

# DeepSeek-Coder：当大语言模型遇见编程——代码智能的崛起

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

<sup>1</sup>DeepSeek-AI

<sup>2</sup>Key Lab of HCST (PKU), MOE; SCS, Peking University

{zhuqh, guodaya}@deepseek.com

<https://github.com/deepseek-ai/DeepSeek-Coder>

## Abstract

大语言模型的快速发展彻底改变了软件开发中的代码智能。然而，闭源模型的主导地位限制了广泛的研究与开发。为解决这一问题，我们推出了 DeepSeek-Coder 系列，这是一组规模从 1.3B 到 33B 的开源代码模型，均在 2 万亿 token 上从头训练。这些模型在高质量的项目级代码语料库上进行预训练，并采用 16K 窗口大小的填空任务以增强代码生成与补全能力。我们的广泛评估表明，DeepSeek-Coder 不仅在多个基准测试中达到了开源代码模型的最先进水平，还超越了 Codex 和 GPT-3.5 等现有闭源模型。此外，DeepSeek-Coder 模型采用宽松许可证，允许用于研究和无限制的商业用途。

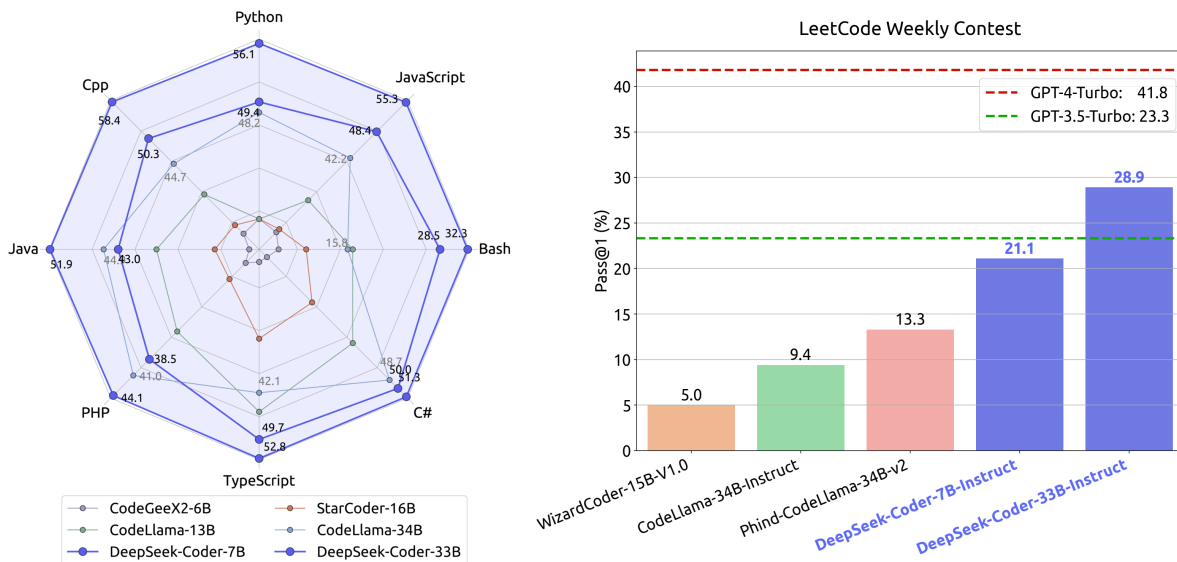


图 1 | DeepSeek-Coder 的性能表现

\* 核心贡献者，按姓名首字母排序。

# 1. 引言

大语言模型的迅猛发展极大地改变了软件开发领域 (OpenAI, 2023; Touvron et al., 2023), 开启了代码智能的新纪元。这些模型有望自动化并简化编程的诸多环节, 从漏洞检测到代码生成, 从而提升生产力并降低人为错误的概率。然而, 该领域面临的一个主要挑战是开源模型 (Li et al., 2023; Nijkamp et al., 2022; Roziere et al., 2023; Wang et al., 2021) 与闭源模型 (Gemini Team, 2023; OpenAI, 2023) 之间的性能差距。尽管大型闭源模型功能强大, 但由于其专有性质, 许多研究人员和开发者往往无法使用。

为应对这一挑战, 我们推出了 DeepSeek-Coder 系列。该系列包含一系列开源代码模型, 规模从 1.3B 到 33B 不等, 每个规模均提供基础版和指令微调版。系列中的每个模型均在源自 87 种编程语言的 2 万亿 token 上从头训练, 确保了对编程语言和语法的全面理解。此外, 我们尝试在仓库级别组织预训练数据, 以增强预训练模型在仓库内跨文件上下文中的理解能力。除了预训练期间采用下一个 token 预测损失外, 我们还引入了中间填充 (Fill-In-Middle, FIM) 方法 (Bavarian et al., 2022; Li et al., 2023)。该方法旨在进一步强化模型的代码补全能力。为满足处理更长代码输入的需求, 我们将上下文长度扩展至 16K。这一调整使我们的模型能够处理更复杂、更庞大的编程任务, 从而提升了其在各种编程场景中的通用性和适用性。

我们利用多种公开的代码相关基准测试进行了全面的实验。结果表明, 在开源模型中, DeepSeek-Coder-Base 33B 在所有基准测试中均持续展现出卓越的性能。此外, DeepSeek-Coder-Instruct 33B 在大多数评估基准上超越了 *OpenAI GPT-3.5 Turbo*, 显著缩小了 *OpenAI GPT-4* 与开源模型之间的性能差距。值得注意的是, 尽管参数量较少, DeepSeek-Coder-Base 7B 在与 CodeLlama-33B (Roziere et al., 2023) 等规模大五倍的模型相比时, 仍展现出极具竞争力的性能。综上所述, 我们的主要贡献如下:

- 我们推出了 DeepSeek-Coder-Base 和 DeepSeek-Coder-Instruct, 这是我们先进的专注于代码的大语言模型 (LLMs)。通过在海量代码语料库上进行广泛训练, 这些模型展现出对 87 种编程语言的精通理解能力。此外, 它们提供多种模型规模, 以满足广泛的计算和应用需求。
- 我们首次尝试在模型的预训练阶段引入仓库级数据构建。我们发现, 它能显著提升跨文件代码生成的能力。
- 我们的分析严格考察了 FIM 训练策略对代码模型预训练阶段的影响。这些全面研究的结果揭示了 FIM 配置中引人入胜的方面, 为代码预训练模型的改进与发展提供了宝贵的见解。
- 我们针对涵盖众多代码相关任务的一系列基准测试, 对我们的代码 LLM 进行了广泛评估。结果表明, DeepSeek-Coder-Base 在这些基准测试中超越了所有现有的开源代码 LLM。此外, 通过使用指令数据进行细致的微调, DeepSeek-Coder-Instruct 在代码相关任务中取得了优于 *OpenAI GPT-3.5 Turbo* 模型的性能。

## 2. 数据收集

DeepSeek-Coder 的训练数据集由 87% 的源代码、10% 的英文代码相关自然语言语料以及 3% 的与代码无关的中文自然语言语料组成。英文语料包含来自 GitHub Markdown 和 StackExchange<sup>1</sup> 的资料，旨在增强模型对代码相关概念的理解，并提升其处理库使用、漏洞修复等任务的能力。同时，中文语料由高质量文章构成，旨在提高模型对中文的理解能力。在本节中，我们将概述代码训练数据的构建过程。该过程包括数据爬取、基于规则的过滤、依赖解析、仓库级去重和质量筛选，如图 2 所示。下文将逐步介绍数据创建的具体流程。



图 2 | 数据集构建流程

### 2.1. GitHub 数据爬取与过滤

我们收集了 GitHub 上创建于 2023 年 2 月之前的公开仓库，并仅保留表 1 中列出的 87 种编程语言。为减少待处理的数据量，我们采用了与 StarCoder 项目 (Li et al., 2023) 相似的过滤规则，初步剔除低质量代码。应用这些过滤规则后，数据总量缩减至原始大小的 32.8%。为使本文内容完整，我们简要介绍 StarCoder Data 项目中使用的过滤规则：

首先，我们过滤掉平均行长度超过 100 个字符或最大行长度超过 1000 个字符的文件。此外，我们移除字母字符占比低于 25% 的文件。除 XSLT 编程语言外，我们进一步过滤掉前 100 个字符中包含字符串 "<?xml version=" 的文件。对于 HTML 文件，我们考虑可见文本与 HTML 代码的比例。我们保留可见文本占比至少为 20% 且不少于 100 个字符的文件。对于通常包含更多数据的 JSON 和 YAML 文件，我们仅保留字符数在 50 到 5000 之间的文件。此举有效剔除了大多数数据量庞大的文件。

### 2.2. 依赖解析

在先前的研究 (Chen et al., 2021; Li et al., 2023; Nijkamp et al., 2022; Roziere et al., 2023) 中，代码大语言模型主要在文件级源代码上进行预训练，这忽略了项目中不同文件之间的依赖关系。然而，在实际应用中，此类模型难以有效扩展以处理整个项目级的代码场景。因此，在本步骤中，我们将探讨如何利用同一仓库内文件之间的依赖关系。具体而言，我们首先解析文件间的依赖关系，然后对这些文件进行排序，以确保每个文件所依赖的上下文在输入序列中位于该文件之前。通过按照依赖关系对齐文件，我们的数据集能够更准确地反映真实的编码实践与结构。这种增强的对齐方式不仅使数据集更具相关性，还有望提升模型在处理项目级代码场景时的实用性和适用性。值得注意的是，我们仅考虑文件之间的调用关系，并使用正则表达式进行提取，例

<sup>1</sup><https://stackexchange.com>

---

**Algorithm 1** 依赖分析的拓扑排序

---

```
1: procedure TOPOLOGICALSORT(files)
2:   graphs  $\leftarrow$  {} ▷ 初始化空的邻接表
3:   inDegree  $\leftarrow$  {} ▷ 初始化用于存储入度的空字典
4:   for each file in files do
5:     graphs[file]  $\leftarrow$  []
6:     inDegree[file]  $\leftarrow$  0
7:   end for
8:
9:   for each fileA in files do
10:    for each fileB in files do
11:      if HASDEPENDENCY(fileA, fileB) then ▷ 如果 fileA 依赖于 fileB
12:        graphs[fileB].append(fileA) ▷ 添加从 B 到 A 的边
13:        inDegree[fileA]  $\leftarrow$  inDegree[fileA] + 1 ▷ 增加 A 的入度
14:      end if
15:    end for
16:  end for
17:
18:  subgraphs  $\leftarrow$  getDisconnectedSubgraphs(graphs) ▷ 识别不连通的子图
19:  allResults  $\leftarrow$  []
20:  for each subgraph in subgraphs do
21:    results  $\leftarrow$  []
22:    while length(results)  $\neq$  NumberOfNodes(subgraph) do
23:      file  $\leftarrow$  argmin({inDegree[file] | file  $\in$  subgraph and file  $\notin$  results})
24:      for each node in graphs[file] do
25:        inDegree[node]  $\leftarrow$  inDegree[node] - 1
26:      end for
27:      results.append(file)
28:    end while
29:    allResults.append(results)
30:  end for
31:
32:  return allResults
33: end procedure
```

---

如 Python 中的 "import"、C# 中的 "using" 以及 C 语言中的 "include"。

算法 1 描述了针对同一项目内文件列表的依赖分析拓扑排序。初始阶段，它设置了两个数据结构：一个名为 "graphs" 的空邻接表，用于表示文件间的依赖关系；以及一个名为 "inDegree" 的空字典，用于存储每个文件的入度。随后，算法遍历每一对文件以识别依赖关系，并相应地更新 "graphs" 和 "inDegree"。接下来，它识别整体依赖图中的任何不连通子图。对于每个子图，算法采用一种改进的拓扑排序方法。与选择入度为零节点的标准方法不同，该算法选择入度最小的节点，从而能够处理图中的环。选定的节点被添加到 "results" 列表中，并减少其相连节点的入度。此过程持续进行，直到为每个子图生成拓扑排序序列。算法最后返回这些排序序列的列表，并将每个序列中的文件拼接起来，形成单个训练样本。为了融入文件路径信息，我们在每个文件的开头添加了一条指示文件路径的注释。该方法确保了路径信息在训练数据中得到保留。

### 2.3. 仓库级去重

近期的研究表明，对大型语言模型（LLMs）的训练数据集进行去重可以带来显著的性能提升。Lee et al. (2022) 指出，语言模型的训练语料库中通常包含大量近似重复的内容，通过移除长重复子串可以提升 LLM 的性能。Kocetkov et al. (2022) 将近似去重方法应用于训练数据，取得了显著的性能改善，并强调近似去重是在代码基准任务上取得竞争性性能的关键预处理步骤。在我们的数据集中，我们也采用了近似去重。然而，我们的方法与以往工作有所不同。我们在代码的仓库级别进行去重，而不是在文件级别，因为后者可能会过滤掉仓库中的某些文件，从而破坏仓库的结构。具体而言，我们将仓库级别的拼接代码视为单个样本，并应用相同的近似去重算法，以确保仓库结构的完整性。

### 2.4. 质量筛选与去污染

除了应用第 2.1 节中提到的过滤规则外，我们还结合编译器、质量模型以及启发式规则，进一步过滤低质量数据。这包括包含语法错误、可读性差以及模块化程度低的代码。我们在表 1 中提供了源代码的统计摘要，共包含 87 种语言，详细列出了每种语言的磁盘大小、文件数量和占比。数据总量为 798 GB，包含 6.03 亿个文件。为确保我们的代码训练数据不受可能存在于 GitHub 上的测试集信息污染，我们实施了 n-gram 过滤流程。该流程涉及移除符合特定条件的任何代码片段。具体而言，我们过滤掉包含来自 HumanEval (Chen et al., 2021)、MBPP (Austin et al., 2021)、GSM8K (Cobbe et al., 2021) 和 MATH (Hendrycks et al., 2021) 等数据源的文档字符串、问题及解答的文件。对于过滤标准，我们应用以下规则：如果一段代码包含与测试数据中任何内容相同的 10-gram 字符串，则将其从训练数据中排除。对于测试数据中包含长度小于 10-gram 但不小于 3-gram 的字符串的情况，我们采用精确匹配方法进行过滤。

## 3. 训练策略

### 3.1. 训练策略

#### 3.1.1. 下一词元预测

我们模型的第一个训练目标被称为 下一词元预测 (next token prediction)。在此过程中，将各种文件拼接成固定长度的条目。随后，使用这些条目对模型进行训练，使其能够根据提供的上下文预测后续的词元。

#### 3.1.2. 中间填充

我们模型的第二个训练目标被称为 中间填充 (fill-in-the-middle)。在代码预训练场景中，通常需要根据给定的上下文和后续文本来生成相应的插入内容。由于编程语言中存在特定的依赖关系，仅依靠下一词元预测不足以学习这种中间填充能力。因此，一些方法 (Bavarian et al., 2022; Li et al., 2023) 提出了中间填充 (Fill-in-the-Middle, FIM) 的预训练方法。该方法涉及将文本随机划分为三个部分，然后打乱这些部分的顺序，并用特殊字符连接它们。此方法旨在训练过程中引

语言	大小 (GB)	文件数 (千)	占比 (%)	语言	大小 (GB)	文件数 (千)	占比 (%)
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	<b>总计</b>	<b>797.92</b>	<b>603,173</b>	<b>100.00</b>

表 1 | 所选编程语言清洗后训练数据的统计摘要。

入一个填空预训练任务。在 FIM 方法中，采用了两种不同的模式：PSM（前缀-后缀-中间）和 SPM（后缀-前缀-中间）。在 PSM 模式下，训练语料库按 *Prefix, Suffix, Middle* 的顺序组织，使中间段落在前缀和后缀之间对齐。相反，SPM 模式将段落排列为 *Suffix, Prefix, Middle*，提出了不同的结构挑战。这些模式对于增强模型处理代码中各种结构排列的能力至关重要，为高级代码预测任务提供了强大的训练框架。

为了确定 FIM 方法中各种超参数的有效性，我们进行了一系列消融实验。

**实验设置：**在本实验中，我们采用 DeepSeek-Coder-Base 1.3B 作为模型架构。为了简化实验流程，我们专注于训练数据集中的 Python 子集。我们的主要目标是评估中间填充（FIM）技术的

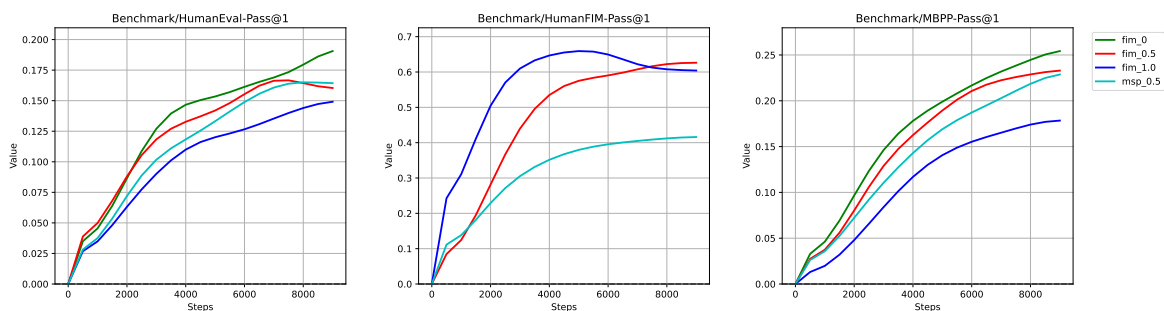


图 3 | 使用 FIM 目标的有效性。

有效性，使用了 HumanEval-FIM 基准测试 (Fried et al., 2022)。该基准测试专门针对 Python 的单个 FIM 任务，其中 HumanEval 解决方案中的一行代码会被随机遮蔽，以测试模型预测缺失代码行的能力。我们假设，与传统下一词元预测目标相比，PSM 模式可能会表现出细微的差异。这主要是因为 PSM 涉及重新排列原始文本的顺序，可能会影响模型的学习动态。因此，我们在四种不同的配置下为 FIM 实现了 PSM 模式：0% FIM 比例、50% FIM 比例、100% FIM 比例和 50% MSP 比例。掩码跨度预测 (Masked Span Prediction, MSP) 策略最初由 T5 (Raffel et al., 2023) 引入，它遮蔽多个文本跨度并训练模型重建这些片段。根据 CodeGen2.5 (Nijkamp et al., 2023) 的研究，与 PSM 相比，MSP 可能会提升 FIM 的性能。因此，我们将该方法纳入我们的对比分析中。

**结果：**我们的实验结果如图 3 所示。虽然模型在 100% FIM 比例下在 HumanEval-FIM 上表现出最佳性能，但该配置也导致代码补全能力最弱。这表明 FIM 能力与代码补全能力之间存在权衡。此外，我们观察到，在 50% PSM 比例下，模型的表现优于 MSP 策略。为了在 FIM 效率和代码补全能力之间取得平衡，我们最终选择 50% PSM 比例作为首选训练策略。

在我们的实现中，我们为此任务引入了三个哨兵词元 (sentinel tokens)。对于每个代码文件，我们首先将其内容划分为三个片段，分别表示为  $f_{pre}$ 、 $f_{middle}$  和  $f_{suf}$ 。使用 PSM 模式，我们按如下方式构建训练样本：

```
< | fim_start | >f_pre< | fim_hole | >f_suf< | fim_end | >f_middle<|eos_token|>
```

正如 Bavarian et al. (2022) 的原始工作所提出的，我们在打包过程之前在文档级别实现了中间填充 (FIM) 方法。该方法采用 PSM 模式，FIM 比例为 0.5。

### 3.2. 分词器

在分词过程中，我们使用 HuggingFace Tokenizer 库<sup>2</sup> 在我们训练语料库的一个子集上训练字节对编码 (Byte Pair Encoding, BPE) 分词器，如 Sennrich 等人 (2015) (Sennrich et al., 2015) 所述。最终，我们使用一个词表大小为 32,000 的分词器。

<sup>2</sup><https://github.com/huggingface/tokenizers>

### 3.3. 模型架构

我们开发了一系列参数规模不同的模型以满足多样化的应用需求，包括 1.3B、6.7B 和 33B 参数规模的模型。这些模型建立在 DeepSeek-AI (2024) 概述的 DeepSeek 大语言模型 (LLM) 相同的框架之上。每个模型都是一个仅解码器 (decoder-only) 的 Transformer，并采用了 Su et al. (2023) 描述的旋转位置编码 (Rotary Position Embedding, RoPE)。值得注意的是，DeepSeek 33B 模型集成了组查询注意力 (Grouped-Query-Attention, GQA)，组大小为 8，从而提升了训练和推理效率。此外，我们采用 FlashAttention v2 (Dao, 2023) 来加速注意力机制相关的计算。我们模型的架构细节总结于表 2。

### 3.4. 优化

遵循 DeepSeek LLM (DeepSeek-AI, 2024) 的做法，我们使用 AdamW (Loshchilov and Hutter, 2019) 作为优化器， $\beta_1$  和  $\beta_2$  的值分别为 0.9 和 0.95。我们根据 DeepSeek LLM 中建议的缩放定律调整批量大小和学习率。对于学习率调度，我们实施了三阶段策略，其中包括 2000 步预热，并将最终学习率设置为初始学习率的 10%。值得注意的是，遵循 DeepSeek LLM (DeepSeek-AI, 2024) 建立的指南，每个阶段的学习率均缩小至前一阶段学习率的  $\sqrt{\frac{1}{10}}$ 。

超参数	DeepSeek-Coder 1.3B	DeepSeek-Coder 6.7B	DeepSeek-Coder 33B
隐藏层激活函数	SwiGLU	SwiGLU	SwiGLU
隐藏层维度	2048	4096	7168
中间层维度	5504	11008	19200
隐藏层数量	24	32	62
注意力头数量	16	32	56
注意力机制	多头	多头	组查询 (8)
批量大小	1024	2304	3840
最大学习率	5.3e-4	4.2e-4	3.5e-4

表 2 | DeepSeek-Coder 的超参数。

### 3.5. 实验环境

我们的实验使用 HAI-LLM (High-Flyer, 2023) 框架进行，该框架以其在训练大语言模型时的高效性和轻量化著称。该框架集成了多种并行策略以优化计算效率，包括张量并行 (Korthikanti et al., 2023)，以及 ZeRO 数据并行 (Rajbhandari et al., 2020) 和 PipeDream 流水线并行 (Narayanan et al., 2019)。我们的实验使用了配备 NVIDIA A100 和 H800 GPU 的集群。在 A100 集群中，每个节点配置了 8 块 GPU，并通过 NVLink 桥接器两两互联。H800 集群的布局类似，每个节点同样包含 8 块 GPU。这些 GPU 通过 NVLink 和 NVSwitch 技术的组合进行互联，确保节点内的高效数据传输。为了促进 A100 和 H800 集群中节点之间的无缝通信，我们采用了以高吞吐量和低延迟著称的 InfiniBand 互连技术。该配置为我们的计算实验提供了强大且高效的基础设施。

### 3.6. 长上下文

为了增强 DeepSeek-Coder 处理扩展上下文的能力，特别是针对仓库级代码处理等场景，我们重新配置了 RoPE (Su et al., 2023) 参数以扩展默认上下文窗口。遵循先前的实践 (Chen et al., 2023; kaiokendev, 2023)，我们采用了线性缩放策略，将缩放因子从 1 增加到 4，并将基础频率从 10000 更改为 100000。模型进行了额外的 1000 步训练，批量大小为 512，序列长度为 16K。学习率保持与最终预训练阶段相同。理论上，这些修改使我们的模型能够处理高达 64K 词元的上下文。然而，经验观察表明，模型在 16K 词元范围内能提供最可靠的输出。未来的研究将继续完善和评估长上下文适应方法，旨在进一步提升 DeepSeek-Coder 在处理扩展上下文时的效率和易用性。

### 3.7. 指令微调

我们通过使用高质量数据进行基于指令的微调来增强 DeepSeek-Coder-Base，从而开发了 DeepSeek-Coder-Instruct。该数据包含有益且客观的人类指令，采用 Alpaca 指令格式 (Taori et al., 2023) 进行结构化。为了区分每个对话轮次，我们使用了一个独特的分隔符词元 `<|EOT|>` 来标记每个片段的结束。在训练过程中，我们使用余弦调度策略，包含 100 步预热，初始学习率为  $1e-5$ 。我们还使用了 4M 词元的批量大小，总训练词元数为 2B。

图 4 展示了使用 DeepSeek-Coder-Instruct 34B 的一个示例。该示例是一个构建贪吃蛇游戏的多轮对话场景。最初，我们要求模型使用 `pygame` 编写一个贪吃蛇游戏。模型成功创建了一个可以无错误运行的基础贪吃蛇游戏。为了改进游戏，我们进一步要求在左上角添加一个计分系统。随后，模型引入了一个 `"score"` 变量和一个 `"display_score"` 函数，并解释了如何集成这些功能。该示例展示了 DeepSeek-Coder-Instruct 在多轮对话设置中提供完整解决方案的能力。更多案例可在附录 A 中找到。

## 4. 实验结果

在本节中，我们在四项任务上对 DeepSeek-Coder 进行了评估，包括代码生成 (§4.1)、FIM 代码补全 (§4.2)、跨文件代码补全 (§4.3) 以及基于程序的数学推理 (§4.4)。我们将 DeepSeek-Coder 与以往最先进的语言模型进行了比较：

- **CodeGeeX2** (Zheng et al., 2023) 代表了多语言代码生成模型 CodeGeeX 的第二代。它基于 ChatGLM2 (Du et al., 2022) 架构开发，并通过大量代码示例数据集进行了增强。
- **StarCoder** (Li et al., 2023) 是一个公开可用的模型，参数量高达 150 亿。它专门在精心策划的 Stack 数据集 (Kocetkov et al., 2022) 子集上进行训练，涵盖 86 种编程语言，确保其在广泛编程任务中的专业能力。
- **CodeLlama** (Roziere et al., 2023) 包含一系列以代码为中心的大型语言模型 (LLM)，它们是 LLaMA2 (Touvron et al., 2023) 的衍生模型。这些模型提供三种规模——7B、13B 和 34B——在庞大的 5000 亿词元代码语料库上进行了持续训练，建立在基础的 LLaMA2 架构之上。

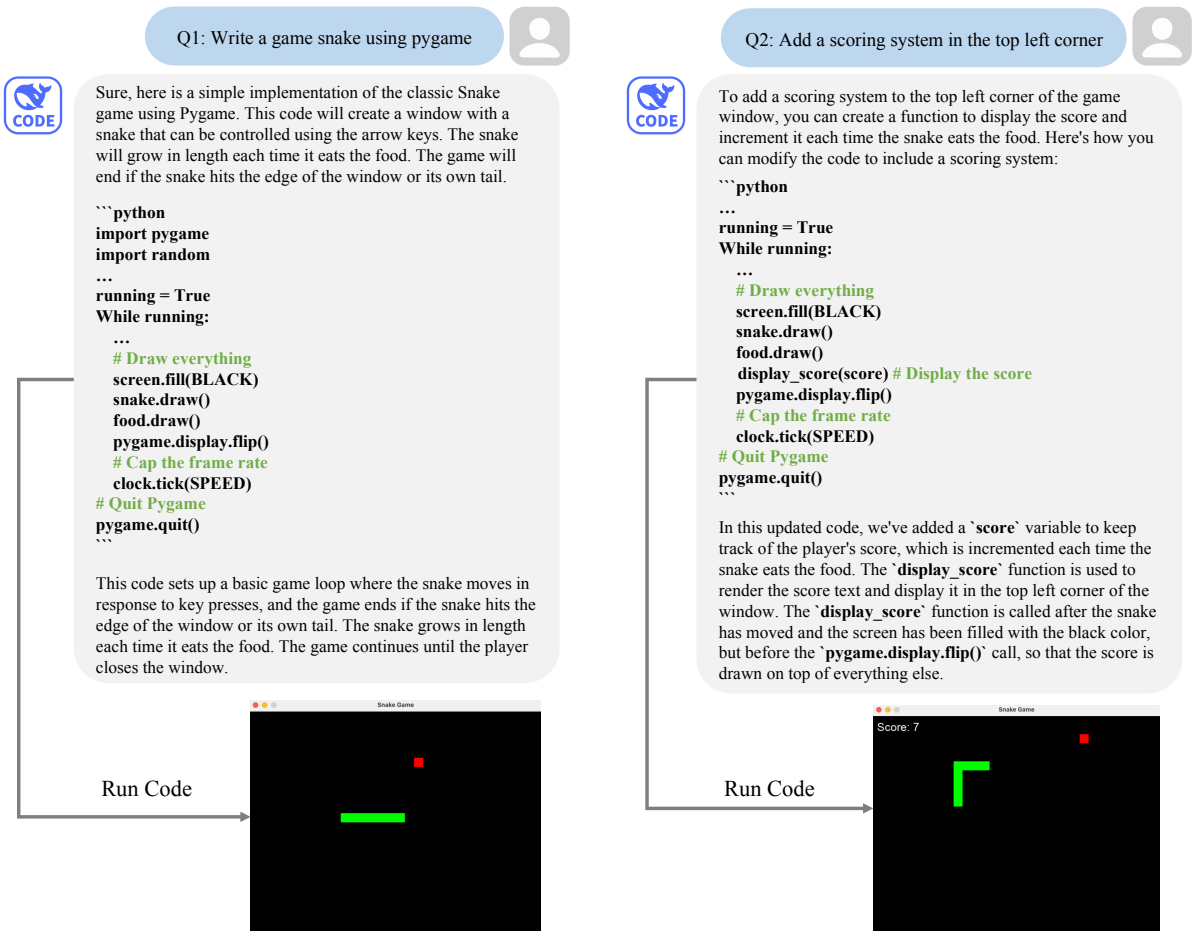


图 4 | DeepSeek-Coder-Instruct 33B 在多轮设置下的回复示例。

- **code-cushman-001** Chen et al. (2021) 是由 OpenAI 开发的 120 亿参数模型，曾作为 GitHub Copilot 的初始模型。
- **GPT-3.5 和 GPT-4** (OpenAI, 2023) 是由 OpenAI 开发的先进生成式 AI 模型。尽管它们并非专门针对代码生成进行训练，但在该领域也展现出了显著的性能。它们在处理代码生成任务方面的有效性主要归功于其庞大的参数量。

#### 4.1. 代码生成

**HumanEval 和 MBPP 基准** HumanEval (Chen et al., 2021) 和 MBPP (Austin et al., 2021) 基准被广泛用于评估代码大语言模型 (LLM)。HumanEval 包含 164 个手工编写的 Python 问题，通过测试用例验证，用于在零样本 (zero-shot) 设置下评估代码 LLM 生成的代码；而 MBPP 基准则包含 500 个问题，采用少样本 (few-shot) 设置。为了评估模型的多语言能力，我们将 HumanEval 基准中的 Python 问题扩展到了另外七种常用的编程语言，即 C++、Java、PHP、TypeScript (TS)、C#、Bash 和 JavaScript (JS) (Cassano et al., 2023)。在这两个基准测试中，我们采用了贪婪搜索策略，并使用相同的脚本和环境重新实现了基线结果，以确保公平比较。

结果如表 3 所示。可以看出，DeepSeek-Coder-Base 在 HumanEval 和 MBPP 上分别取得

模型	规模	Python	C++	Java	PHP	TS	C#	Bash	JS	平均	MBPP
多语言基础模型											
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	<b>56.1%</b>	<b>58.4%</b>	<b>51.9%</b>	<b>44.1%</b>	<b>52.8%</b>	<b>51.3%</b>	<b>32.3%</b>	<b>55.3%</b>	<b>50.3%</b>	<b>66.0%</b>
指令微调模型											
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	-	<b>84.1%</b>	<b>76.4%</b>	<b>81.6%</b>	<b>77.2%</b>	<b>77.4%</b>	<b>79.1%</b>	<b>58.2%</b>	<b>78.0%</b>	<b>76.5%</b>	<b>80.0%</b>
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	<b>79.3%</b>	<b>68.9%</b>	<b>73.4%</b>	<b>72.7%</b>	<b>67.9%</b>	<b>74.1%</b>	<b>43.0%</b>	<b>73.9%</b>	<b>69.2%</b>	<b>70.0%</b>

表 3 | 各方法在多语言 HumanEval 和 MBPP 基准上的性能表现。

了 50.3% 和 66.0% 的平均准确率，达到了当前最佳性能。与参数量相近的开源模型 CodeLlama-Base 34B 相比，我们的模型在准确率上分别提升了 9% 和 11%。值得注意的是，即使是我们较小的 DeepSeek-Coder-Base 6.7B 模型，其性能也超越了 CodeLlama-Base 34B。经过指令微调后，我们的模型在 HumanEval 基准测试中超越了闭源模型 GPT-3.5-Turbo，显著缩小了 OpenAI GPT-4 与开源模型之间的性能差距。

**DS-1000 基准测试** HumanEval 和 MBPP 存在一个显著缺陷，即它们严重依赖于简单的编程任务，可能无法准确反映大多数程序员日常编写的代码类型。相比之下，Lai et al. (2023) 中引入的 DS-1000 基准测试提供了涵盖七个不同库的 1,000 个实用且贴近现实的数据科学工作流。该基准通过针对特定测试用例执行代码来评估代码生成能力。DS-1000 的独特之处在于它根据涉及的库对问题进行了分类，涵盖 Matplotlib、NumPy、Pandas、SciPy、Scikit-Learn、PyTorch 和 TensorFlow。该基准评估了基础模型在代码补全设置下的性能，我们提供了每个库的 pass@1 结果以及总体得分。

DS-1000 基准测试的结果如表 4 所示。从表中可以看出，DeepSeek-Coder 模型在所有库上均取得了较高的准确率，这表明我们的模型不仅能够生成高质量的代码，还能在实际的数据科学工作流中更准确地使用各类库。

模型	参数量	Matplotlib	Numpy	Pandas	Pytorch	Scipy	Scikit-Learn	Tensorflow	平均
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	<b>56.1%</b>	<b>49.6%</b>	<b>25.8%</b>	<b>36.8%</b>	<b>36.8%</b>	<b>40.0%</b>	<b>46.7%</b>	<b>40.2%</b>

表 4 | 不同方法在 DS-1000 任务上的性能表现。

**LeetCode 竞赛基准测试** 为了进一步验证模型在真实编程问题上的能力，我们构建了 LeetCode 竞赛基准测试<sup>3</sup>。LeetCode<sup>4</sup> 提供竞赛级别的问题，带来了巨大的挑战，用于测试模型的问题理解能力和代码生成技能。我们收集了 LeetCode 竞赛中的最新题目，以防止题目或其解决方案出现在我们的预训练数据中。我们从 2023 年 7 月到 2024 年 1 月共收集了 180 道题目。对于每道题目，我们收集了 100 个测试用例以确保测试覆盖率。我们使用模板 “{problem\_description}\n 请完成以下代码以解决上述问题：\n```\npython\n{code\_template}\n```” 来构建指令提示。

评估结果如表 5 所示。在我们的评估中，DeepSeek-Coder 模型相较于当前的开源代码模型展现出卓越的性能。具体而言，DeepSeek-Coder-Instruct 6.7B 和 33B 在该基准测试中分别取得了 19.4% 和 27.8% 的 Pass@1 得分。这一性能显著超越了 CodeLlama-33B 等现有开源模型。DeepSeek-Coder-Instruct 33B 是唯一在该任务中超越 OpenAI GPT-3.5-Turbo 的开源模型。然而，与更先进的 GPT-4-Turbo 相比，仍存在较大的性能差距。

模型	参数量	简单 (45)	中等 (91)	困难 (44)	总计 (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	-	<b>73.3%</b>	<b>31.9%</b>	<b>25.0%</b>	<b>40.6%</b>
GPT-4-Turbo + CoT	-	71.1%	<b>35.2%</b>	<b>25.0%</b>	<b>41.8%</b>
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	<b>57.8%</b>	22.0%	9.1%	27.8%
DeepSeek-Coder-Instruct + CoT	33B	53.3%	<b>25.3%</b>	<b>11.4%</b>	<b>28.9%</b>

表 5 | 不同模型在 LeetCode 竞赛基准测试上的性能表现。

我们的分析表明，引入思维链 (Chain-of-Thought, CoT) 提示显著提升了 DeepSeek-Coder-Instruct 模型的能力。这种提升在更具挑战性的任务子集中尤为明显。通过在初始提示后添加指令“你需要先编写逐步大纲，然后再编写代码”，我们观察到了性能的提升。这一观察使我们认为，首先编写详细的代码描述有助于模型更有效地理解和解决编程任务中逻辑与依赖关系的复杂性，尤其是高复杂度的任务。因此，我们强烈建议在利用 DeepSeek-Coder-Instruct 模型应对复杂编程挑战时采用 CoT 提示策略。这种方法促进了更系统、更逻辑化的问题解决框架，有望在代码生成任务中带来更精确和高效的结果。

需要指出的是，尽管我们竭尽全力收集最新的编程题目用于模型测试，但仍无法完全排除数据污染的可能性。我们观察到，GPT-4-Turbo 和 DeepSeek-Coder 模型在七月和八月举行的 LeetCode 竞赛中取得了更高的分数。我们鼓励研究社区在未来使用我们发布的 LeetCode 数据

<sup>3</sup>我们已在 <https://github.com/deepseek-ai/DeepSeek-Coder/tree/main/Evaluation/LeetCode> 发布了该基准测试。

<sup>4</sup><https://leetcode.com/>

进行模型评估时，充分考虑数据污染的潜在问题。

## 4.2. 中间填充代码补全

DeepSeek-Coder 模型在预训练阶段采用了 0.5 的 FIM (Fill-In-the-Middle, 中间填充) 训练比例。这种专门的训练策略使模型能够熟练地根据给定代码片段的前后文 (前缀和后缀) 上下文来填充空白并生成代码。该能力在代码补全工具领域具有显著优势。目前已有多个具备类似能力的开源模型涌现, 其中较为著名的包括 SantaCoder (Allal et al., 2023)、StarCoder (Li et al., 2023) 和 CodeLlama (Roziere et al., 2023)。这些模型在代码生成与补全领域树立了标杆。在评估 DeepSeek-Coder 模型的性能时, 我们与上述模型进行了对比分析。本次对比采用的基准是 Allal et al. (2023) 提出的单行填充 (Single-Line Infilling) 基准测试, 涵盖三种不同的编程语言。该基准测试以行级精确匹配准确率作为评估指标。

模型	规模	Python	Java	JavaScript	均值
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	<b>68.3%</b>	77.6%	80.7%	75.5%
DeepSeek-Coder-Base	1B	57.4%	82.2%	71.7%	70.4%
DeepSeek-Coder-Base	7B	66.6%	<b>88.1%</b>	79.7%	80.7%
DeepSeek-Coder-Base	33B	65.4%	86.6%	<b>82.5%</b>	<b>81.2%</b>

表 6 | 不同方法在 FIM 任务上的性能表现。

评估结果如表 6 所示。尽管 DeepSeek-Coder 是参数量仅为 13 亿的最小模型, 但在此基准测试中仍优于规模更大的 StarCoder 和 CodeLlama。这种卓越的性能可归因于 DeepSeek-Coder 所用预训练数据的高质量。此外, 一个显著的趋势是模型规模与其性能之间存在正相关关系。随着模型规模的增加, 性能也随之稳步提升。这一趋势凸显了模型容量在实现更高代码补全准确率方面的重要性。基于这些发现, 我们建议在代码补全工具中部署 DeepSeek-Coder-Base 6.7B 模型。该建议基于该模型在效率与准确性之间展现出的良好平衡。DeepSeek-Coder-Base 6.7B 模型凭借其庞大的参数量, 在代码补全场景中已被证明具有极高的有效性, 是将其先进计算能力集成到编程环境中的理想选择。

## 4.3. 跨文件代码补全

在本节中, 我们将评估现有开源模型在跨文件代码补全任务上的性能表现。与上一节讨论的代码生成不同, 跨文件代码补全要求模型能够访问并理解跨越多个文件且包含大量跨文件依赖关系的代码仓库。我们使用 CrossCodeEval (Ding et al., 2023) 来评估当前可用的 7B 规模开源代码模型在跨文件补全任务中的能力。该数据集基于四种流行编程语言 (Python、Java、TypeScript 和 C#) 中多样化的真实世界、开源且采用宽松许可证的代码仓库构建而成。该数据集专门设计为严格依赖跨文件上下文才能实现准确补全。值得注意的是, 该数据集构建于 2023 年 3 月至 6 月期间创建的代码仓库, 而我们的预训练数据仅包含 2023 年 2 月之前创建的代码, 这确保了该数据集未出现在我们的预训练数据中, 从而避免了数据泄露。

模型	规模	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%
+ 检索		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%
+ 检索		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%
+ 检索		13.02%	64.30%	16.41%	<b>64.64%</b>	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%
+ 检索		<b>16.14%</b>	<b>66.51%</b>	<b>17.72%</b>	63.18%	<b>14.03%</b>	<b>61.77%</b>	<b>16.23%</b>	<b>63.42%</b>
+ 检索 (无仓库级预训练)		16.02%	<b>66.65%</b>	16.64%	61.88%	13.23%	60.92%	14.48%	62.38%

表 7 | 不同模型在跨文件代码补全任务上的性能表现。

在对各种模型的评估中，我们将最大序列长度设置为 2048 个 token，最大输出长度设置为 50 个 token，并将跨文件上下文的限制设置为 512 个 token。对于跨文件上下文，我们使用了 Ding et al. (2023) 提供的官方 BM25 搜索结果。评估指标包括精确匹配 (Exact Match) 和编辑相似度 (Edit Similarity)。如表 7 所示的结果表明，DeepSeek-Coder 在多种语言的跨文件补全任务中始终优于其他模型，展现了其卓越的实用应用能力。当仅使用文件级代码语料库 (**w/o Repo Pre-training**) 对 DeepSeek-Coder 进行预训练时，我们观察到其在 Java、TypeScript 和 C# 语言上的性能有所下降，这表明了仓库级预训练的有效性。

#### 4.4. 基于程序的数学推理

基于程序的数学推理旨在评估模型通过编程理解和解决数学问题的能力。此类推理在数据分析和科学计算等领域至关重要。为进行此项评估，我们采用了 Gao et al. (2023) 中提出的程序辅助数学推理 (PAL) 方法。该方法应用于七个不同的基准测试，每个基准测试都具有独特的挑战性和应用场景。这些基准测试包括 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) 和 MAWPS (Gou et al., 2023)。在这些基准测试中，模型被提示交替使用自然语言描述解题步骤，随后通过代码执行该步骤。如表 8 所示，DeepSeek-Coder 模型在所有基准测试中均取得了卓越的性能，尤其是 33B 参数版本，这展示了此类模型在需要复杂数学计算和解决问题能力的应用中的巨大潜力。

模型	规模	GSM8k	MATH	GSM-Hard	SVAMP	TabMWP	ASDiv	MAWPS	平均
多语言基础模型									
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	<b>60.7%</b>	<b>29.1%</b>	<b>54.1%</b>	<b>71.6%</b>	<b>75.3%</b>	<b>76.7%</b>	<b>93.3%</b>	<b>65.8%</b>

表 8 | 不同方法在程序辅助数学推理任务上的性能表现。

## 5. 从通用大语言模型进行继续预训练

为了进一步提升 DeepSeek-Coder 模型的自然语言理解和数学推理能力，我们在通用语言模型 DeepSeek-LLM-7B Base (DeepSeek-AI, 2024) 的基础上，使用 2 万亿 tokens 的数据进行了额外的预训练，从而得到了 DeepSeek-Coder-v1.5 7B。在此次预训练中，我们专门使用了表 9 中列出的数据源。与 DeepSeek-Coder 不同，DeepSeek-Coder-v1.5 在预训练阶段仅采用下一个 token 预测目标，且上下文长度为 4K。

数据源	占比
源代码	70%
Markdown 与 StackExchange	10%
与代码相关的自然语言	7%
与数学相关的自然语言	7%
双语（中英）自然语言	6%

表 9 | DeepSeek-Coder-v1.5 7B 预训练的数据源

我们对 DeepSeek-Coder-v1.5 7B 与 DeepSeek-Coder 6.7B 进行了对比，并使用我们的评估流水线重新运行了所有基准测试，以确保对比的公平性。我们在广泛的任务上评估了模型性能，这些任务可分为以下几类：

- **编程**：该类别包括使用 Chen et al. (2021) 的 HumanEval 数据集进行的多语言设置评估，以及使用 Austin et al. (2021) 的 MBPP 数据集进行的 Python 设置评估。
- **数学推理**：我们使用 GSM8K 基准测试 (Cobbe et al., 2021) 和 MATH 基准测试 (Hendrycks et al., 2021) [4] 来评估模型在数学推理任务上的性能。这些任务涉及通过生成程序来解决数学问题。
- **自然语言**：我们在自然语言任务上的评估包括 MMLU (Hendrycks et al., 2020)、BBH (Suzgun et al., 2022)、HellaSwag (Zellers et al., 2019)、Winogrande (Sakaguchi et al., 2021) 以及 ARC-Challenge (Clark et al., 2018) 基准测试。

Base 和 Instruct 模型的结果如表 10 所示。观察发现，尽管 DeepSeek-Coder-Base-v1.5 模型在编程性能上略有下降，但与 DeepSeek-Coder-Base 模型相比，它在大多数任务上均表现出显著提升。特别是在数学推理和自然语言类别中，DeepSeek-Coder-Base-v1.5 在所有基准测试上均显著优于其前身，这也证明了其在数学推理和自然语言处理能力方面取得了显著进步。

## 6. 结论

在本技术报告中，我们介绍了一系列专为编程设计的专用大型语言模型 (LLMs)，命名为 DeepSeek-Coder，提供三种不同的规模：1.3B、6.7B 和 33B 参数。这些模型在精心策划的项目级代码语料库上进行了独特训练，采用“填空”预训练目标以增强代码补全能力。一项重大进展是将模型的上下文窗口扩展至 16,384 个 token，从而极大地提高了其在处理大规模代码生成任务时的有

模型	参数量	编程		数学推理		自然语言				
		HumanEval	MBPP	GSM8K	MATH	MMLU	BBH	HellaSwag	WinoG	ARC-C
DeepSeek-Coder-Base	6.7B	<b>44.7%</b>	<b>60.6%</b>	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%	<b>62.4%</b>	<b>24.7%</b>	<b>49.1%</b>	<b>55.2%</b>	<b>69.9%</b>	<b>63.8%</b>	<b>47.2%</b>
DeepSeek-Coder-Instruct	6.7B	<b>66.1%</b>	<b>65.4%</b>	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%	<b>72.6%</b>	<b>34.1%</b>	<b>49.5%</b>	<b>53.3%</b>	<b>72.2%</b>	<b>63.4%</b>	<b>48.1%</b>

表 10 | DeepSeek-Coder-Base 与 DeepSeek-Coder-Base-v1.5 的性能对比分析。数学任务通过编程方式求解。

效性。我们的评估表明，该系列中最先进的模型 DeepSeek-Coder-Base 33B 在各种标准测试中均超越了现有的开源代码模型。令人印象深刻的是，尽管规模较小，DeepSeek-Coder-Base 6.7B 模型的表现仍与 34B 参数的 CodeLlama 相当，这充分证明了我们预训练语料库的高质量。

为了增强 DeepSeek-Coder-Base 模型的零样本指令遵循能力，我们使用高质量的指令数据对其进行了微调。这使得 DeepSeek-Coder-Instruct 33B 模型在一系列编程相关任务中超越了 OpenAI 的 GPT-3.5 Turbo，展现了其在代码生成和理解方面的卓越能力。

为了进一步提升 DeepSeek-Coder-Base 模型的自然语言理解能力，我们基于 DeepSeek-LLM 7B 检查点进行了额外的预训练。该额外训练处理了包含 20 亿个 token 的多样化数据集，涵盖自然语言、代码和数学数据。由此诞生了全新且改进的代码模型 DeepSeek-Coder-v1.5。我们的观察表明，DeepSeek-Coder-v1.5 不仅保持了其前身的高水平编程性能，还展现出增强的自然语言理解能力。这一进展印证了我们的观点：最有效的编程专用大型语言模型 (LLMs) 是建立在强大的通用 LLM 基础之上的。原因显而易见：为了有效解释和执行编程任务，这些模型还必须具备对人类指令的深刻理解，而这些指令通常以各种自然语言形式呈现。展望未来，我们致力于开发并公开分享基于更大规模通用 LLM 的更强大的编程专用 LLM。

## 致谢

我们衷心感谢 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 和 Zhewen Hao 在训练 DeepSeek-Coder 模型期间提供的宝贵讨论与协助。

## 参考文献

- 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](https://arxiv.org/abs/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.
- Y. Ding, Z. Wang, W. U. Ahmad, H. Ding, M. Tan, N. Jain, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth, et al. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In [Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track](#), 2023.
- Z. Du, Y. Qian, X. Liu, M. Ding, J. Qiu, Z. Yang, and J. Tang. Glm: General language model pretraining with autoregressive blank infilling. In [Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics \(Volume 1: Long Papers\)](#), pages 320–335, 2022.
- D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis. Incoder: A generative model for code infilling and synthesis. [arXiv preprint arXiv:2204.05999](#), 2022.
- L. Gao, A. Madaan, S. Zhou, U. Alon, P. Liu, Y. Yang, J. Callan, and G. Neubig. Pal: Program-aided language models. In [International Conference on Machine Learning](#), pages 10764–10799. PMLR, 2023.

- G. Gemini Team. Gemini: A family of highly capable multimodal models, 2023. URL <https://google.com/gemini-paper>.
- Z. Gou, Z. Shao, Y. Gong, Y. Yang, M. Huang, N. Duan, W. Chen, et al. Tora: A tool-integrated reasoning agent for mathematical problem solving. [arXiv preprint arXiv:2309.17452](https://arxiv.org/abs/2309.17452), 2023.
- D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt. Measuring massive multitask language understanding. In [International Conference on Learning Representations](#), 2020.
- D. Hendrycks, C. Burns, S. Kadavath, A. Arora, S. Basart, E. Tang, D. Song, and J. Steinhardt. Measuring mathematical problem solving with the math dataset. [arXiv preprint arXiv:2103.03874](https://arxiv.org/abs/2103.03874), 2021.
- High-Flyer. Hai-llm: An efficient and lightweight tool for training large models. 2023. URL <https://www.high-flyer.cn/en/blog/hai-llm>.
- kaiokendev. Things i’m learning while training superhot. <https://kaiokendev.github.io/tutorials/ai#extending-context-to-8k>, 2023.
- D. Kocetkov, R. Li, L. Jia, C. Mou, Y. Jernite, M. Mitchell, C. M. Ferrandis, S. Hughes, T. Wolf, D. Bahdanau, et al. The stack: 3 tb of permissively licensed source code. [Transactions on Machine Learning Research](#), 2022.
- V. A. Korthikanti, J. Casper, S. Lym, L. McAfee, M. Andersch, M. Shoeybi, and B. Catanzaro. Reducing activation recomputation in large transformer models. [Proceedings of Machine Learning and Systems](#), 5, 2023.
- Y. Lai, C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, W.-t. Yih, D. Fried, S. Wang, and T. Yu. Ds-1000: A natural and reliable benchmark for data science code generation. In [International Conference on Machine Learning](#), pages 18319–18345. PMLR, 2023.
- K. Lee, D. Ippolito, A. Nystrom, C. Zhang, D. Eck, C. Callison-Burch, and N. Carlini. Duplicating training data makes language models better. In [Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics \(Volume 1: Long Papers\)](#), pages 8424–8445, 2022.
- R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, et al. Starcoder: may the source be with you! [arXiv preprint arXiv:2305.06161](https://arxiv.org/abs/2305.06161), 2023.
- I. Loshchilov and F. Hutter. Decoupled weight decay regularization, 2019.

- P. Lu, L. Qiu, K.-W. Chang, Y. N. Wu, S.-C. Zhu, T. Rajpurohit, P. Clark, and A. Kalyan. Dynamic prompt learning via policy gradient for semi-structured mathematical reasoning. In The Eleventh International Conference on Learning Representations, 2022.
- S.-Y. Miao, C.-C. Liang, and K.-Y. Su. A diverse corpus for evaluating and developing english math word problem solvers. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pages 975–984, 2020.
- D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In Proceedings of the 27th ACM Symposium on Operating Systems Principles, pages 1–15, 2019.
- E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong. Codegen: An open large language model for code with multi-turn program synthesis. arXiv preprint arXiv:2203.13474, 2022.
- E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou. Codegen2: Lessons for training llms on programming and natural languages, 2023.
- OpenAI. Gpt-4 technical report, 2023.
- A. Patel, S. Bhattamishra, and N. Goyal. Are nlp models really able to solve simple math word problems? In Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pages 2080–2094, 2021.
- C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2023.
- S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. Zero: Memory optimizations toward training trillion parameter models. In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–16. IEEE, 2020.
- B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Re-  
mez, J. Rapin, et al. Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950, 2023.
- K. Sakaguchi, R. L. Bras, C. Bhagavatula, and Y. Choi. Winogrande: An adversarial winograd schema challenge at scale. Communications of the ACM, 64(9):99–106, 2021.
- R. Sennrich, B. Haddow, and A. Birch. Neural machine translation of rare words with subword units. arXiv preprint arXiv:1508.07909, 2015.
- J. Su, Y. Lu, S. Pan, A. Murtadha, B. Wen, and Y. Liu. Roformer: Enhanced transformer with rotary position embedding, 2023.

- M. Suzgun, N. Scales, N. Schärli, S. Gehrmann, Y. Tay, H. W. Chung, A. Chowdhery, Q. V. Le, E. H. Chi, D. Zhou, , and J. Wei. Challenging big-bench tasks and whether chain-of-thought can solve them. arXiv preprint arXiv:2210.09261, 2022.
- R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto. Stanford alpaca: An instruction-following llama model. [https://github.com/tatsu-lab/stanford\\_alpaca](https://github.com/tatsu-lab/stanford_alpaca), 2023.
- H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. arXiv preprint arXiv:2307.09288, 2023.
- Y. Wang, W. Wang, S. Joty, and S. C. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859, 2021.
- R. Zellers, A. Holtzman, Y. Bisk, A. Farhadi, and Y. Choi. Hellaswag: Can a machine really finish your sentence? In Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, pages 4791–4800, 2019.
- Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, L. Shen, Z. Wang, A. Wang, Y. Li, et al. 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, pages 5673–5684, 2023.

## A. 与 DeepSeek-Coder-Instruct 对话的案例

我们将展示两个与 DeepSeek-Coder-Instruct 交互的案例，其中一个涉及关于创建数据库和执行数据分析的多轮对话，另一个则围绕使用模型解决 LeetCode 上的示例问题展开。

在第一个场景中（如图 5 所示），我们指示模型使用 Python 构建一个学生数据库，并随机插入 10 条信息。随后，在第二轮对话中，我们继续要求模型分析学生的年龄分布。从图 5 中可以明显看出，该模型能够生成无错误且全面的代码，并附带详细的解释说明。在第二个场景中（如图 6 所示），我们通过测试一个分布外的 LeetCode 竞赛问题来进一步评估模型的能力。该问题发布于 2023 年 11 月，晚于我们的数据收集时间，因此不属于模型的训练数据。结果表明，我们的模型在解决超出其训练分布的问题方面表现出色。

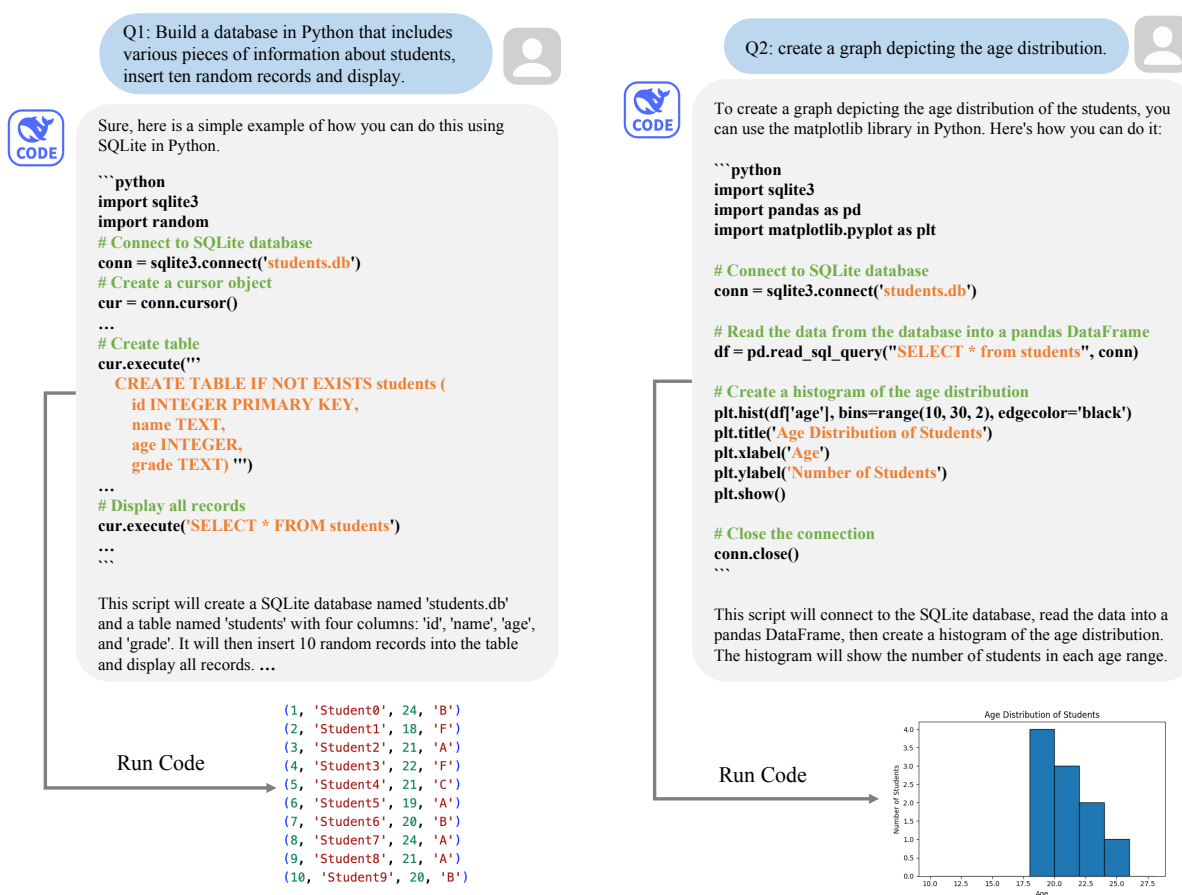


图 5 | 构建数据库与数据分析的示例。



**Q:** There are  $n$  teams numbered from 0 to  $n - 1$  in a tournament; each team is also a node in a DAG.

You are given the integer  $n$  and a 0-indexed 2D integer array `edges` of length  $m$  representing the DAG, where `edges[i] = [ui, vi]` indicates that there is a directed edge from team  $u_i$  to team  $v_i$  in the graph.

A directed edge from  $a$  to  $b$  in the graph means that team  $a$  is **stronger** than team  $b$  and team  $b$  is **weaker** than team  $a$ .

Team  $a$  will be the **champion** of the tournament if there is no team  $b$  that is **stronger** than team  $a$ .

Return the team that will be the **champion** of the tournament if there is a **unique** champion, otherwise, return  $-1$ .

**Notes**

• A **cycle** is a series of nodes  $a_1, a_2, \dots, a_n, a_{n+1}$  such that node  $a_1$  is the same node as node  $a_{n+1}$ , the nodes  $a_1, a_2, \dots, a_n$  are distinct, and there is a directed edge from the node  $a_i$  to node  $a_{i+1}$  for every  $i$  in the range  $[1, n]$ .

• A DAG is a directed graph that does not have any cycle.

• **Example 1:**

• **Example 2:**

•••

**Constraints:**

•••

Please follow the template below to solve the problem:

```

python
class Solution:
    def findChampion(self, n: int, edges: List[List[int]]) -> int:
    ...
  
```

### 2924. Find Champion II

Medium Topics Companies Hint



To solve this problem, we can use a topological sorting algorithm. The idea is to ...

```

python
from typing import List

class Solution:
    def findChampion(self, n: int, edges: List[List[int]]) -> int:
        # Create an adjacency list to represent the DAG
        adj_list = {}
        for i in range(n):
            adj_list[i] = []

        # Initialize a list to keep track of the number of ...
        in_degree = [0] * n

        # Calculate the in-degree for each team
        for u, v in edges:
            adj_list[u].append(v)
            in_degree[v] += 1

        # Initialize a list to keep track of the teams with no incoming edges
        no_incoming_edges = [i for i in range(n) if in_degree[i] == 0]

        # If there is more than one team with no incoming edges, there is no unique
        # champion
        if len(no_incoming_edges) != 1:
            return -1

        # Return the team with no incoming edges, which is the champion
        return no_incoming_edges[0]
  
```

This code first creates an adjacency list to represent the DAG. Then, ...

Submit Code



图 6 | 解决 LeetCode 问题的示例。

## B. DeepSeek-Coder-Base 训练期间的基准测试曲线

在图 7 中，我们展示了 DeepSeek-Coder-Base 模型在训练阶段的基准测试曲线。为了进行验证，我们使用了一个精心挑选的训练语料库子集，该子集包含 8,000 个代码文件。该子集经过刻意选择，以确保样本的多样性和代表性，这对于准确评估模型的能力至关重要。这些模型的性能指标在图 7 的最后两个子图中进行了详细说明，直观地展示了它们在整个训练过程中的有效性。

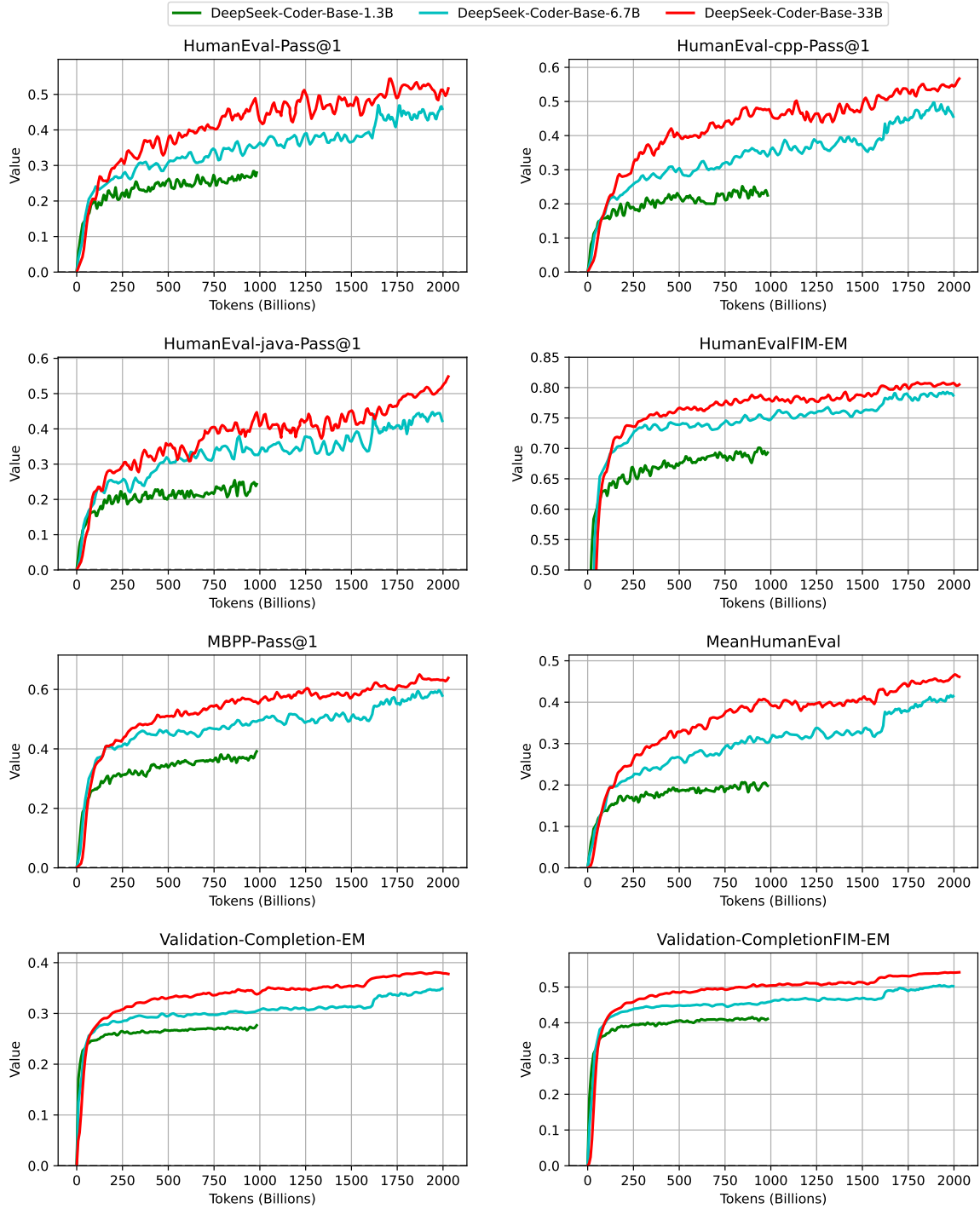


图 7 | DeepSeek-Coder-Base 训练期间的基准测试曲线。