DeepSeek-Coder: When the Large Language Model Meets Programming - The Rise of Code Intelligence

Daya Guo*[1], Qihao Zhu*[1,2], Dejian Yang[1], Zhenda Xie[1], Kai Dong[1], Wentao Zhang[1]
Guanting Chen[1], Xiao Bi [1], Y. Wu[1], Y.K. Li[1], Fuli Luo[1], Yingfei Xiong[2], Wenfeng Liang[1]

[1]DeepSeek-AI
[2]Key Lab of HCST (PKU), MOE; SCS, Peking University
{zhuqh, guodaya}@deepseek.com
https://github.com/deepseek-ai/DeepSeek-Coder

Abstract

The rapid development of large language models has revolutionized code intelligence in software development. However, the predominance of closed-source models has restricted extensive research and development. To address this, we introduce the DeepSeek-Coder series, a range of open-source code models with sizes from 1.3B to 33B, trained from scratch on 2 trillion tokens. These models are pre-trained on a high-quality project-level code corpus and employ a fill-in-the-blank task with a 16K window to enhance code generation and infilling. Our extensive evaluations demonstrate that DeepSeek-Coder not only achieves state-of-the-art performance among open-source code models across multiple benchmarks but also surpasses existing closed-source models like Codex and GPT-3.5. Furthermore, DeepSeek-Coder models are under a permissive license that allows for both research and unrestricted commercial use.

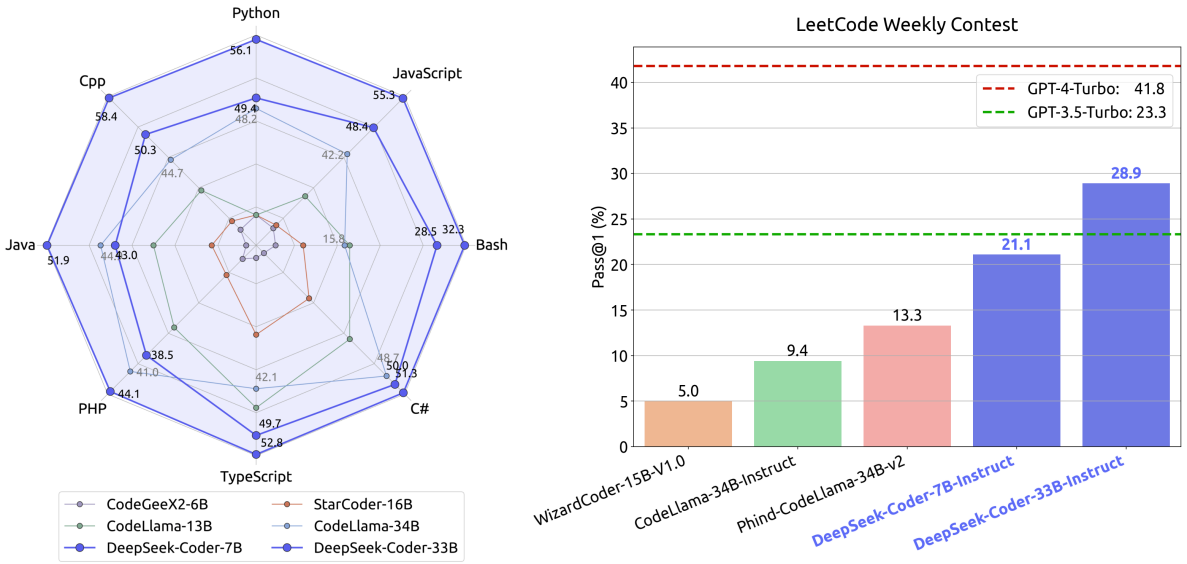

Figure 1 | The Performance of DeepSeek-Coder

*Core contributors, ordered alphabetically by the name.

1. Introduction

The field of software development has been significantly transformed by the swift advancement of large language models (OpenAI, 2023; Touvron et al., 2023), which have brought about a new era of code intelligence. These models have the potential to automate and streamline many aspects of coding, from bug detection to code generation, thereby enhancing productivity and reducing the likelihood of human error. However, a major challenge in this field is the performance gap between open-source models (Li et al., 2023; Nijkamp et al., 2022; Roziere et al., 2023; Wang et al., 2021) and closed-source models (Gemini Team, 2023; OpenAI, 2023). The giant closed-source models, while powerful, are often inaccessible to many researchers and developers due to their proprietary nature.

In response to this challenge, we present the DeepSeek-Coder series. This series comprises a range of open-source code models, varying in size from 1.3B to 33B, including the base version and instructed version for each size. Each model in the series has been trained from scratch on 2 trillion tokens sourced from 87 programming languages, ensuring a comprehensive understanding of coding languages and syntax. Besides, we attempt to organize the pre-training data at the repository level to enhance the pre-trained model's understanding capability within the context of cross-files within a repository. In addition to employing the next token prediction loss during pre-training, we have also incorporated the Fill-In-Middle (FIM) approach (Bavarian et al., 2022; Li et al., 2023). This approach is designed to further bolster the model's code completion capabilities. To meet the requirements of handling longer code inputs, we have extended the context length to 16K. This adjustment allows our models to handle more complex and extensive coding tasks, thereby increasing their versatility and applicability in various coding scenarios.

We have carried out comprehensive experiments using a variety of public code-related benchmarks. The findings reveal that among open-source models, DeepSeek-Coder-Base 33B consistently delivers superior performance across all benchmarks. Furthermore, DeepSeek-Coder-Instruct 33B surpasses *OpenAI GPT-3.5 Turbo* in the majority of the evaluation benchmarks, significantly narrowing the performance gap between *OpenAI GPT-4* and open-source models. Remarkably, despite having fewer parameters, DeepSeek-Coder-Base 7B demonstrates competitive performance when compared to models that are five times larger, such as CodeLlama-33B (Roziere et al., 2023). To summarize, our main contributions are:

- We introduce DeepSeek-Coder-Base and DeepSeek-Coder-Instruct, our advanced code-focused large language models (LLMs). Developed through extensive training on an expansive code corpus, these models exhibit proficiency in understanding 87 programming languages. Additionally, they are available in various model scales to cater to a wide range of computational and application needs.
- We make the first attempt to incorporate repository-level data construction during the pre-training phase of our models. We find that it can significantly boost the capability of cross-file code generation.
- Our analysis rigorously examines the impact of FIM training strategies on the pretraining phase of code models. The outcomes of these comprehensive studies shed light on intriguing aspects of FIM configurations, offering valuable insights that significantly contribute to the enhancement and development of code pretrained models.
- We conduct extensive evaluations of our code LLMs against a wide array of benchmarks encompassing numerous code-related tasks. The findings demonstrate that DeepSeek-Coder-Base surpasses all existing open-source code LLMs across these benchmarks. Furthermore,

with meticulous fine-tuning using instructional data, DeepSeek-Coder-Instruct achieves better performance compared to the *OpenAI GPT-3.5 Turbo* model in code-related tasks.

2. Data Collection

The training dataset of DeepSeek-Coder is composed of 87% source code, 10% English code-related natural language corpus, and 3% code-unrelated Chinese natural language corpus. The English corpus consists of materials from GitHub's Markdown and StackExchange[1], which are used to enhance the model's understanding of code-related concepts and improve its ability to handle tasks like library usage and bug fixing. Meanwhile, the Chinese corpus consists of high-quality articles aimed at improving the model's proficiency in understanding the Chinese language. In this section, we will provide an overview of how we construct the code training data. This process involves data crawling, rule-based filtering, dependency parsing, repository-level deduplication, and quality screening, as illustrated in Figure 2. In the following, we will describe the data creation procedure step by step.

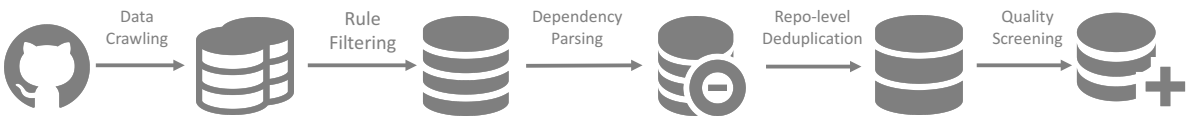

Figure 2 | The Procedure of Dataset Creation

2.1. GitHub Data Crawling and Filtering

We collect public repositories created before February 2023 on GitHub and retain only 87 programming languages, as listed in Table 1. To reduce the amount of data to be processed, we apply filtering rules similar to those used in the StarCoder project (Li et al., 2023) to preliminarily filter out lower-quality code. By applying these filtering rules, we reduce the total amount of data to only 32.8% of its original size. To make the paper self-contained, we briefly describe the filter rules used in the StarCoder Data project:

Firstly, we filter out files with an average line length exceeding 100 characters or a maximum line length surpassing 1000 characters. Additionally, we remove files with fewer than 25% alphabetic characters. Except for the XSLT programming language, we further filter out files where the string `"<?xml version="` appeared in the first 100 characters. For HTML files, we consider the ratio of visible text to HTML code. We retain files where the visible text constitutes at least 20% of the code and is no less than 100 characters. For JSON and YAML files, which typically contain more data, we only keep files that have a character count ranging from 50 to 5000 characters. This effectively removes most data-heavy files.

2.2. Dependency Parsing

In previous works (Chen et al., 2021; Li et al., 2023; Nijkamp et al., 2022; Roziere et al., 2023), large language models for code are mainly pre-trained on file-level source code, which ignores the dependencies between different files in a project. However, in practical applications, such models struggle to effectively scale to handle entire project-level code scenarios. Therefore, we

[1] `https://stackexchange.com`

Algorithm 1 Topological Sort for Dependency Analysis

```
procedure TOPOLOGICALSORT(files)
    graphs ← {}                                   ▷ Initialize an empty adjacency list
    inDegree ← {}                                 ▷ Initialize an empty dictionary for in-degrees
    for each file in files do
        graphs[file] ← []
        inDegree[file] ← 0
    end for

    for each fileA in files do
        for each fileB in files do
            if HASDEPENDENCY(fileA, fileB) then   ▷ If fileA depends on fileB
                graphs[fileB].append(fileA)       ▷ Add edge from B to A
                inDegree[fileA] ← inDegree[fileA] + 1   ▷ Increment in-degree of A
            end if
        end for
    end for

    subgraphs ← getDisconnectedSubgraphs(graphs)   ▷ Identify disconnected subgraphs
    allResults ← []
    for each subgraph in subgraphs do
        results ← []
        while length(results) ≠ NumberOfNodes(subgraph) do
            file ← argmin({inDegree[file] | file ∈ subgraph and file ∉ results})
            for each node in graphs[file] do
                inDegree[node] ← inDegree[node] − 1
            end for
            results.append(file)
        end while
        allResults.append(results)
    end for

    return allResults
end procedure
```

will consider how to leverage the dependencies between files within the same repository in this step. Specifically, we first parse the dependencies between files and then arrange these files in an order that ensures the context each file relies on is placed before that file in the input sequence. By aligning the files in accordance with their dependencies, our dataset more accurately represents real coding practices and structures. This enhanced alignment not only makes our dataset more relevant but also potentially increases the practicality and applicability of the model in handling project-level code scenarios. It's worth noting that we only consider the invocation relationships between files and use regular expressions to extract them, such as **"import"** in Python, **"using"** in C#, and **"include"** in C.

The algorithm 1 describes a topological sort for dependency analysis on a list of files within the same project. Initially, it sets up two data structures: an empty adjacency list named **"graphs"** to represent dependencies between files and an empty dictionary called **"inDegree"** for storing the in-degrees of each file. The algorithm then iterates over each file pair to identify depen-

dencies, updating **"graphs"** and **"inDegree"** accordingly. Next, it identifies any disconnected subgraphs within the overall dependency graph. For each subgraph, the algorithm employs a modified topological sort. Unlike the standard approach that selects nodes with zero in-degrees, this algorithm selects nodes with minimal in-degrees, which allows it to handle cycles within the graph. Selected nodes are added to a **"results"** list, and the in-degrees of their connected nodes are decreased. This process continues until a topologically sorted sequence is generated for each subgraph. The algorithm concludes by returning a list of these sorted sequences, and each sequence's files are concatenated to form a single training sample. To incorporate file path information, a comment indicating the file's path is added at the beginning of each file. This method ensures that the path information is preserved in the training data.

2.3. Repo-Level Deduplication

Recent studies have demonstrated the significant performance improvements that can be achieved by deduplicating training datasets for Large Language Models (LLMs). Lee et al. (2022) have shown that language model training corpora often contain numerous near-duplicates, and the performance of LLMs can be enhanced by removing long repetitive substrings. Kocetkov et al. (2022) have applied a near-deduplication method to training data, resulting in dramatic improvements, and they emphasize that near-deduplication is a crucial preprocessing step for achieving competitive performance on code benchmark tasks. In our dataset, we have also employed near-deduplication. However, there is a distinction in our approach compared to previous works. We perform deduplication at the repository level of code, rather than at the file level, as the latter approach may filter out certain files within a repository, potentially disrupting the structure of the repository. Specifically, we treat the concatenated code from the repository level as a single sample and apply the same near-deduplication algorithm to ensure the integrity of the repository structure.

2.4. Quality Screening and Decontamination

In addition to applying the filtering rules mentioned in Section 2.1, we also employ a compiler and a quality model, combined with heuristic rules, to further filter out low-quality data. This includes code with syntax errors, poor readability, and low modularity. We provide the statistical summary of source code in Table 1, which includes a total of 87 languages, detailing the disk size, number of files, and percentage for each language. The total data volume is 798 GB with 603 million files. To ensure that our code training data is not contaminated by information from the test set, which may be present on GitHub, we've implemented an n-gram filtering process. This process involves the removal of any code segments that match specific criteria. Specifically, we filter out files containing docstrings, questions, and solutions from sources such as HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), GSM8K (Cobbe et al., 2021) and MATH (Hendrycks et al., 2021). For the filtering criteria, we apply the following rules: if a piece of code includes a 10-gram string identical to any in the test data, it is excluded from our training data. In cases where the test data comprises strings that are shorter than 10-grams but no less than 3-grams, we use an exact match approach for filtering.

Language	Size (GB)	Files (k)	Prop. (%)	Language	Size (GB)	Files (k)	Prop. (%)
Ada	0.91	126	0.11	Literate Haskell	0.16	20	0.02
Agda	0.26	59	0.03	Lua	0.82	138	0.10
Alloy	0.07	24	0.01	Makefile	0.92	460	0.12
ANTLR	0.19	38	0.02	Maple	0.03	6	0.00
AppleScript	0.03	17	0.00	Mathematica	0.82	10	0.10
Assembly	0.91	794	0.11	MATLAB	0.01	1	0.00
Augeas	0.00	1	0.00	OCaml	0.91	139	0.11
AWK	0.09	53	0.01	Pascal	0.79	470	0.10
Batchfile	0.92	859	0.12	Perl	0.81	148	0.10
Bluespec	0.10	15	0.01	PHP	58.92	40,627	7.38
C	28.64	27,111	3.59	PowerShell	0.91	236	0.11
C#	58.56	53,739	7.34	Prolog	0.03	5	0.00
Clojure	0.90	295	0.11	Protocol Buffer	0.92	391	0.12
CMake	0.90	359	0.11	Python	120.68	75,188	15.12
CoffeeScript	0.92	361	0.12	R	0.92	158	0.11
Common Lisp	0.92	105	0.11	Racket	0.09	13	0.01
C++	90.87	36,006	11.39	RMarkdown	6.83	1,606	0.86
CSS	5.63	11,638	0.71	Ruby	15.01	18,526	1.88
CUDA	0.91	115	0.11	Rust	0.61	692	0.08
Dart	0.89	264	0.11	SAS	0.92	70	0.11
Dockerfile	0.04	48	0.00	Scala	0.81	971	0.10
Elixir	0.91	549	0.11	Scheme	0.92	216	0.12
Elm	0.92	232	0.12	Shell	13.92	10,890	1.74
Emacs Lisp	0.91	148	0.11	Smalltalk	0.92	880	0.12
Erlang	0.92	145	0.12	Solidity	0.85	83	0.11
F#	0.91	340	0.11	Sparql	0.10	88	0.01
Fortran	1.67	654	0.21	SQL	15.14	7,009	1.90
GLSL	0.92	296	0.11	Stan	0.20	41	0.03
Go	2.58	1,365	0.32	Standard ML	0.74	117	0.09
Groovy	0.89	340	0.11	Stata	0.91	122	0.11
Haskell	0.87	213	0.11	SystemVerilog	0.91	165	0.11
HTML	30.05	14,998	3.77	TCL	0.90	110	0.11
Idris	0.11	32	0.01	Tcsh	0.17	53	0.02
Isabelle	0.74	39	0.09	Tex	20.46	2,867	2.56
Java	148.66	134,367	18.63	Thrift	0.05	21	0.01
Java Server Pages	0.86	1072	0.11	TypeScript	60.62	62,432	7.60
JavaScript	53.84	71,895	6.75	Verilog	0.01	1	0.00
JSON	4.61	11956	0.58	VHDL	0.85	392	0.11
Julia	0.92	202	0.12	Visual Basic	0.75	73	0.09
Jupyter Notebook	14.38	2,555	1.80	XSLT	0.36	48	0.04
Kotlin	6.00	3,121	0.75	Yacc	0.72	67	0.09
Lean	0.52	68	0.07	YAML	0.74	890	0.09
Literate Agda	0.05	4	0.01	Zig	0.81	70	0.10
Literate CoffeeScript	0.01	3	0.00	**Total**	**797.92**	**603,173**	**100.00**

Table 1 | A summary of the cleaned training data for the selected programming languages.

3. Training Policy

3.1. Training Strategy

3.1.1. Next Token Prediction

The first training objective for our model is known as *next token prediction*. In this process, various files are concatenated to form a fixed-length entry. Then, these entries are used to train the model, enabling it to predict the subsequent token based on the provided context.

3.1.2. *Fill-in-the-Middle*

The second training objective for our model is known as *fill-in-the-middle*. In the code pre-training scenario, it is often necessary to generate corresponding inserted content based on the given context and subsequent text. Due to specific dependencies in a programming language, relying solely on next token prediction is insufficient to learn this fill-in-the-middle capability. Therefore, several approaches (Bavarian et al., 2022; Li et al., 2023) propose the pretraining method of Fill-in-the-Midlle (FIM). This approach involves randomly dividing the text into three parts, then shuffling the order of these parts and connecting them with special characters. This method aims to incorporate a fill-in-the-blank pretraining task during the training process. Within the FIM methodology, two distinct modes are employed: PSM (Prefix-Suffix-Middle) and SPM (Suffix-Prefix-Middle). In the PSM mode, the training corpus is organized in the sequence of $Prefix, Suffix, Middle$, aligning the text in a way that the middle segment is flanked by the prefix and suffix. Conversely, the SPM mode arranges the segments as $Suffix, Prefix, Middle$, presenting a different structural challenge. These modes are instrumental in enhancing the model's capability to handle various structural arrangements in code, providing a robust training framework for advanced code prediction tasks.

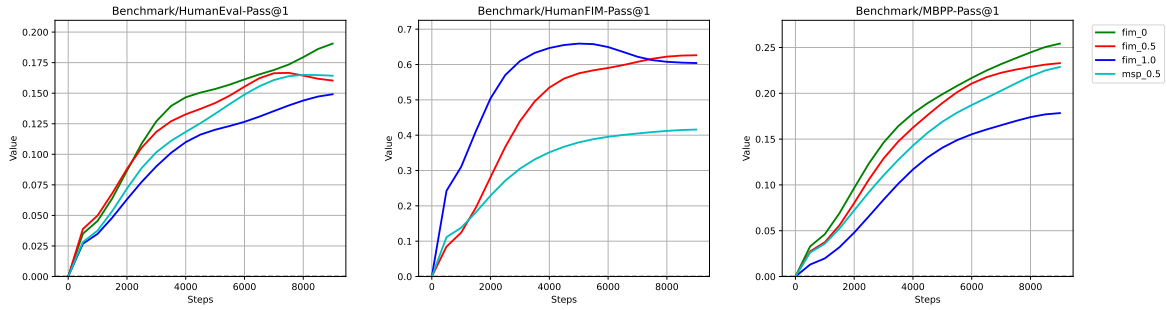

Figure 3 | The effectiveness of using FIM objective.

To determine the effectiveness of various hyperparameters within the FIM approach, we conducted a series of ablation experiments.

Experiment Settings: In this experiment, we employ DeepSeek-Coder-Base 1.3B as our model architecture. We focused on a Python subset from our training dataset to streamline the experimental process. Our primary objective was to assess the efficacy of the Fill-in-the-Middle (FIM) technique, utilizing the HumanEval-FIM benchmark (Fried et al., 2022). This benchmark specializes in a single-line FIM task for Python, in which one line of code from a HumanEval solution is randomly obscured, testing the model's proficiency in predicting the missing line. We hypothesize that the PSM mode may exhibit subtle differences compared to the traditional next-token prediction objective. This is primarily because PSM involves rearranging the order of the original text, potentially impacting the learning dynamics of the model. Therefore, we implement the PSM mode for FIM across four distinct configurations: 0% FIM rate, 50% FIM rate, 100% FIM rate, and 50% MSP rate. The Masked Span Prediction (MSP) strategy, initially introduced in T5 (Raffel et al., 2023), conceals multiple text spans and trains the model to reconstruct these segments. According to CodeGen2.5 (Nijkamp et al., 2023), MSP may enhance FIM performance compared to PSM. Thus, we include this method in our comparative analysis.

Results: The outcomes of our experiment are illustrated in Figure 3. While the model demonstrates peak performance on the HumanEval-FIM with a 100% FIM rate, this configuration also results in the weakest code completion capability. This indicates a trade-off between FIM and

code completion abilities. Moreover, we observe that with a 50% PSM rate, the model outperforms the MSP strategy. To achieve a balance between FIM efficiency and code completion proficiency, we ultimately choose the 50% PSM rate as our preferred training policy.

In our implementation, we have introduced three sentinel tokens specifically for this task. For each code file, we initially divide its content into three segments, denoted as f_{pre}, f_{middle}, and f_{suf}. Using the PSM mode, we construct the training example as follows:

$$\texttt{<|fim\_start|>}f_{pre}\texttt{<|fim\_hole|>}f_{suf}\texttt{<|fim\_end|>}f_{middle}\texttt{<|eos\_token|>}$$

We implement the Fill-in-the-Middle (FIM) method at the document level before the packing process, as proposed in the original work by Bavarian et al. (2022). This is done with an FIM rate of 0.5, following the PSM mode.

3.2. Tokenizer

For the tokenization process, we employ the HuggingFace Tokenizer library[2] to train Byte Pair Encoding (BPE) tokenizers, as outlined in Sennrich et al. (2015) (Sennrich et al., 2015), on a subset of our training corpus. Ultimately, we utilize a tokenizer configured with a vocabulary size of 32,000.

3.3. Model Architecture

We develop a range of models with varying parameters to cater to diverse applications, including models with 1.3B, 6.7B, and 33B parameters. These models are built upon the same framework as the DeepSeek Large Language Model (LLM) outlined by DeepSeek-AI (2024). Each model is a decoder-only Transformer, incorporating Rotary Position Embedding (RoPE) as described by Su et al. (2023). Notably, the DeepSeek 33B model integrates Grouped-Query-Attention (GQA) with a group size of 8, enhancing both training and inference efficiency. Additionally, we employ FlashAttention v2 (Dao, 2023) to expedite the computation involved in the attention mechanism. The architectural details of our models are summarized in Table 2.

3.4. Optimization

Following DeepSeek LLM (DeepSeek-AI, 2024), we use AdamW (Loshchilov and Hutter, 2019) as the optimizer with β_1 and β_2 values of 0.9 and 0.95. We adapt batch sizes and learning rates by the scaling laws suggested in DeepSeek LLM. For the learning rate scheduling, we implement a three-stage policy, which includes 2000 warm-up steps, and set the final learning rate to 10% of the initial rate. Notably, the learning rate at each stage is scaled down to $\sqrt{\frac{1}{10}}$ of the preceding stage's rate, following the guidelines established in DeepSeek LLM (DeepSeek-AI, 2024).

3.5. Environments

Our experiments are conducted using the HAI-LLM (High-Flyer, 2023) framework, known for its efficiency and lightweight approach in training large language models. This framework incorporates a variety of parallelism strategies to optimize computational efficiency. These include tensor parallelism (Korthikanti et al., 2023), alongside ZeRO data parallelism (Rajbhandari et al., 2020) and PipeDream pipeline parallelism (Narayanan et al., 2019). Our experiments

[2] `https://github.com/huggingface/tokenizers`

Hyperparameter	DeepSeek-Coder 1.3B	DeepSeek-Coder 6.7B	DeepSeek-Coder 33B
Hidden Activation	SwiGLU	SwiGLU	SwiGLU
Hidden size	2048	4096	7168
Intermediate size	5504	11008	19200
Hidden layers number	24	32	62
Attention heads number	16	32	56
Attention	Multi-head	Multi-head	Grouped-query (8)
Batch Size	1024	2304	3840
Max Learning Rate	5.3e-4	4.2e-4	3.5e-4

Table 2 | Hyperparameters of DeepSeek-Coder.

utilize clusters outfitted with NVIDIA A100 and H800 GPUs. In the A100 cluster, each node is configured with 8 GPUs, interconnected in pairs using NVLink bridges. The H800 cluster is similarly arranged, with each node containing 8 GPUs. These GPUs are interconnected using a combination of NVLink and NVSwitch technologies, ensuring efficient data transfer within nodes. To facilitate seamless communication between nodes in both A100 and H800 clusters, we employ InfiniBand interconnects, known for their high throughput and low latency. This setup provides a robust and efficient infrastructure for our computational experiments.

3.6. Long Context

To enhance the capabilities of DeepSeek-Coder in handling extended contexts, particularly for scenarios like repository-level code processing, we have reconfigured the RoPE (Su et al., 2023) parameters to extend the default context window. Following previous practices (Chen et al., 2023; kaiokendev, 2023), we employed a linear scaling strategy, increasing the scaling factor from 1 to 4 and altering the base frequency from 10000 to 100000. The model underwent an additional 1000 steps of training, using a batch size of 512 and a sequence length of 16K. The learning rate was maintained as in the final pre-training phase. Theoretically, these modifications enable our model to process up to 64K tokens in context. However, empirical observations suggest that the model delivers its most reliable outputs within a 16K token range. Future research will continue to refine and evaluate the long-context adaptation methodology, aiming to further enhance DeepSeek-Coder's efficiency and user-friendliness in processing extended contexts.

3.7. Instruction Tuning

We develop DeepSeek-Coder-Instruct by enhancing the DeepSeek-Coder-Base through instruction-based fine-tuning using high-quality data. This data comprises helpful and impartial human instructions, structured by the Alpaca Instruction format (Taori et al., 2023). To demarcate each dialogue turn, we employed a unique delimiter token <|EOT|> to signify the conclusion of each segment. For training, we use a cosine schedule with 100 warm-up steps and an initial learning rate 1e-5. We also use a batch size of 4M tokens and 2B tokens in total.

An example of using DeepSeek-Coder-Instruct 34B is depicted in Figure 4. This example is a multi-turn dialogue scenario for building a snake game. Initially, we ask the model to write a game snake using pygame. The model successfully creates a basic snake game that can run without bugs. To improve the game, we further request adding a scoring system in the top left corner. The model then introduces a **"score"** variable and a **"display_score"** function, along with an explanation of how to integrate these features. This example illustrates DeepSeek-Coder-Instruct's ability to provide complete solutions in multi-turn dialogue settings. More cases can be found in the Appendix A.

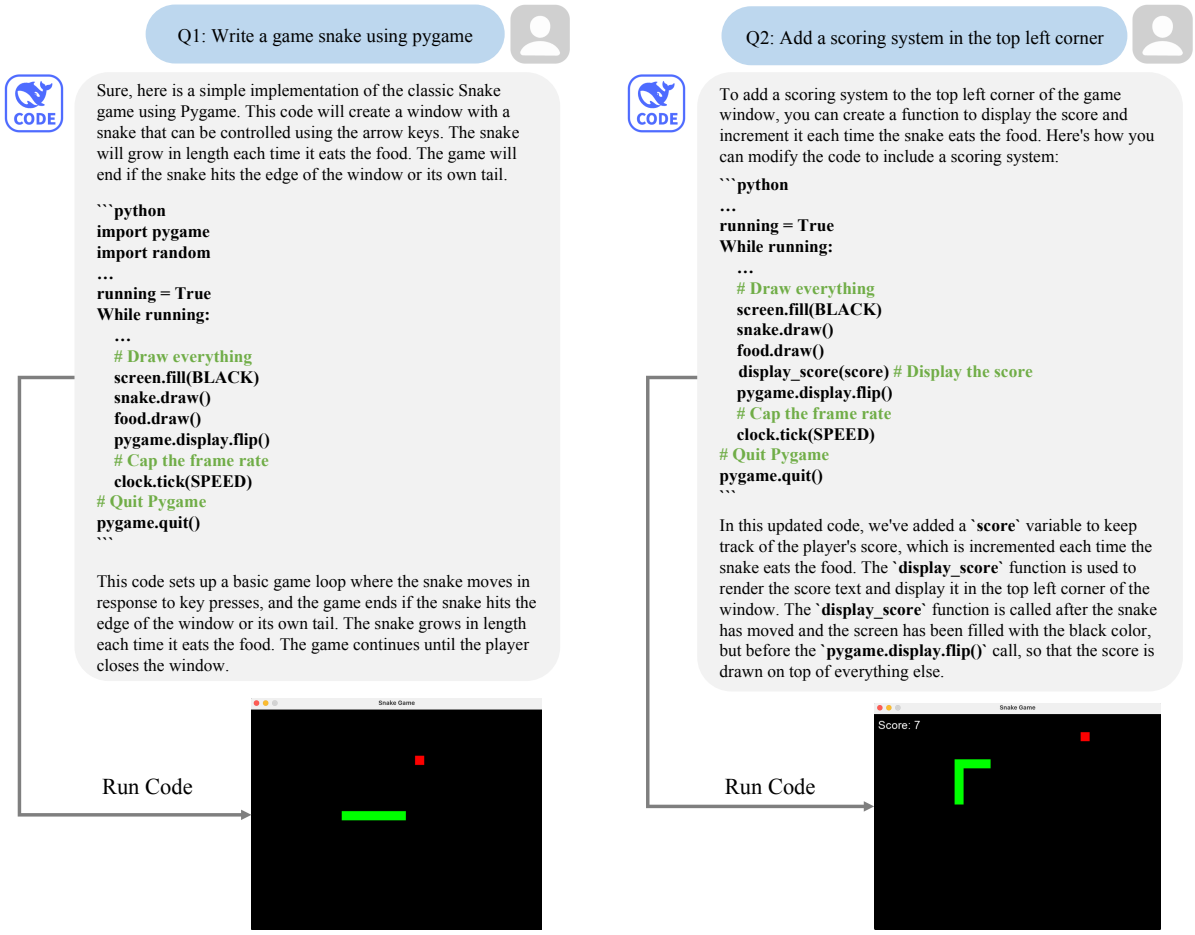

Figure 4 | An example of responses from DeepSeek-Coder-Instruct 33B in a multi-turn setting.

4. Experimental Results

In this section, we evaluate DeepSeek-Coder on four tasks, including code generation (§4.1), FIM code completion (§4.2), cross-file code completion (§4.3) and program-based math reasoning (§4.4). We compare DeepSeek-Coder with the previous state-of-the-art large language models:

- **CodeGeeX2** (Zheng et al., 2023) represents the second generation of the multilingual code generation model CodeGeeX. It is developed using the ChatGLM2 (Du et al., 2022) architecture and is enhanced with an extensive dataset of coding examples.
- **StarCoder** (Li et al., 2023) is a publicly accessible model with a substantial parameter count of 15 billion. It is specifically trained on a meticulously curated subset of the Stack dataset (Kocetkov et al., 2022), covering 86 programming languages, ensuring its proficiency across a wide range of coding tasks.
- **CodeLlama** (Roziere et al., 2023) encompasses a series of code-centric Large Language Models (LLMs) that are derivatives of LLaMA2 (Touvron et al., 2023). Available in three sizes — 7B, 13B, and 34B — these models undergo continued training on a vast 500 billion token code corpus, building upon the foundational LLaMA2 architecture.
- **code-cushman-001** Chen et al. (2021) is a 12 billion parameter model developed by OpenAI and served as the initial model for Github Copilot.
- **GPT-3.5 and GPT-4** (OpenAI, 2023) are advanced generative AI models developed by OpenAI. While they are not explicitly trained for code generation, they also demonstrate

notable performance in this domain. Their effectiveness in handling code generation tasks is largely attributed to their massive scale in terms of parameter count.

4.1. Code Generation

HumanEval and MBPP Benchmarks The HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) benchmarks are widely used for evaluating code LLMs. HumanEval consists of 164 hand-written Python problems that are validated using test cases to assess the code generated by a Code LLM in a zero-shot setting, while the MBPP benchmark includes 500 problems in a few-shot setting. To evaluate the model's multilingual capabilities, we expanded the Python problems of Humaneval Benchmark to seven additional commonly used programming languages, namely C++, Java, PHP, TypeScript (TS), C#, Bash, and JavaScript (JS) (Cassano et al., 2023). For both benchmarks, We adopted a greedy search approach and re-implemented the baseline results using the same script and environment for fair comparison.

Model	Size	Python	C++	Java	PHP	TS	C#	Bash	JS	Avg	MBPP
Multilingual Base Models											
code-cushman-001	12B	33.5%	31.9%	30.6%	28.9%	31.3%	22.1%	11.7%	-	-	-
CodeGeeX2	6B	36.0%	29.2%	25.9%	23.6%	20.8%	29.7%	6.3%	24.8%	24.5%	36.2%
StarCoderBase	16B	31.7%	31.1%	28.5%	25.4%	34.0%	34.8%	8.9%	29.8%	28.0%	42.8%
CodeLlama	7B	31.7%	29.8%	34.2%	23.6%	36.5%	36.7%	12.0%	29.2%	29.2%	38.6%
CodeLlama	13B	36.0%	37.9%	38.0%	34.2%	45.2%	43.0%	16.5%	32.3%	35.4%	48.4%
CodeLlama	34B	48.2%	44.7%	44.9%	41.0%	42.1%	48.7%	15.8%	42.2%	41.0%	55.2%
DeepSeek-Coder-Base	1.3B	34.8%	31.1%	32.3%	24.2%	28.9%	36.7%	10.1%	28.6%	28.3%	46.2%
DeepSeek-Coder-Base	6.7B	49.4%	50.3%	43.0%	38.5%	49.7%	50.0%	28.5%	48.4%	44.7%	60.6%
DeepSeek-Coder-Base	33B	**56.1%**	**58.4%**	**51.9%**	**44.1%**	**52.8%**	**51.3%**	**32.3%**	**55.3%**	**50.3%**	**66.0%**
Instruction-Tuned Models											
GPT-3.5-Turbo	-	76.2%	63.4%	69.2%	60.9%	69.1%	70.8%	42.4%	67.1%	64.9%	70.8%
GPT-4	-	**84.1%**	**76.4%**	**81.6%**	**77.2%**	**77.4%**	**79.1%**	**58.2%**	**78.0%**	**76.5%**	**80.0%**
DeepSeek-Coder-Instruct	1.3B	65.2%	45.3%	51.9%	45.3%	59.7%	55.1%	12.7%	52.2%	48.4%	49.4%
DeepSeek-Coder-Instruct	6.7B	78.6%	63.4%	68.4%	68.9%	67.2%	72.8%	36.7%	72.7%	66.1%	65.4%
DeepSeek-Coder-Instruct	33B	**79.3%**	**68.9%**	**73.4%**	**72.7%**	**67.9%**	**74.1%**	**43.0%**	**73.9%**	**69.2%**	**70.0%**

Table 3 | Performance of approaches on the Multilingual HumanEval and MBPP Benchmarks.

The results are presented in Table 3. As we can see, DeepSeek-Coder-Base achieves state-of-the-art performance with an average accuracy of 50.3% on HumanEval and 66.0% on MBPP. In comparison to the similarly sized open-source model CodeLlama-Base 34B, our model has demonstrated a notable improvement of 9% and 11% in accuracy, respectively. It's worth noting that even our smaller model, DeepSeek-Coder-Base 6.7B, surpasses the performance of CodeLlama-Base 34B. After instruction fine-tuning, our model surpasses the closed-source GPT-3.5-Turbo model in HumanEval benchmark, significantly reducing the performance gap between OpenAI GPT-4 and open-source models.

DS-1000 Benchmark HumanEval and MBPP have a significant drawback in that they rely heavily on straightforward programming tasks that may not accurately represent the kind of code most programmers typically write. In contrast, the DS-1000 benchmark, as introduced in the work by Lai et al. (2023), offers a comprehensive collection of 1,000 practical and realistic data science workflows across seven different libraries. This benchmark evaluates code generation by executing it against specific test cases. What sets DS-1000 apart is its categorization of problems based on the libraries involved, which encompass Matplotlib, NumPy, Pandas, SciPy, Scikit-

Learn, PyTorch, and TensorFlow. The benchmark assesses the performance of base models in the code completion setting and we provide pass@1 results for each library, as well as overall score.

The results of DS-1000 benchmark are shown in Table 4. As can be seen from the table, the DeepSeek-Coder model achieves relatively high accuracy in all libraries, demonstrating that our model is not only capable of generating good code but also of using libraries more accurately in real data science workflows.

Model	Size	Matplotlib	Numpy	Pandas	Pytorch	Scipy	Scikit-Learn	Tensorflow	Avg
CodeGeeX2	6B	38.7%	26.8%	14.4%	11.8%	19.8%	27.0%	17.8%	22.9%
StarCoder-Base	16B	43.2%	29.1%	11.0%	20.6%	23.6%	32.2%	15.6%	24.6%
CodeLlama-Base	7B	41.9%	24.6%	14.8%	16.2%	18.9%	17.4%	17.8%	22.1%
CodeLlama-Base	13B	46.5%	28.6%	18.2%	19.1%	18.9%	27.8%	33.3%	26.8%
CodeLlama-Base	34B	50.3%	42.7%	23.0%	25.0%	28.3%	33.9%	40.0%	34.3%
DeepSeek-Coder-Base	1.3B	32.3%	21.4%	9.3%	8.8%	8.5%	16.5%	8.9%	16.2%
DeepSeek-Coder-Base	6.7B	48.4%	35.5%	20.6%	19.1%	22.6%	38.3%	24.4%	30.5%
DeepSeek-Coder-Base	33B	**56.1%**	**49.6%**	**25.8%**	**36.8%**	**36.8%**	**40.0%**	**46.7%**	**40.2%**

Table 4 | Performance of different approaches on the DS-1000-Tasks.

LeetCode Contest Benchmark To further validate the model's capability in real-world programming problems, we construct the LeetCode Contest benchmark[3]. LeetCode[4] presents competition-level problems, offering significant challenges that test the model's problem understanding and code generation skills. We collected the latest problems from LeetCode Contests to prevent the appearance of both the problems or their solutions in our pre-training data. A total of 180 problems were collected from July 2023 to January 2024. For each problem, we collected 100 test cases to ensure the test coverage. We use the template "{problem_description}\nPlease complete the code below to solve the above problem:\n```python\n{code_template}\n```" to build the instruction prompt.

The evaluation results are shown in Table 5. In our evaluation, the DeepSeek-Coder models demonstrate remarkable performance over current open-source coding models. Specifically, the DeepSeek-Coder-Instruct 6.7B and 33B achieve Pass@1 scores of 19.4% and 27.8% respectively in this benchmark. This performance notably surpasses existing open-sourced models such as Code-Llama-33B. The DeepSeek-Coder-Instruct 33B is the only open-sourced model that outperforms OpenAI's GPT-3.5-Turbo in this task. However, there remains a substantial performance gap when compared to the more advanced GPT-4-Turbo.

Our analysis indicates that the implementation of Chain-of-Thought (CoT) prompting notably enhances the capabilities of DeepSeek-Coder-Instruct models. This improvement becomes particularly evident in the more challenging subsets of tasks. By adding the directive, "You need first to write a step-by-step outline and then write the code." following the initial prompt, we have observed enhancements in performance. This observation leads us to believe that the process of first crafting detailed code descriptions assists the model in more effectively understanding and addressing the intricacies of logic and dependencies in coding tasks, particularly those of higher complexity. Therefore, we strongly recommend employing CoT prompting strategies when utilizing DeepSeek-Coder-Instruct models for complex coding challenges. Such an approach promotes a more methodical and logical framework for problem-solving, potentially resulting in more precise and efficient outcomes in code generation tasks.

[3]We have published this benchmark in `https://github.com/deepseek-ai/DeepSeek-Coder/tree/main/Evaluation/LeetCode`.

[4]`https://leetcode.com/`

Model	Size	Easy (45)	Medium (91)	Hard (44)	Overall(180)
WizardCoder-V1.0	15B	17.8%	1.1%	0.0%	5.0%
CodeLlama-Instruct	34B	24.4%	4.4%	4.5%	9.4%
Phind-CodeLlama-V2	34B	26.7%	8.8%	9.1%	13.3%
GPT-3.5-Turbo	-	46.7%	15.4 %	15.9%	23.3%
GPT-3.5-Turbo + CoT	-	42.2%	15.4%	20.5%	23.3%
GPT-4-Turbo	-	**73.3%**	31.9%	25.0%	40.6%
GPT-4-Turbo + CoT	-	71.1%	**35.2%**	**25.0%**	**41.8%**
DeepSeek-Coder-Instruct	1.3B	22.2%	1.1%	4.5%	7.2%
DeepSeek-Coder-Instruct + CoT	1.3B	22.2%	2.2%	2.3%	7.2%
DeepSeek-Coder-Instruct	6.7B	44.4%	12.1%	9.1%	19.4%
DeepSeek-Coder-Instruct + CoT	6.7B	44.4%	17.6%	4.5%	21.1%
DeepSeek-Coder-Instruct	33B	**57.8%**	22.0%	9.1%	27.8%
DeepSeek-Coder-Instruct + CoT	33B	53.3%	**25.3%**	**11.4%**	**28.9%**

Table 5 | Performance of different models on the LeetCode Contest Benchmark.

It is important to acknowledge that despite our diligent efforts to gather the most recent code questions for model testing, the possibility of data contamination cannot be entirely ruled out. We observed that the GPT-4-Turbo and DeepSeek-Coder models achieved higher scores in the LeetCode Contest held in July and August. We encourage the research community to consider the potential issue of data contamination when evaluating models in future studies using our released LeetCode data.

4.2. Fill-in-the-Middle Code Completion

DeepSeek-Coder models are trained with a 0.5 FIM (Fill-In-the-Middle) rate during their pre-training phase. This specialized training strategy empowers the model to proficiently generate code by filling in blanks based on the surrounding context, both prefix and suffix, of the given code snippet. This capability is particularly advantageous in the realm of code completion tools. Several open-source models have emerged with similar capabilities. Notable among these are SantaCoder (Allal et al., 2023), StarCoder (Li et al., 2023), and CodeLlama (Roziere et al., 2023). These models have set a precedent in the field of code generation and completion. In evaluating the performance DeepSeek-Coder models, we conducted a comparative analysis with the aforementioned models. The benchmark for this comparison was the Single-Line Infilling benchmarks, encompassing three different programming languages, as proposed by Allal et al. (2023). This benchmark uses the line exact match accuracy as the evaluation metric.

Model	Size	python	java	javascript	Mean
SantaCoder	1.1B	44.0%	62.0%	74.0%	69.0%
StarCoder	16B	62.0%	73.0%	74.0%	69.7%
CodeLlama-Base	7B	67.6%	74.3%	80.2%	69.7%
CodeLlama-Base	13B	**68.3**%	77.6%	80.7%	75.5%
DeepSeek-Coder-Base	1B	57.4%	82.2%	71.7%	70.4%
DeepSeek-Coder-Base	7B	66.6%	**88.1**%	79.7%	80.7%
DeepSeek-Coder-Base	33B	65.4%	86.6%	**82.5**%	**81.2**%

Table 6 | Performance of different approaches on the FIM-Tasks.

The evaluation results are shown in Table 6. Despite being the smallest model with a capacity

of 1.3 billion parameters, DeepSeek-Coder outperforms its larger counterparts, StarCoder and CodeLlama, in these benchmarks. This superior performance can be attributed to the high quality of the pre-trained data utilized by DeepSeek-Coder. Furthermore, a notable trend observed is the correlation between the size of the model and its performance. As the model size increases, there is a corresponding and responsible enhancement in performance. This trend underscores the importance of model capacity in achieving higher accuracy in code completion tasks. Based on these findings, we recommend the deployment of the DeepSeek-Coder-Base 6.7B model in code completion tools. This recommendation is grounded in the model's demonstrated balance between efficiency and accuracy. The DeepSeek-Coder-Base 6.7B model, with its substantial parameter size, has proven to be highly effective in the context of code completion, making it an ideal choice for integrating advanced computational capabilities into coding environments.

4.3. Cross-File Code Completion

In this section, we will evaluate the performance of existing open-source models in cross-file code completion tasks. Unlike code generation discussed in the previous section, cross-file code completion requires the model to access and understand repositories that span multiple files with numerous cross-file dependencies. We use CrossCodeEval (Ding et al., 2023) to evaluate the capabilities of currently available open-source code models of 7B scale in cross-file completion tasks. This dataset is constructed on a diverse set of real-world, open-sourced, permissively licensed repositories in four popular programming languages: Python, Java, TypeScript, and C#. The dataset is specifically designed to strictly require cross-file context for accurate completion. Notably, this dataset was constructed from repositories created between March and June 2023, while our pre-training data only includes code created before February 2023, which ensures that this dataset was not present in our pre-training data, thus avoiding data leakage.

Model	Size	Python		Java		TypeScript		C#	
		EM	ES	EM	ES	EM	ES	EM	ES
CodeGeex2	6B	8.11%	59.55%	7.34%	59.60%	6.14%	55.50%	1.70%	51.66%
+ Retrieval		10.73%	61.76%	10.10%	59.56%	7.72%	55.17%	4.64%	52.30%
StarCoder-Base	7B	6.68%	59.55%	8.65%	62.57%	5.01%	48.83%	4.75%	59.53%
+ Retrieval		13.06%	64.24%	15.61%	64.78%	7.54%	42.06%	14.20%	65.03%
CodeLlama-Base	7B	7.32%	59.66%	9.68%	62.64%	8.19%	58.50%	4.07%	59.19%
+ Retrieval		13.02%	64.30%	16.41%	**64.64%**	12.34%	60.64%	13.19%	63.04%
DeepSeek-Coder-Base	6.7B	9.53%	61.65%	10.80%	61.77%	9.59%	60.17%	5.26%	61.32%
+ Retrieval		**16.14%**	66.51%	**17.72%**	63.18%	**14.03%**	**61.77%**	**16.23%**	**63.42%**
+ Retrieval w/o Repo Pre-training		16.02%	**66.65%**	16.64%	61.88%	13.23%	60.92%	14.48%	62.38%

Table 7 | Performance of different models on cross-file code completion.

In our evaluation of various models, we set the maximum sequence length to 2048 tokens, the maximum output length to 50 tokens, and a limit of 512 tokens for the cross-file context. For the cross-file context, we utilize the official BM25 search results provided by Ding et al. (2023). Evaluation metrics include exact match and edit similarity. The results, presented in Table 7, demonstrate that DeepSeek-Coder consistently outperforms other models in cross-file completion tasks across multiple languages, showcasing its superior practical application capabilities. When only utilizing file-level code corpus (**w/o Repo Pre-training**) to pre-train DeepSeek-Coder, we observe a decrease in performance in the Java, TypeScript, and C# languages, indicating the effectiveness of the repository-level pre-training.

4.4. Program-based Math Reasoning

Program-based math reasoning involves evaluating a model's ability to understand and solve mathematical problems through programming. This type of reasoning is critical in fields such as data analysis and scientific computing. To conduct this assessment, we utilize the Program-Aided Math Reasoning (PAL) method as outlined in Gao et al. (2023). This approach is applied across seven distinct benchmarks, each offering unique challenges and contexts. These benchmarks includes GSM8K (Cobbe et al., 2021), MATH (Hendrycks et al., 2021), GSM-Hard (Gao et al., 2023), SVAMP (Patel et al., 2021), TabMWP (Lu et al., 2022), ASDiv (Miao et al., 2020) and MAWPS (Gou et al., 2023). In each of these benchmarks, the model is prompted to alternately describe a solution step in natural language and then execute that step with code. As seen in Table 8, DeepSeek-Coder models achieve a remarkable performance across all benchmarks, especially the 33B variant, which demonstrates the potential of using such models in applications that require complex mathematical computations and problem-solving abilities.

Model	Size	GSM8k	MATH	GSM-Hard	SVAMP	TabMWP	ASDiv	MAWPS	Avg
Multilingual Base Models									
CodeGeex-2	7B	22.2%	9.7%	23.6%	39.0%	44.6%	48.5%	66.0%	36.2%
StarCoder-Base	16B	23.4%	10.3%	23.0%	42.4%	45.0%	54.9%	81.1%	40.0%
CodeLlama-Base	7B	31.2%	12.1%	30.2%	54.2%	52.9%	59.6%	82.6%	46.1%
CodeLlama-Base	13B	43.1%	14.4%	40.2%	59.2%	60.3%	63.6%	85.3%	52.3%
CodeLlama-Base	34B	58.2%	21.2%	51.8%	70.3%	69.8%	70.7%	91.8%	62.0%
DeepSeek-Coder-Base	1.3B	14.6%	16.8%	14.5%	36.7%	30.0%	48.2%	62.3%	31.9%
DeepSeek-Coder-Base	6.7B	43.2%	19.2%	40.3%	58.4%	67.9%	67.2%	87.0%	54.7%
DeepSeek-Coder-Base	33B	**60.7%**	**29.1%**	**54.1%**	**71.6%**	**75.3%**	**76.7%**	**93.3%**	**65.8%**

Table 8 | Performance of different approaches on the program-aid math reasoning tasks.

5. Continue Pre-Training From General LLM

To further enhance the natural language understanding and mathematical reasoning abilities of the DeepSeek-Coder model, we perform additional pre-training from the general language model DeepSeek-LLM-7B Base (DeepSeek-AI, 2024) on 2 trillion tokens, resulting in DeepSeek-Coder-v1.5 7B. For this pre-training, we specifically use the data sources listed in Table 9. Unlike DeepSeek-Coder, DeepSeek-Coder-v1.5 employs solely a next token prediction objective with a 4K context length during its pre-training phase.

Data Source	Percentage
Source Code	70%
Markdown and StackExchange	10%
Natural language related to code	7%
Natural language related to math	7%
Bilingual (Chinese-English) natural language	6%

Table 9 | Data sources for DeepSeek-Coder-v1.5 7B pre-training

We conduct a comparison between DeepSeek-Coder-v1.5 7B and DeepSeek-Coder 6.7B, and re-run all benchmarks using our evaluation pipeline to ensure a fair comparison. We evaluate performance across a wide range of tasks, which can be categorized as follows:

- **Programming**: This category includes evaluations in a multilingual setting using the HumanEval dataset by Chen et al. (2021), as well as evaluations in a Python setting using the MBPP dataset by Austin et al. (2021)
- **Math Reasoning**: We assess performance on math reasoning tasks using the GSM8K benchmark (Cobbe et al., 2021) and the MATH (Hendrycks et al., 2021) benchmark [4]. These tasks involve solving math problems by generating programs.
- **Natural Language** Our evaluation in natural language tasks includes MMLU (Hendrycks et al., 2020), BBH (Suzgun et al., 2022), HellaSwag (Zellers et al., 2019), Winogrande (Sakaguchi et al., 2021), and ARC-Challenge (Clark et al., 2018) benchmarks.

The results for the Base and Instruct models are presented in Table 10. It is observed that the DeepSeek-Coder-Base-v1.5 model, despite a slight decrease in coding performance, shows marked improvements across most tasks when compared to the DeepSeek-Coder-Base model. In particular, in the Math Reasoning and Natural Language categories, DeepSeek-Coder-Base-v1.5 significantly outperforms its predecessor across all benchmarks, which also demonstrates significant improvements in its mathematical reasoning and natural language processing capabilities.

		Programming		Math Reasoning		Natural Language				
Models	Size	HumanEval	MBPP	GSM8K	MATH	MMLU	BBH	HellaSwag	WinoG	ARC-C
DeepSeek-Coder-Base	6.7B	**44.7%**	**60.6%**	43.2%	19.2%	36.6%	44.3%	53.8%	57.1%	32.5%
DeepSeek-Coder-Base-v1.5	6.9B	43.2%	60.4%	**62.4%**	**24.7%**	**49.1%**	**55.2%**	**69.9%**	**63.8%**	**47.2%**
DeepSeek-Coder-Instruct	6.7B	**66.1%**	**65.4%**	62.8%	28.6%	37.2%	46.9%	55.0%	57.6%	37.4%
DeepSeek-Coder-Instruct-v1.5	6.9B	64.1%	64.6%	**72.6%**	**34.1%**	**49.5%**	**53.3%**	**72.2%**	**63.4%**	**48.1%**

Table 10 | Comparative analysis of performance between DeepSeek-Coder-Base and DeepSeek-Coder-Base-v1.5. Math tasks are solved through programming.

6. Conclusion

In this technical report, we introduce a series of specialized Large Language Models (LLMs) for coding, named DeepSeek-Coder, available in three distinct scales: 1.3B, 6.7B, and 33B parameters. These models are uniquely trained on a meticulously curated project-level code corpus, utilizing a "fill-in-the-blank" pre-training objective to enhance code infilling capabilities. A significant advancement is the extension of the models' context window to 16,384 tokens, thereby greatly improving their effectiveness in handling extensive code generation tasks. Our evaluations reveal that the most advanced model in our series, DeepSeek-Coder-Base 33B surpasses existing open-source code models across a variety of standard tests. Impressively, the DeepSeek-Coder-Base 6.7B model, despite its smaller scale, delivers performance on par with the 34B parameter CodeLlama, a testament to the high quality of our pretraining corpus.

To augment the zero-shot instruction capabilities of the DeepSeek-Coder-Base models, we have fine-tuned them with high-quality instructional data. This has led to the DeepSeek-Coder-Instruct 33B model outperforming OpenAI's GPT-3.5 Turbo in a range of coding-related tasks, showcasing its exceptional proficiency in code generation and understanding.

To further improve the natural language understanding capabilities of the DeepSeek-Coder-Base models, we have conducted additional pretraining based on the DeepSeek-LLM 7B checkpoint. This additional training involved processing a diverse dataset comprising 2 billion tokens, including natural language, code, and mathematical data. The result is the creation of a new

and improved code model, DeepSeek-Coder-v1.5. Our observations indicate that DeepSeek-Coder-v1.5 not only maintains its predecessor's high-level coding performance but also exhibits enhanced natural language comprehension. This advancement underscores our belief that the most effective code-focused Large Language Models (LLMs) are those built upon robust general LLMs. The reason is evident: to effectively interpret and execute coding tasks, these models must also possess a deep understanding of human instructions, which often come in various forms of natural language. Looking ahead, our commitment is to develop and openly share even more powerful code-focused LLMs based on larger-scale general LLMs.

Acknowledgements

We would like to express our gratitude to Bo Liu, Chengqi Deng, Chong Ruan, Damai Dai, Jiashi Li, Kang Guan, Mingchuan Zhang, Panpan Huang, Shuiping Yu, Shirong Ma, Yaofeng Sun, Yishi Piao, Zhihong Shao, and Zhewen Hao for their invaluable discussions and assistance during training DeepSeek-Coder models.

References

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

J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton. Program synthesis with large language models, 2021.

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. arXiv preprint arXiv:2207.14255, 2022.

F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman, et al. Multipl-e: a scalable and polyglot approach to benchmarking neural code generation. IEEE Transactions on Software Engineering, 2023.

M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374, 2021.

S. Chen, S. Wong, L. Chen, and Y. Tian. Extending context window of large language models via positional interpolation. arXiv preprint arXiv:2306.15595, 2023.

P. Clark, I. Cowhey, O. Etzioni, T. Khot, A. Sabharwal, C. Schoenick, and O. Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. arXiv preprint arXiv:1803.05457, 2018.

K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, et al. Training verifiers to solve math word problems. arXiv preprint arXiv:2110.14168, 2021.

T. Dao. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023.

DeepSeek-AI. Deepseek llm: Scaling open-source language models with longtermism. arXiv preprint arXiv:2401.02954, 2024.