality data.** LLMs are inherently data-driven, and their performance is significantly influenced by the quantity and quality of the training data. We establish a rigorous data collection pipeline, including data crawling, data cleaning, deduplication, code quality checks, and sensitive information removal. We leverage this pipeline to collect a substantial amount of high-quality training data. **We continuously feed the training data into aiXcoder-7B, consuming a total of 1.2 trillion unique tokens.** This vast volume of data enables aiXcoder-7B to learn a broad distribution of code data, allowing it to perform exceptionally well across different code completion scenarios. More details of our data collection pipeline are in Section II.

We assess the effectiveness of aiXcoder-7B in three code completion tasks, including Fill-In-the-Middle (FIM), cross-file code completion, and Natural language to Code (NL2Code). We experiment with six code completion benchmarks, five of which are popular public datasets and one is FIM-Eval collected by this paper. FIM-Eval is a benchmark for FIM, consisting of 16,136 samples and covering four languages (*i.e.,* Java, Python, C++, JavaScript). FIM-Eval additionally labels the types of code to be completed, including 13 types, *e.g.,* function signatures and comments. Then, we compare aiXcoder-7B to 10 recently released LLMs (from 7B to 34B) on these benchmarks and yield the following insights: ❶ aiXcoder-7B substantially outperforms LLMs with similar sizes and even surpasses larger LLMs in six benchmarks. For example, in a popular benchmark - HumanEval, aiXcoder-7B achieves a Pass@1 score of 54.9%, outperforming CodeLlama-34B (*i.e.,* 48.2%) and StarCoder2-15B (*i.e.,* 46.3%). The improvements show that aiXcoder-7B achieves higher code completion accuracy while having smaller scales. ❷ Based on our FIM-Eval, we analyze the performance of aiXcoder-7B in completing different types of code. aiXcoder-7B outperforms LLMs with similar sizes on most types (max: 13, min: 8). The results show the strong generalization ability of aiXcoder-7B in code completion. ❸ We show that existing LLMs are prone to generate longer code in FIM, while the code generated by aiXcoder-7B is closer in length to human-written reference code. The result shows that the code generated by aiXcoder-7B is more concise and closer to the human coding style.

Insights of training LLMs for code. Based on our practices in aiXcoder-7B, we summarize three valuable insights, including scaling up training data and introducing the inter-file relationships and code structures into the training. These insights can help practitioners train the next generations of LLMs for code.

We summarize the key contributions of this paper:

- We present aiXcoder-7B, a lightweight and effective LLM with 7 billion parameters for code completion. We have released its weights and code [1]. As of the submission date, aiXcoder-7B has received 2,193 GitHub Stars.
- We propose a novel training objective - Structured Fill-In-the-Middle, which considers the syntax structures in code and effectively improves the performance of LLMs.
- We propose a new data sampling algorithm for code, which considers inter-file relationships and enhances the capability of LLMs in understanding cross-file contexts.
- We release a new code completion benchmark, consisting of 16,136 samples and covering four languages.
- We evaluate the effectiveness of aiXcoder-7B in six code completion benchmarks. aiXcoder-7B substantially outperforms 6 LLMs with similar sizes and even surpasses 4 larger LLMs (15B and 34B).

II. Data Collection Pipeline

This section presents the process of collecting the pre-training data of aiXcoder-7B. Figure 1 shows an overview of our data collection pipeline, consisting of five stages: data crawl (Section II-A), data cleaning (Section II-B), data deduplication (Section II-C), code quality checking (II-D), and sensitive and personally identifiable information removal (Section II-E). Through this pipeline, we collect and clean 2.8TB of natural language data and 3.5TB of source code data. Figure 2 visualizes the distributions of the top 10 programming languages in the code data. Next, we describe the details of the data collection pipeline in the following sections.

A. Data Crawling

The pre-training data of aiXcoder-7B consists of two parts: natural language data and source code data.

Natural Language Data. We collect natural language data from two public datasets: WuDaoCorpora [7] and RefineWeb [8], driven by two key motivations. First, these datasets are highly diverse, covering a wide range of domains and languages. They include a broad spectrum of natural language text from the internet, such as social media conversations, books, and technical papers, and cover two mainstream languages, *i.e.,* English and Chinese. Second, both datasets have been thoroughly cleaned and deduplicated in previous studies, which significantly reduces the preprocessing workload and allows us to focus on processing code data. Finally, we collect 2.8TB of natural language data for pre-training.

Source Code Data. The raw source code data comes from two sources: one is the open-source dataset - The Stack v1.2, and the other is the code data we crawled ourselves.

- *The Stack v1.2* [9] is a comprehensive dataset comprising approximately 6TB of permissively licensed source code sourced from public GitHub repositories, spanning 358 programming languages, with notable representation from

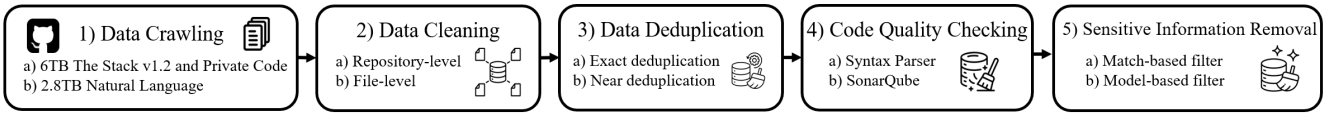

Fig. 1: An overview of our data collection pipeline.

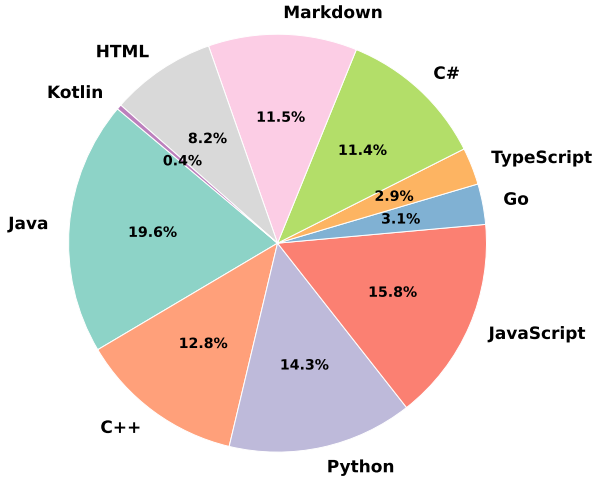

Fig. 2: The distributions of the top 10 programming languages in our source code training data.

HTML, JavaScript, Java, and C. This dataset has undergone rigorous cleaning processes to enhance data integrity and prevent inflated performance metrics due to duplicate content. Only repositories with permissive licenses were retained, while non-essential files, such as binaries and those exceeding 1MB, were systematically excluded. The version 1.2 of The Stack has excluded opt-out requests submitted by February 9, 2023, as well as initially flagged malicious files (this exclusion is not exhaustive).

- *Our crawled data.* Popular repositories we have been crawling from GitHub for the past decade.

B. Data Cleaning

In this stage, we clean the collected data by removing invalid or low-quality data. Because the natural language data has already undergone rigorous cleaning, we focus on cleaning the source code data. Our cleaning process comprises two steps: repository-level cleaning and file-level cleaning. Below, we provide a detailed explanation of each step.

Repository-level Cleaning. Our goal is to remove repositories with impressive licenses and low-quality repositories. To achieve this goal, our cleaning is performed in three steps:

- *Collecting permissive licenses.* We build a list of permissive licenses based on the Blue Oak Council[1] and previous work [9]. This list includes various permissive licenses with minimal restrictions on software copying, modification, and redistribution. Only repositories with licenses from this list are retained for pre-training.
- *Identifying repositories' licenses.* GHArchive provides license information when repository owners explicitly set the code license through the web interface. We first extract each repository's license from GHArchive. If a license is not listed in GHArchive, we leverage the go-license-detector[2] to identify the most likely license.
- *Removing repositories with impressive licenses.* After identifying the licenses, we exclude repositories whose licenses do not appear on the permissive license list.
- *Removing low-quality repositories.* We score the repositories from different aspects, including the number of stars, the number of git commits, and the number of test files. Then, we sort the repositories in descending order based on their scores and remove the lowest 10%.

File-level Cleaning. Next, we filter out low-quality files in repositories. Specifically, we empirically design some rules to filter out low-quality documents: ❶ Trivial files, including empty files, corrupted files, non-text files, and auto-generated files. ❷ Too long files. Too long files typically contain wordy or repetitive content and are not suitable as training data. If a line in a file exceeds 1000 characters, the total number of lines in the file exceeds 10,000, or the size of the file exceeds 1MB, we consider it a long file.

C. Data Deduplication

Previous work [8] has shown that data deduplication can significantly improve the performance of trained models. It is particularly necessary for code data, where code reuse leads to a large amount of duplicate content. Therefore, in this stage, we eliminate duplicate code files within the repositories. Our deduplication process consists of two steps:

- *Exact deduplication.* We extract file contents, find the files with exactly the same content, and keep only one copy.
- *Near deduplication.* Exact deduplication is too strict and may cause false positives. Thus, we further perform near deduplication. We compute the MinHash [10] with 256 permutations of all files and use Locality Sensitive Hashing [11] to find clusters of duplicates. We further reduce the clusters by ensuring that each file in the original cluster is similar to at least one other file in the reduced cluster. We consider two files near-duplicate when their Jaccard similarity exceeds 0.85.

D. Code Quality Checking

In this stage, we use code analysis tools to assess the quality of code data and filter out low-quality code. Low-quality code often contains syntax errors, code defects, vulnerabilities, and

[1]https://blueoakcouncil.org/list

[2]https://github.com/src-d/go-license-detector

misleading models that generate unreliable code. Specifically, we use the following tools to assess the quality of code:

Syntax Parser. Syntax correctness is one of the basic principles that source code should satisfy. We use a public syntax parser - tree-sitter[3] to parse all code files and delete files that fail to parse or time out.

SonarQube. SonarQube[4] is an open-source tool for the inspection of code quality. It can detect code defects, vulnerabilities, code smells, and technical debt in various programming languages. We use SonarQube to identify problematic code files and delete them.

E. Sensitive Information Removal

In this section, we remove the sensitive information in the pre-training data, *e.g.,* texts involving sensitive topics and personally identifiable information (PII). We remove this information in two steps:

Match-based filter. We manually build a list of sensitive words, which covers a broad range of sensitive topics (*e.g.,* politics). Then, we scan all pre-training data and delete the data containing sensitive words.

Model-based filter. Following previous work [4], we use a Named Entity Recognition (NER) model to identify PII in the data. Specifically, we reuse a trained NER model in previous work [4], which can identify six categories of PII, including emails, names, IP addresses, usernames, passwords, and keys. Then, we replace the detected PII entities with the following special tokens: `<EMAIL>`, `<NAME>`, `<IP_ADDRESS>`, `<USERNAME>`, `<PASSWORD>`, `<KEY>`.

III. Model Training

In this section, we describe the pre-training procedure of aiXcoder-7B, including model architecture, data sampling algorithm, and training objectives.

A. Model Architecture

aiXcoder-7B is built upon an auto-regressive dense Transformer architecture [12]. aiXcoder-7B consists of 32 Transformer decoder layers, with a hidden state size of 4096 and an intermediate size of 14464. More details are in our open-sourced repository [1]. Our tokenizer is trained with SentencePiece [13] upon 500GB of training data. The vocabulary size is 49,512. We adopt Rotary Positional Encodings (RoPE) [14] to enhance the representation of positional information in sequences, following [2], [4]. RoPE allows for a more flexible encoding of position than absolute position encoding. Additionally, we implement Grouped Query Attention (GQA) [15], which enhances the efficiency of the attention mechanism by grouping queries, allowing for a more scalable attention computation. We maintain a balanced design in our attention heads, with a configuration of 32 query attention heads and 8 key-value attention heads.

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

[4]https://www.sonarsource.com/products/sonarqube/

B. Data Sampling Algorithm

Through the pipeline in Section II, we collect extensive code repositories and natural language articles. We randomly shuffle these repositories and articles and iterate through them. If a natural language article is sampled, we process it into training sequences based on the training objectives (Section III-C).

If a code repository is sampled, we design an algorithm for sampling files from the repository, as described in Algorithm 1. The algorithm contains four strategies: sampling based on file content similarity, sampling based on file path similarity, sampling based on inter-file dependencies, and random sampling. The first three strategies simulate common cross-file code completion scenarios, such as code completion augmented by similar code and cross-file API completion, helping aiXcoder-7B better understand and utilize dependencies across files. The fourth strategy, random sampling, is to simulate other potential code completion scenarios. For each repository, the probability of selecting each of the first three strategies is 30%, and the probability of selecting the last strategy is 10%. These sampled files are further converted into training sequences based on the training objectives (Section III-C).

C. Training Objectives

The training objectives of aiXcoder-7B consist of the Next-Token Prediction (NTP) and Structured Fill-In-the-Middle (SFIM), detailed as follows.

Next-Token Prediction (NTP). It is similar to code completion, training models to predict the subsequent token based on the provided context. Given a code file or natural language article $\mathbf{x} = \{x_0, x_1, \ldots, x_l\}$, NTP trains models predict the next token x_i based on previous tokens $\{x_{<i}\}$. The objective is to minimize the following loss function:

$$\text{loss}_{NTP} = -\sum_{i=0}^{l-1} \log p\left(x_i \mid x_{t<i}\right) \tag{1}$$

Fill-In-the-Middle (FIM) [16]. The motivation behind this training objective is that human developers frequently modify existing code, *e.g.,* inserting new code snippets. Thus, FIM trains models to predict the middle content based on the preceding and following context. Specifically, given a code file or natural language article $\mathbf{x} = \{x_0, \ldots, x_l\}$, we randomly select a span of contiguous tokens as the $\mathbf{middle} = \{x_i, \ldots, x_j\}$, using the content above the span as the $\mathbf{prefix} = \{x_0, \ldots, x_{i-1}\}$ and the content below as the $\mathbf{suffix} = \{x_{j+1}, \ldots, x_l\}$. We employ two distinct modes to construct the training sequence: PSM (*i.e.,* $[\mathbf{prefix}; \mathbf{suffix}; \mathbf{middle}]$) or SPM (*i.e.,* $[\mathbf{suffix}; \mathbf{prefix}; \mathbf{middle}]$). $[;]$ means the concatenation of multiple strings using special tokens. Previous work [3] has found that models work best when PSM and SPM account for 50% each. Thus, we choose the probability of each mode being 50%.

Finally, we feed the training sequence into aiXcoder-7B and minimize the following loss function:

$$\begin{aligned} \text{loss}_{FIM} = &- \log p\left([\mathbf{prefix}; \mathbf{suffix}; \mathbf{middle}]\right) \\ &- \log p\left([\mathbf{suffix}; \mathbf{prefix}; \mathbf{middle}]\right) \end{aligned} \tag{2}$$

Algorithm 1 Sampling code files within a repository.

Inputs:
A list of code files $Files$
Outputs:
An ordered list of code files $orderedFiles$

$orderedFiles \leftarrow []$
$randomValue \leftarrow random(0, 1)$
if $randomValue < 0.3$ **then**
  // Sampling based on file content similarity
  $tfIdfList \leftarrow []$
  for $file$ in $Files$ **do**
    $tfIdfList.append(\text{TFIDF}(file))$
  end for
  $clusterNum \leftarrow min((random(1, 20), Files.size))$
  $Clusters \leftarrow \text{KMEANS}(clusterNum, tfIdfList)$
  for $Cluster$ in $\text{SHUFFLE}(Clusters)$ **do**
    $orderedFiles.extend(\text{SHUFFLE}(Cluster))$
  end for
else if $randomValue < 0.6$ **then**
  // Sampling based on file path similarity
  while $Files.size > 0$ **do**
    $K \leftarrow random(1, Files.size)$
    $randomFile \leftarrow random(Files)$
    for $File$ in $\text{KNN\_PATH}(randomFile, K)$ **do**
      $orderedFiles.append(File)$
      $Files.remove(File)$
    end for
    $orderedFiles.append(randomFile)$
  end while
else if $randomValue < 0.9$ **then**
  // Sampling based on file dependencies
  $callGraph \leftarrow \text{CALL\_GRAPH}(Files)$
  $leafNodes \leftarrow \text{GET\_LEAFS}(callGraph)$
  while $leafNodes.isNotEmpty()$ **do**
    for $node$ in $leafNodes$ **do**
      $lastPredecessors \leftarrow \{node\}$
      while $lastPredecessors.isNotEmpty()$ **do**
        for $node$ in $\text{SHUFFLE}(lastPredecessors)$ **do**
          for p in $node.predecessors$ **do**
            $p.successors.remove(node)$
            if $p.successors.isEmpty()$ **then**
              $leafNodes.add(p)$
            end if
          end for
          $orderedFiles.append(node.file)$
        end for
        $lastPredecessors \leftarrow node.predecessors$
      end while
    end for
  end while
else
  // Random sampling
  $orderedFiles.extend(\text{SHUFFLE}(Files))$
end if
return $orderedFiles$

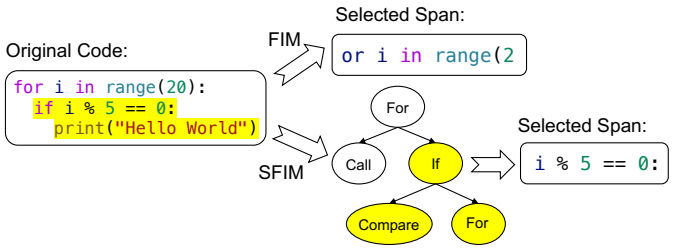

Fig. 3: Examples of selected spans in FIM and SFIM.

Structured Fill-In-the-Middle (SFIM). As shown in Figure 3, FIM randomly selects spans and trains models to predict incomplete and irregular code snippets (*e.g.,* `or i in range(2)`. However, developers often expect models to complete the current code into a complete snippet, such as a completed code line or loop block, instead of suggesting an incomplete code snippet. To address this, we propose SFIM, which trains aiXcoder-7B to predict complete code snippets, enabling aiXcoder-7B to align with the practical needs of developers. Given a code file, SFIM uses a new strategy for selecting spans: ❶ randomly select a function from the file and parse the function into a syntax tree; ❷ randomly choose a non-root, non-leaf node from the tree and locate the code snippet corresponding to this node; ❸ within this code snippet, randomly select a span, where the start position is randomly determined, but the end position must be the end of a code line. As shown in Figure 3, SFIM selects an If node and mines a relatively complete code snippet (*i.e.,* `i%5==0:`) as the span.

Subsequently, we follow the FIM and convert the select span into a training sequence in the format of PSM or SPM. Based on preliminary experiments, we set the probability of selecting PSM to 30% and SPM to 70%. We input the training sequence into aiXcoder-7B and minimize the following loss function:

$$\begin{aligned} \text{loss}_{SFIM} = &- \log p\left([\textbf{prefix}; \textbf{suffix}; \textbf{middle}]\right) \\ &- \log p\left([\textbf{suffix}; \textbf{prefix}; \textbf{middle}]\right) \end{aligned} \tag{3}$$

Multi-objective Training. We optimize the above three objectives alternately. Given a code repository, we choose SFIM with a probability of 70% and FIM and NTP with a probability of 15%, respectively. Given a natural language article, we choose FIM and NTP with a probability of 50%, respectively. We determine these probabilities based on previous work [3]–[5] and our preliminary experiments.

D. Training Details

We leverage Megatron to train aiXcoder-7B. The training process is conducted on 128 A100 40GB GPUs, consuming a total of 1.2 trillion unique tokens. The hyper-parameters used during training are shown in Table I.

IV. Study Design

We design a large-scale study to evaluate the effectiveness of aiXcoder-7B. This section presents the details of our study, including research questions, benchmarks, compared baselines, and evaluation metrics.

TABLE I: Training Hyperparameters for aiXcoder-7B

Hyperparameter	aiXcoder-7B
lr-decay-iters	320000
weight-decay	0.01
lr-decay-style	cosine
clip-grad	1.0
hidden-dropout	0.1
attention-dropout	0.05
adam-beta1, beat2	0.9, 0.98
Batch Size	512
Max Learning Rate	1e-5
Context window	32768

A. Research Questions

Our study aims to answer the following Research Questions (RQs). They evaluate aiXcoder-7B in three code completion tasks, including Natural Language to Code (NL2Code), Fill-In-the-Middle (FIM), and cross-file code completion.

RQ1: How does aiXcoder-7B perform on NL2Code task compared to existing LLMs? NL2Code is the task of completing the source code based on a natural language description or function signature.

RQ2: How does aiXcoder-7B perform on Fill-In-the-Middle task compared to existing LLMs? FIM simulate scenarios where developers modify existing code by predicting the missing middle portion using bidirectional contexts.

RQ3: How does aiXcoder-7B perform on Cross-File Code Completion compared to existing LLMs? This task requires completing code by using relevant context from other files within the current repository.

In the RQs, we apply aiXcoder-7B on 6 benchmarks totally. These benchmarks cover 6 programming languages. To show the superiority of aiXcoder-7B, we also select 10 popular LLMs as baselines for comparison. Then, we report the execution-based and text-based metrics (Section IV-D) of completed programs.

B. Compared Models

Note that our aiXcoder-7B was open-sourced in March 2024. Thus, we select LLMs released before March 2024 for comparison. Specifically, we select 10 popular LLMs for comparison, and they can be divided into two groups:

❶ **LLMs with similar sizes.** The first group contains six popular LLMs, which have similar sizes to aiXcoder-7B.

- **CodeGen2.5-7B** [17], released by Salesforce, is a 7B parameter model specialized in code generation and understanding, trained on a diverse set of programming languages.
- **CodeGeex2-7B** [18], developed by Zhipu AI, is a 7B parameter model designed for code completion and bug fixing, leveraging a large corpus of code data.
- **CodeLlama-7B** [2], an open-source model by Meta AI, is a 7B parameter architecture fine-tuned on a vast collection of code and natural language data based on Llama2 [19].
- **CodeShell-7B** [20], introduced by Shell AI, is a 7B parameter model focused on shell scripting and code interpretation, trained on a mixture of code and command-line data.
- **StarCoder2-7B** [4], from BigCode, is a 7B parameter model trained on The Stack v2 dataset, specializing in code understanding and generation across multiple programming languages.
- **DeepSeekCoder-7B** [3], by DeepSeek AI, is a 7B parameter model trained on a blend of code and natural language data, designed for programming tasks.

❷ **LLMs with larger sizes.** We also select four larger LLMs as baselines to demonstrate the superiority of aiXcoder-7B:

- **CodeLlama-13B** [2] is an enhanced version of the CodeLlama model with 13B parameters.
- **StarCoder-15B** [4] is the expanded version of the StarCoder model with 15B parameters, delivering improved accuracy for code synthesis and interpretation.
- **StarCoder2-15B** [5] is a 15B parameter model upgraded on the original StarCoder, offering refined code generation and more diverse programming languages.
- **CodeLlama-34B** [2] is the largest variant of the CodeLlama series with 34B parameters.

C. Benchmarks

NL2Code Benchmarks. Following previous studies [2]–[4], we select three popular NL2Code benchmarks in our experiments, detailed as follows.

- **HumanEval [21] and MBPP [22]** consist of 164 and 974 Python programming problems. Each problem includes a function signature, a detailed docstring, and several test cases. LLMs are required to complete the function body based on the signature and docstring. The generated code is checked by executing test cases, being considered correct only if all tests pass.
- **MultiPL-E [23]** is the multilingual version of HumanEval, covering multiple programming languages, *e.g.,* C++, Java, and JavaScript.

FIM Benchmarks. Code is rarely composed in a straightforward left-to-right sequence. Simulating when a developer modifies existing code, FIM refers to the task of completing missing a middle code snippet leveraging bidirectional contexts.

- **Santacoder-data [24]** is a popular FIM benchmarks consisting of 4,792 samples. It is built from MultiPL-E [23] and requires LLMs to predict a single line of code based on the preceding and following context.
- **FIM-Eval** is a large-scale FIM benchmark collected by this paper. We construct FIM-Eval from some real-world repositories, which are excluded from the training data of aiXcoder-7B. We extract 13 types of code snippets from these repositories and randomly mine spans from these code snippets. These 13 types of code snippets encompass common code completion scenarios, including method signatures, method bodies, single-line statements, methods with comments, empty code blocks, specific positions within a

TABLE II: The Pass@1 of LLMs on NL2Code benchmarks.

Model	HumanEval	MBPP	MultiPL-E (C++)	MultiPL-E (Java)	MultiPL-E (JS)	Average
CodeGen2.5-7B	28.7%	39.2%	25.7%	26.1%	26.2%	29.1%
CodeGeex2-7B	36.0%	36.2%	29.2%	25.9%	24.8%	30.4%
CodeLlama-7B	31.7%	38.6%	29.8%	34.2%	29.2%	32.7%
CodeShell-7B	34.4%	38.6%	28.2%	30.4%	33.2%	32.9%
StarCoder2-7B	35.4%	54.4%	33.6%	29.4%	35.4%	37.6%
DeepSeekCoder-7B	49.4%	60.6%	50.3%	43.0%	48.4%	50.3%
aiXcoder-7B	**54.9%**	**66.0%**	**58.2%**	**57.0%**	**64.5%**	**60.1%**
StarCoder-15B	31.7%	42.8%	31.1%	28.5%	29.8%	32.8%
CodeLlama-13B	36.0%	48.4%	37.9%	38.0%	32.3%	38.5%
StarCoder2-15B	46.3%	66.2%	41.4%	33.9%	44.2%	46.4%
CodeLlama-34B	48.2%	55.2%	44.7%	44.9%	42.2%	47.0%

method body (top, middle, and bottom), specific control statements (*i.e.,* if statements, for loops, while loops, try statements, and switch-case statements). Finally, we collect 16,140 samples covering four programming languages: C++ (4,080 samples), Java (4,080 samples), Python (3,900 samples), and JavaScript (4,080 samples). FIM-Eval provides a reliable, practical, and diverse evaluation platform for FIM. FIM-Eval has been open-sourced in our repository [1].

Cross-File Code Completion Benchmarks. This task requires LLMs to complete the code based on cross-file context within the same project. Building upon insights from prior research [3], [4], detailed as follows.

- **CrossCodeEval [25]** covers four popular programming languages: 2,665 Python samples, 2,139 Java samples, 3,356 TypeScript samples, and 1,768 C# samples. Each sample is provided in three formats: no cross-file context, retrieved cross-file context, and retrieval with reference. The LLMs completed code snippet is compared using text-based metrics.

D. Evaluation Metrics

We describe the evaluation metrics used in different code completion tasks.

NL2Code. NL2Code benchmarks provide test cases for evaluation. Thus, we execute test cases to check the correctness of the generated code and report Pass@k [21]. Specifically, we generate $n \geq k$ code snippets per testing sample, count the number of correct code snippets $c \leq n$ that pass all test cases, and calculate the Pass@k:

$$\text{Pass@}k := \mathop{\mathbb{E}}_{\text{Samples}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \tag{4}$$

FIM and cross-file code completion. We consider the LLMs' completions as predictions and the human-written completions as references. We compare the predictions to references and compute the following metric:

- **BLEU [26]** measures the n-gram similarity between predictions and references. n is empirically set to 4.
- **CodeBLEU [27]** is a variant of BLEU for code. It considers not only the n-gram similarity but also the syntax and data flow similarity.
- **Exact Match (EM)** evaluates the percentage of cases where the prediction exactly matches the reference, providing a strict measure of how often LLMs produce correct code without deviations.
- **Edit Similarity (ES)** measures the similarity between the prediction and the reference based on the number of edits required to transform one into the other, typically using metrics like Levenshtein distance [28].

V. Results and Analyses

TABLE III: The exact match of LLMs on the SantaCoder-data benchmark.

Model	Python	JavaScript	Java	Avg
StarCoder2-7B	61.1%	77.5%	81.1%	73.2%
CodeLlama-7B	67.6%	74.3%	80.2%	74.0%
CodeLlama-13B	68.3%	77.6%	80.7%	75.5%
DeepSeekCoder-7B	66.6%	79.7%	**88.1**%	78.1%
aiXcoder-7B	**73.3**%	**81.7**%	83.0%	**79.3**%

A. RQ1: Performance on NL2Code

Following recent work on LLMs [3], [5], we use greedy decoding and report Pass@1. Table II shows the results of different LLMs on NL2Code benchmarks. From Table II, we draw the following observations:

- Compared to LLMs of similar sizes, our aiXcoder-7B achieves the current best results, outperforming the top-performing model DeepSeekCoder-7B by an average of 9.8%. Moreover, it significantly surpasses CodeGen2.5-7B with a 31% absolute advantage.
- aiXcoder-7B even surpasses four larger LLMs (*e.g.,* StarCoder2-15B and CodeLlama-34B), achieving a lead of 13.1% over CodeLlama-34B, which is nearly five times larger, and 13.7% over StarCoder2-15B on average.
- Across languages like Java, Python, C++, and JavaScript, our aiXcoder-7B shows strong performance. It surpasses DeepSeekCoder-7B by 16.1% in JavaScript and exceeds by 5.5% in Python.

B. RQ2: Performance on Fill-In-the-Middle (FIM)

Generally, FIM closely mirrors how developers modify existing code, making it an ideal method for evaluating models in real-world programming scenarios.

Based on the experimental findings outlined in Table III, aiXcoder-7B demonstrates the highest overall performance

TABLE IV: Performance of LLMs on CrossCodeEval.

Model	Python		Java		TypeScript		C#		Average	
	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES
Base Model										
CodeLlama-7B	22.3	55.2	27.9	66.9	10.8	70.9	45.8	77.2	26.7	67.6
StarCoder2-7B	22.5	57.3	25.9	65.9	28.9	71.6	39.5	70.5	29.2	66.3
DeepSeekCoder-7B	27.2	62.3	33.4	73.2	36.6	77.3	45.9	77.0	35.8	72.4
aiXcoder-7B	**30.0**	**70.8**	**34.9**	**77.8**	**35.3**	**79.6**	**49.9**	**86.4**	**37.5**	**78.7**
+ Retrieval BM25										
CodeLlama-7B	23.5	53.5	33.9	68.4	11.5	71.5	50.6	75.3	29.9	67.2
StarCoder2-7B	25.3	58.0	31.4	67.4	33.3	73.2	43.5	69.8	33.4	67.1
DeepSeekCoder-7B	29.9	62.9	39.8	74.8	39.0	77.0	52.2	78.1	40.2	73.2
aiXcoder-7B	**35.3**	**74.3**	**42.2**	**80.4**	**39.9**	**81.3**	**57.7**	**88.8**	**43.8**	**81.2**
+ Retrieval w/ Ref.										
CodeLlama-7B	26.7	54.9	36.3	69.0	12.8	72.9	52.8	75.0	32.1	67.9
StarCoder2-7B	28.5	59.0	35.0	69.2	36.0	72.6	47.9	71.6	36.9	68.1
DeepSeekCoder-7B	33.2	64.5	43.7	76.1	43.4	78.4	55.4	78.7	43.9	74.4
aiXcoder-7B	**40.4**	**76.3**	**47.0**	**82.4**	**45.0**	**83.8**	**61.0**	**89.4**	**48.4**	**83.0**

TABLE V: The performance of LLMs on FIM-Eval.

Model	Java			
	EM(%)	BLEU-4	CodeBLEU	Average
CodeLLama-7B	38.1	56.9	69.9	55.0
StarCoder2-7B	37.7	57.7	69.2	54.9
DeepSeekCoder-7B	43.4	63.4	71.7	59.5
aiXcoder-7B	**49.4**	**70.6**	**74.0**	**64.7**
Model	**C++**			
	EM(%)	**BLEU-4**	**CodeBLEU**	**Average**
CodeLLama-7B	25.4	44.2	63.9	44.5
StarCoder2-7B	22.6	41.9	61.2	41.9
DeepSeekCoder-7B	29.1	50.0	65.8	48.3
aiXcoder-7B	**37.3**	**60.3**	**67.4**	**55.0**
Model	**JavaScript**			
	EM(%)	**BLEU-4**	**CodeBLEU**	**Average**
CodeLLama-7B	25.7	44.1	60.4	43.4
StarCoder2-7B	23.9	42.0	57.4	41.1
DeepSeekCoder-7B	29.3	49.0	60.3	46.2
aiXcoder-7B	**36.5**	**58.0**	**64.3**	**52.9**
Model	**Python**			
	EM(%)	**BLEU-4**	**CodeBLEU**	**Average**
CodeLLama-7B	21.6	39.9	60.0	40.5
StarCoder2-7B	24.4	43.5	59.5	42.5
DeepSeekCoder-7B	29.6	51.1	**63.4**	48.0
aiXcoder-7B	**35.0**	**56.1**	63.0	**51.4**

on **SantaCoder-data**, achieving the best results in Python, JavaScript, and Java among the models tested.

Table V shows the average generation performance on **FIM-Eval**. Figure 4 shows the performance of LLMs in predicting different types of code. Based on the results, we obtain the following observations:

- In real-world programming, aiXcoder-7B performs well in FIM. When evaluated on Java, C++, and JavaScript in FIM-Eval, aiXcoder-7B surpasses DeepSeekCoder-7B by an average of 5.2, 6.7, and 6.4 in FIM metrics for these three languages, highlighting its multilingual versatility. It is highest in C++, exceeding StarCoder2-7B by 11.8.
- aiXcoder-7B offers no clear edge over DeepSeekCoder-7B in Python, likely due to lower training data proportion. When calculating CodeBLEU in FIM-Eval, aiXcoder-7B's score of 63.0 is slightly lower than DeepSeekCoder-7B's score of 63.4. In several aspects, such as method body top/mid and if statement, it falls behind by up to 10%, indicating the need for a better understanding of method initiations and conditional branches. Moreover, in the SantaCoder-data benchmark, aiXcoder-7B's 83.0% EM of Java is 5.1% lower than the best score of 88.1%. This will be rectified by boosting Python and Java data in training.

C. RQ3: Performance on Cross-File Code Completion

Another important capability of LLMs is the ability to understand code context across files, as developers often need to consider information from other files within the current project when writing code. In Table IV, we fix the context length for all LLMs at 16K and format the input using the PSM pattern in FIM. All LLMs employ the greedy search to generate code.

We design three experimental settings: ❶ **Base Setting.** As a baseline, LLMs complete based solely on the current file without cross-fire context. ❷ **Retrieval BM25.** Based on the current file context in the base settings, It additionally uses BM25 to match repository code fragments. The top 5 matches, capped at 512 tokens, are added to the prompt, along with formatted class definitions from other files. ❸ **Retrieval w/Ref.** In this setting, we make use of not only the in-file context (as in Retrieval BM25 setting) but also the reference to retrieve the cross-file context. We prepend the retrieved context to the in-file context to construct the prompt for this setting.

The subsequent conclusions can be made:

- Under three experimental settings, aiXcoder-7B performs very well, achieving EM of 30.0 and ES of 70.8 in Python, outperforming CodeLlama-7B by 7.7 and 21.4 in the base setting. In the other two experimental settings with retrieval, aiXcoder-7B has an average EM that is higher than

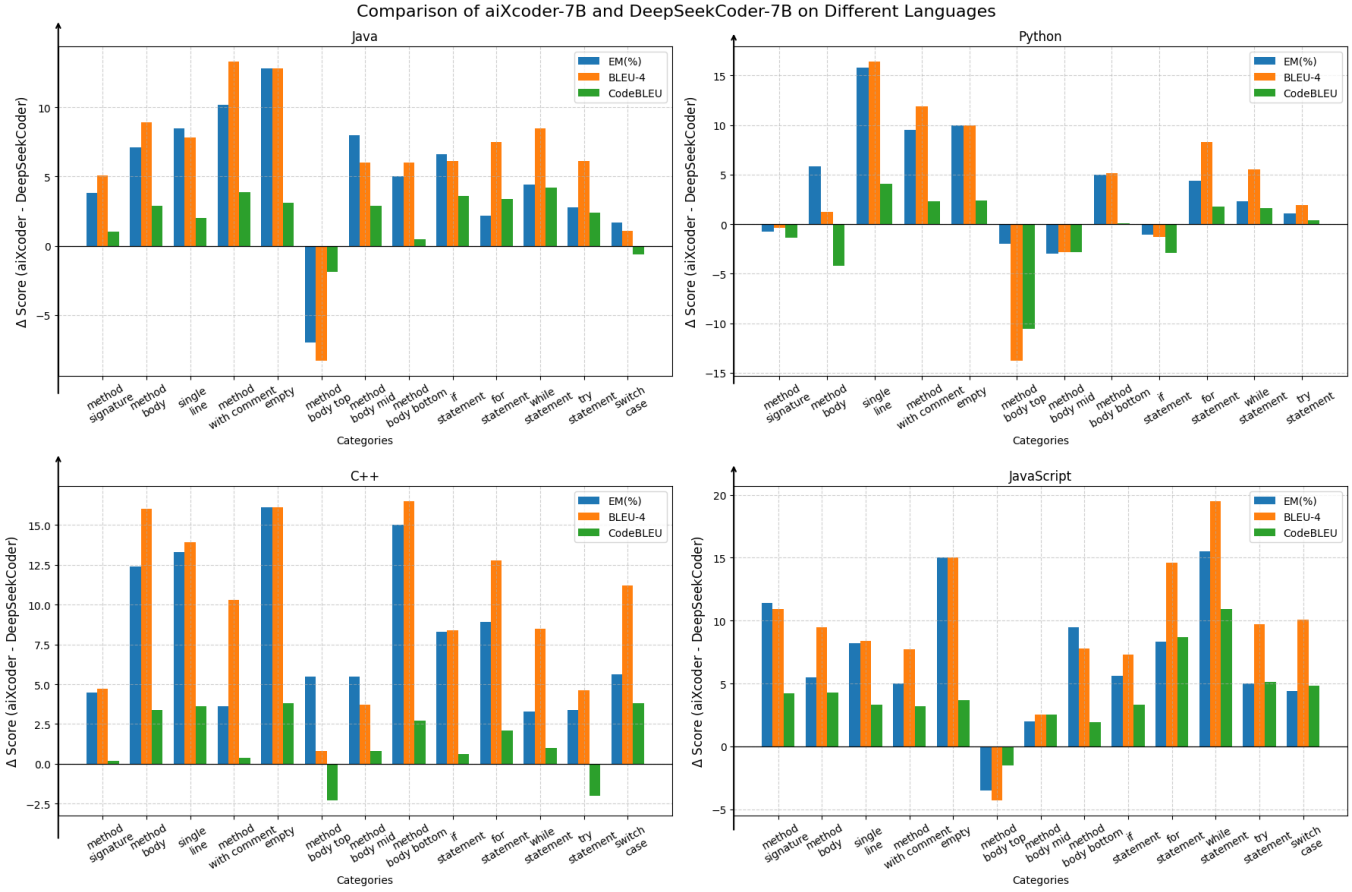

Fig. 4: Performance of LLMs on different types of code in FIM-Eval.

DeepSeekCoder-7B by 3.6 and 4.5 and an average ES that is higher by 4.5 and 8.6.

- The model's performance varies across different languages. aiXcoder-7B excels in C# and achieves an EM of 61 in the third setting. In other languages, the improvements of aiXcoder-7B slightly decrease. For example, it only achieves an EM of 40.4 in Python. In the future, we will continue to enhance the model's performance across different languages.

VI. Discussion

A. Comparison in the Length of Code

We propose a novel evaluation perspective in FIM, *i.e.,* comparing the code length between human-written reference code and code generated by LLMs. It is essential not only to ensure that the completed code is functionally correct but also that its length is consistent with what a human programmer would produce.

To gain insights into this aspect, we evaluate LLMs' performance using FIM-Eval (Section IV-C), which includes a variety of scenarios. Additionally, we present the code length ratio, which is calculated as the ratio of the number of tokens in the prediction to the number of tokens in the ground truth code. Based on the experimental results in Table VI below, we observe that existing LLMs tend to over-generate, producing code that is substantially longer than necessary. Too long code will increase the burden on users and reduce maintainability. For example, CodeLlama-7B produces ratios of 2.14 for Java and 3.02 for C++, while StarCoder2-7B reaches even higher ratios, such as 3.62 for C++. In contrast, aiXcoder-7B consistently generates code predictions that are similar in length to human-written code, achieving ratios of 0.97 for Java and 0.87 for Python. We attribute this performance to aiXcoder-7B's Structured Fill-In-the-Middle (SFIM) training objective, which helps align model outputs with human-written code, resulting in more efficient coding practices.

TABLE VI: The length comparison between the generated code and reference code

Model	Java	C++	JavaScript	Python
CodeLlama-7B	2.14(340/159)	3.02(486/161)	2.39(407/170)	3.28(547/167)
StarCoder2-7B	2.22(353/159)	3.62(583/161)	2.69(458/170)	2.92(488/167)
DeepSeekCoder-7B	1.37(217/159)	2.05(330/161)	1.37(232/170)	1.65(275/167)
aiXcoder-7B	0.97(154/159)	1.35(217/161)	1.04(177/170)	0.87(146/167)

B. Insights of Training LLMs for Code

Based on our practices in aiXcoder-7B, we summarize the following insights to help practitioners train the next generations of LLMs for code.

Scaling up training data can continuously improve the performance of LLMs. Although the scaling law [29] provides a relationship between model size and the amount of training data, we discover that further scaling up training data is necessary. Even if the training loss of LLMs is already

small, continually training with new data still can improve the performance of models. Similar phenomena have also been found in other works [30].

Exploiting the relationships between code files during training can enhance the LLMs' ability to understand cross-file context. In practical applications, LLMs often need to predict code based on cross-file context [25]. Thus, during the training process, we should organize files based on their relationships and train LLMs to understand and utilize the cross-file context. For example, we sample files based on the similarity of their content, which is closer to retrieval-augmented code completion scenarios.

Incorporating the code structures into the training objectives can improve the performance of LLMs. The code is highly structured. However, previous LLMs [2]–[4] view the code into plain text, ignoring the underlying code structures. This paper first incorporates code structures into the training objectives of LLMs and proposes SFIM. SFIM constructs code spans based on syntax trees of code and trains LLMs to generate accurate and concise code. The results in Section V show the effectiveness of our SFIM. This inspires practitioners to explore new training objectives to model valuable code structures, *e.g.,* control flows and data flows.

C. Threats to Validity

We summarize two main threats to this paper.

Data Leakage. A critical threat when training LLMs is the potential inclusion of evaluation data within the training set, which can undermine the reliability of evaluation outcomes. To address this threat, we exclude any data related to our evaluation datasets during data collection. Additionally, the FIM-Eval dataset we constructed and used in our experiments was further emphasized to ensure its independence from the training data. While we cannot guarantee the absence of data leakage in other models due to lack of access, our benchmarks demonstrate that aiXcoder-7B outperforms them reliably.

The selection of hyper-parameters. Another threat to the validity of our study lies in the selection of hyperparameters and rules used during the training of aiXcoder-7B, including model architecture hyperparameters, thresholds for data cleaning, data deduplication parameters, code quality assessment criteria, and sensitive information removal strategies. We selected these based on our preliminary experiments and prior empirical knowledge. We did not conduct a comprehensive hyperparameter search due to the substantial computational costs involved in pre-training, potentially resulting in suboptimal configurations. However, this does not affect our contributions, as future improvements in hyperparameter optimization or heuristic rules can be easily integrated into our framework.

VII. Related Work

This section provides an overview of the evolution of LLMs for code. We categorize LLMs into closed-source and open-source models.

Closed-Source LLMs for Code. One of the earliest notable breakthroughs is Codex [21], introduced by OpenAI. Codex is fine-tuned from GPT-3 using a high-quality code dataset and demonstrates strong performance in Python code completion. Following Codex, OpenAI continued to lead with the development of GPT-4, GPT-4 Turbo, and GPT-4o, all of which exhibit strong capabilities in both code generation and code completion [31]. Alongside OpenAI, Google introduced Gemini [32], and Anthropic contributed with Claude3 [33], all of which integrate code-related tasks into broader conversational capabilities. These closed-source models dominate the landscape with outstanding performance in code-related tasks due to their extensive training datasets and advanced optimization techniques.

Open-Source LLMs for Code. Parallel to the development of closed-source LLMs, open-source LLMs have significantly expanded access to code-related technologies, thereby fostering exploration and innovation in the research community. Meta AI's CodeLlama [2], built upon Llama2 [19], showcased advanced capabilities, including fill-in-the-blank and zero-shot programming. DeepSeek AI's DeepSeek Coder [3], trained on a diverse dataset of 2 trillion tokens, offered multiple model sizes to meet various needs. The BigCode community released StarCoder [4], trained on The Stack v2 dataset [9], which outperformed other open-source models in Python and other programming languages at the time. These models, along with newer iterations like DeepSeek Coder V2 [34], have effectively reduced the performance gap with closed-source models while promoting transparency, reproducibility, and community-driven development.

Our aiXcoder-7B is part of the open-source community's ongoing efforts to advance code completion. It outperforms existing LLMs with similar sizes in six code completion benchmarks, serving as a lightweight and effective function model for academia and industry.

VIII. Conclusion and Future Work

Conclusion. This paper presents aiXcoder-7B, a lightweight and effective LLM for code completion. aiXcoder-7B is trained with 1.2 trillion unique tokens and employs some novel training techniques, including diverse data sampling strategies and multi-objective training. We conduct extensive experiments on six code completion benchmarks covering six programming languages. The results show that aiXcoder-7B outperforms the latest six LLMs with similar sizes and even surpasses four larger LLMs (*e.g.,* CodeLLaMa-34B). We also provide some valuable insights for helping practitioners train the next generations of LLMs for code.

Future Work. In the future, we will train more powerful lightweight LLMs for code completion. Specifically, we plan to design a model architecture dedicated to code. Compared with Transformer, it can explicitly model code structures, *e.g.,* syntax structures. In addition, we will soon release instruction fine-tuned versions of aiXcoder-7B to support more software engineering tasks, *e.g.,* code summarization and code repair.

References

[1] aiXcoder, "aixcoder-7b," https://github.com/aixcoder-plugin/aiXcoder-7B, 2024.

[2] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. Canton-Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," *CoRR*, vol. abs/2308.12950, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2308.12950

[3] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, "Deepseek-coder: When the large language model meets programming - the rise of code intelligence," *CoRR*, vol. abs/2401.14196, 2024.

[4] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. M. V, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Moustafa-Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder: may the source be with you!" *CoRR*, vol. abs/2305.06161, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.06161

[5] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, T. Liu, M. Tian, D. Kocetkov, A. Zucker, Y. Belkada, Z. Wang, Q. Liu, D. Abulkhanov, I. Paul, Z. Li, W. Li, M. Risdal, J. Li, J. Zhu, T. Y. Zhuo, E. Zheltonozhskii, N. O. O. Dade, W. Yu, L. Krauß, N. Jain, Y. Su, X. He, M. Dey, E. Abati, Y. Chai, N. Muennighoff, X. Tang, M. Oblokulov, C. Akiki, M. Marone, C. Mou, M. Mishra, A. Gu, B. Hui, T. Dao, A. Zebaze, O. Dehaene, N. Patry, C. Xu, J. J. McAuley, H. Hu, T. Scholak, S. Paquet, J. Robinson, C. J. Anderson, N. Chapados, and et al., "Starcoder 2 and the stack v2: The next generation," *CoRR*, vol. abs/2402.19173, 2024.

[6] GitHub, "Github copilot," https://github.com/features/copilot, 2023.

[7] BAAI, "Wudaocorporatext," https://data.baai.ac.cn/details/WuDaoCorporaText, 2023.

[8] G. Penedo, Q. Malartic, D. Hesslow, R. Cojocaru, H. Alobeidli, A. Cappelli, B. Pannier, E. Almazrouei, and J. Launay, "The refinedweb dataset for falcon LLM: outperforming curated corpora with web data only," in *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., 2023.

[9] D. Kocetkov, R. Li, L. B. Allal, J. Li, C. Mou, Y. Jernite, M. Mitchell, C. M. Ferrandis, S. Hughes, T. Wolf, D. Bahdanau, L. von Werra, and H. de Vries, "The stack: 3 TB of permissively licensed source code," *Trans. Mach. Learn. Res.*, vol. 2023, 2023.

[10] A. Z. Broder, "Identifying and filtering near-duplicate documents," in *Combinatorial Pattern Matching, 11th Annual Symposium, CPM 2000, Montreal, Canada, June 21-23, 2000, Proceedings*, ser. Lecture Notes in Computer Science, R. Giancarlo and D. Sankoff, Eds., vol. 1848. Springer, 2000, pp. 1–10. [Online]. Available: https://doi.org/10.1007/3-540-45123-4_1

[11] S. Har-Peled, P. Indyk, and R. Motwani, "Approximate nearest neighbor: Towards removing the curse of dimensionality," *Theory Comput.*, vol. 8, no. 1, pp. 321–350, 2012. [Online]. Available: https://doi.org/10.4086/toc.2012.v008a014

[12] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *NIPS*, 2017, pp. 5998–6008.

[13] T. Kudo and J. Richardson, "Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing," in *EMNLP (Demonstration)*. Association for Computational Linguistics, 2018, pp. 66–71.

[14] J. Su, M. H. M. Ahmed, Y. Lu, S. Pan, W. Bo, and Y. Liu, "Roformer: Enhanced transformer with rotary position embedding," *Neurocomputing*, vol. 568, p. 127063, 2024.

[15] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai, "GQA: training generalized multi-query transformer models from multi-head checkpoints," in *EMNLP*. Association for Computational Linguistics, 2023, pp. 4895–4901.

[16] M. Bavarian, H. Jun, N. Tezak, J. Schulman, C. McLeavey, J. Tworek, and M. Chen, "Efficient training of language models to fill in the middle," *CoRR*, vol. abs/2207.14255, 2022.

[17] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, "Codegen2: Lessons for training llms on programming and natural languages," *CoRR*, vol. abs/2305.02309, 2023.

[18] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, Z. Wang, L. Shen, A. Wang, Y. Li, T. Su, Z. Yang, and J. Tang, "Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023, pp. 5673–5684.

[19] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. Canton-Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," *CoRR*, vol. abs/2307.09288, 2023.

[20] R. Xie, Z. Zeng, Z. Yu, C. Gao, S. Zhang, and W. Ye, "Codeshell technical report," *CoRR*, vol. abs/2403.15747, 2024.

[21] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021.

[22] J. Austin, A. Odena, M. I. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. J. Cai, M. Terry, Q. V. Le, and C. Sutton, "Program synthesis with large language models," *CoRR*, vol. abs/2108.07732, 2021.

[23] F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman, A. Guha, M. Greenberg, and A. Jangda, "A scalable and extensible approach to benchmarking nl2code for 18 programming languages," *CoRR*, vol. abs/2208.08227, 2022.

[24] L. B. Allal, R. Li, D. Kocetkov, C. Mou, C. Akiki, C. M. Ferrandis, N. Muennighoff, M. Mishra, A. Gu, M. Dey, L. K. Umapathi, C. J. Anderson, Y. Zi, J. Lamy-Poirier, H. Schoelkopf, S. Troshin, D. Abulkhanov, M. Romero, M. Lappert, F. D. Toni, B. G. del Río, Q. Liu, S. Bose, U. Bhattacharyya, T. Y. Zhuo, I. Yu, P. Villegas, M. Zocca, S. Mangrulkar, D. Lansky, H. Nguyen, D. Contractor, L. Villa, J. Li, D. Bahdanau, Y. Jernite, S. Hughes, D. Fried, A. Guha, H. de Vries, and L. von Werra, "Santacoder: don't reach for the stars!" *CoRR*, vol. abs/2301.03988, 2023.

[25] Y. Ding, Z. Wang, W. U. Ahmad, H. Ding, M. Tan, N. Jain, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth, and B. Xiang, "Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion," in *NeurIPS*, 2023.

[26] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[27] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *CoRR*, vol. abs/2009.10297, 2020.

[28] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8. Soviet Union, 1966, pp. 707–710.

[29] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *CoRR*, vol. abs/2001.08361, 2020. [Online]. Available: https://arxiv.org/abs/2001.08361

[30] A. Power, Y. Burda, H. Edwards, I. Babuschkin, and V. Misra, "Grokking: Generalization beyond overfitting on small algorithmic datasets," *CoRR*, vol. abs/2201.02177, 2022. [Online]. Available: https://arxiv.org/abs/2201.02177
[31] OpenAI, "GPT-4 technical report," *CoRR*, vol. abs/2303.08774, 2023.
[32] G. Team, R. Anil, S. Borgeaud, Y. Wu, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth *et al.*, "Gemini: a family of highly capable multimodal models," *arXiv preprint arXiv:2312.11805*, 2023.
[33] "The claude 3 model family: Opus, sonnet, haiku." [Online]. Available: https://api.semanticscholar.org/CorpusID:268232499
[34] Q. Zhu, D. Guo, Z. Shao, D. Yang, P. Wang, R. Xu, Y. Wu, Y. Li, H. Gao, S. Ma *et al.*, "Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence," *arXiv preprint arXiv:2406.11931*, 2024.