Evaluation of LLMs on Syntax-Aware Code Fill-in-the-Middle Tasks

Linyuan Gong [1] **Sida Wang** [2] **Mostafa Elhoushi** [2] **Alvin Cheung** [1]

Abstract

We introduce **S**yntax-**A**ware **F**ill-**i**n-the-**M**iddle (SAFIM), a new benchmark for evaluating Large Language Models (LLMs) on the code Fill-in-the-Middle (FIM) task. This benchmark focuses on syntax-aware completions of program structures such as code blocks and conditional expressions, and includes 17,720 examples from multiple programming languages, sourced from recent code submissions after April 2022 to minimize data contamination. SAFIM provides a robust framework with various prompt designs and novel syntax-aware post-processing techniques, facilitating accurate and fair comparisons across LLMs. Our comprehensive evaluation of 15 LLMs shows that FIM pretraining not only enhances FIM proficiency but also improves Left-to-Right (L2R) inference using LLMs. Our findings challenge conventional beliefs and suggest that pretraining methods and data quality have more impact than model size. SAFIM thus serves as a foundational platform for future research in effective pretraining strategies for code LLMs. The evaluation toolkit and dataset are available at `https://github.com/gonglinyuan/safim`, and the leaderboard is available at `https://safimbenchmark.com`.

1. Introduction

Recent advances in Large Language Models (LLMs) such as GPT-3.5 (Ouyang et al., 2022), GPT-4 (OpenAI, 2023), and CodeLLaMa (Rozière et al., 2023) have revolutionized coding-related tasks. However, existing benchmarks like HumanEval (Chen et al., 2021a) and MBPP (Austin et al., 2021) focus on generating standalone functions or single-file code from natural language descriptions, and do not consider the more common practice of modifying and expanding existing code during development.

Recognizing this gap, we introduce the **S**yntax-**A**ware **F**ill-**i**n-the-**M**iddle (SAFIM) benchmark. SAFIM emphasizes syntax-aware completion within code's Abstract Syntax Tree (AST), targeting algorithmic blocks, control-flow expressions, and API function calls, unlike existing Fill-in-the Middle (FIM) benchmarks such as HumanEval-Infilling (Bavarian et al., 2022), which are based on filling randomly masked lines or character spans. SAFIM is sourced from code on Codeforces and GitHub created after April 2022, deliberately aiming to avoid overlap with mainstream open-source pretraining corpora like The Stack (Kocetkov et al., 2022). This approach reduces the risks of data contamination caused by memoization of test cases, thereby bolstering the credibility of our results. SAFIM, with its 17,720 examples from 8,590 code files, not only surpasses the scale of HumanEval-Infilling, which draws from 164 short code files, but also expands the scope to include multiple programming languages. SAFIM primarily relies on execution-based evaluation, and uses syntactical match evaluation only when execution is not feasible due to external API calls.

Our comprehensive evaluation of 15 LLMs on SAFIM reveals its effectiveness in providing a fair comparison of models. We implement five distinct prompt designs to accommodate various model types and introduce a syntax-aware truncation algorithm for post-processing the outputs. Our approach unveils the true capabilities of non-FIM-trained models, allowing for a fair comparison with FIM-trained models.

Moreover, SAFIM sheds light on the strengths of various pretraining paradigms and challenges some prevalent beliefs in the field. Specifically, our findings suggest that FIM pretraining not only improves LLMs' performance in FIM inference but also enhances their performance in classical Left-to-Right (L2R) inference scenarios. This supports the growing trend of using FIM as the primary pretraining objective in code LLM development. We also observe that pretraining methods and data quality often outweigh the sheer model size—smaller models with sophisticated pretraining paradigms often outperform larger models. This is particularly evident in task-specific performances on

[1]Department of EECS, University of California at Berkeley, Berkeley, California, USA [2]AI at Meta, USA. Correspondence to: Linyuan Gong <gly@berkeley.edu>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

SAFIM, where models pretrained with additional repo-level information excel in API function call completion, while those trained with code execution feedback perform better in control-flow expression generation. However, it is crucial to note that these comparisons across different model families are not controlled experiments and could be influenced by differences in pretraining environments. This suggests future work in pretraining such models under the same environment to validate these observations further. That said, our benchmark, SAFIM, provides a solid foundation for such future research, and opens up new opportunities in designing effective pretraining and fine-tuning paradigms for code LLMs.

2. Related Work

Large Language Models for Code. The emergence of Large Language Models (LLMs) like GPT-3 (Brown et al., 2020) in natural language processing has led to the understanding that merely increasing the number of parameters in pretrained language models will ensure superior performance on unseen tasks. This has led to the application of LLMs to code-related tasks, particularly in code generation. For such tasks, decoder-only models are typically used. Initially, these models, such as Codex (Chen et al., 2021a), PaLM (Chowdhery et al., 2022), PolyCoder (Xu et al., 2022), and CodeGen (Nijkamp et al., 2023), primarily focused on Left-to-Right (L2R) pretraining, a.k.a. "Next Token Prediction." However, the Fill-in-the-Middle (FIM) objective, a.k.a. "Infilling," has become increasingly popular, with models like InCoder (Fried et al., 2023), StarCoder (Li et al., 2023), SantaCoder (Allal et al., 2023), DeepSeek-Coder (Guo et al., 2024), and CodeLLaMa (Rozière et al., 2023) showing their effectiveness. Additionally, proprietary models such as GPT-3.5 (Ouyang et al., 2022), GPT-4 (OpenAI, 2023), and Gemini (Team et al., 2023), which use undisclosed pretraining methods, also contribute to this domain. While GLM-like models (Du et al., 2022) or encoder-decoder models, including CodeGeeX (Zheng et al., 2023), PLBART (Ahmad et al., 2021), AlphaCode (Li et al., 2022), CodeT5 (Wang et al., 2021; 2023a), and AST-T5 (Gong et al., 2024a) exist, they are outside of our paper's scope. Our paper evaluates a select group of these LMs using the SAFIM benchmark. We develop insights into their performance in code FIM tasks, explore the strengths and weaknesses of various pretraining paradigms, and challenge the prevailing belief that a larger number of parameters automatically leads to better performance.

Benchmarking Generative Code LLMs. Existing benchmarks for code generation in LLMs have a gap in effectively evaluating code generation capability for real-world development. Widely-used benchmarks like HumanEval (Chen et al., 2021a) and MBPP (Austin et al., 2021) are limited to single Python functions and also subject to data contamination (Yang et al., 2023). Extensions like HumanEval-X (Zheng et al., 2023), MultiPLe (Cassano et al., 2022), and MBXP (Athiwaratkun et al., 2023) expand these benchmarks to other programming languages. Competition-style coding benchmarks like APPS (Hendrycks et al., 2021) and CodeContests (Li et al., 2022), broaden the scope to file-level code generation. However, they still do not reflect typical development, which often involves iterative codebase expansion and invoking external API libraries. On the other hand, contextually richer benchmarks, such as JuICe (Agashe et al., 2019), DS-1000 (Lai et al., 2022), ARCADE (Yin et al., 2022), NumpyEval (Zhang et al., 2023b), and PandasEval (Jain et al., 2021), PlotCoder (Chen et al., 2021b), ADELT (Gong et al., 2024b) in data science, and APIBench (Patil et al., 2023), RepoBench (Liu et al., 2023), ODEX (Wang et al., 2023b), SWE-Bench (Jimenez et al., 2023), GoogleCodeRepo (Shrivastava et al., 2023), RepoEval (Zhang et al., 2023a), and CoCoMIC-Data (Ding et al., 2023) in software engineering, are often very small, heavily reliant on imperfect match-based evaluation metrics, or lacking in execution-based evaluation. Our SAFIM benchmark, based on Fill-in-the-Middle (FIM) tasks, bridges this gap by providing a comprehensive evaluation framework.

Fill-in-the-Middle in Training and Evaluating Code LLMs. Fill-in-the-Middle (FIM) originates from masked language modeling (MLM) for training encoder-only models (Devlin et al., 2019) and T5-style span corruption for training encoder-decoder models (Raffel et al., 2020), with span lengths usually limited to 1 to 5 tokens, with the goal of targeting representation learning rather than generation. For coding tasks, InCoder (Fried et al., 2023) shows the effectiveness of FIM as a pretraining objective for decoder-only models. Fried et al. (2023) further establishes the HumanEval-Infilling benchmark, further explored by Bavarian et al. (2022) in evaluating GPT-3/Codex variants, showing that a pretraining mix with a 90% FIM ratio does not harm Left-to-Right (L2R) generation performance. CodeLLaMa's evaluations on HumanEval-Infilling support these findings, underscoring the value of FIM in pretraining code-focused LLMs (Rozière et al., 2023). However, this benchmark, limited to the 164 tiny Python snippets of HumanEval, emphasize the need for a more robust benchmark. SAFIM addresses this need by introducing a comprehensive, syntax-aware FIM benchmark for more detailed evaluations.

3. Benchmark Construction

The SAFIM benchmark is designed to evaluate Large Language Models (LLMs) on the Fill-in-the-Middle (FIM) of various code structures. In this section, we describe the collection of the corpora, the generation and filtering of completion tasks, and the evaluation protocols.

```
Calculate max path sum in grid,
only right or down moves allowed

n, m = len(a), len(a[0])
f = np.zeros((n + 1, m + 1))
for i in range(1, n + 1):
  for j in range(1, m + 1):
    v = max(f[i-1,j], f[i,j-1])
    f[i, j] += v
print(f[n, m])
```

Algorithmic Block Completion

```
Calculate (a ^ b) mod m for
large positive integers a, b, m

result = 1
while b > 0:
  if b % 2:
    result = (result * a) % m
  a = (a * a) % m
  b //= 2
print(result)
```

Control-Flow Completion

```
Define word embedding & learned
positional embedding layers

d_model = args.model_dim
n_words = args.vocab_size
max_len = args.max_src_len
self.word_emb = nn.Embedding(
  n_words, d_model)
self.pos_emb = nn.Embedding(
  max_len, d_model)
```

API Function Call Completion

Figure 1. Three splits in the SAFIM benchmark illustrated with code examples. Each example includes a problem description and a code snippet, with a contiguous code segment highlighted in yellow to indicate the part to be masked and completed by LLMs. Contexts in these examples are shortened for clarity.

3.1. Corpora Collection

The SAFIM benchmark is constructed using corpora from two primary sources: *Codeforces* and *GitHub*. Codeforces,[1] a competitive programming platform, offers a wealth of coding problems, unit tests, and solutions. From Codeforces, we scrape problems, unit tests, and their corresponding code solutions. For GitHub, we gather git commits from the GH Archive[2]. From both sources, we gather Python, Java, C++, and C# code files created between April 1, 2022, and January 1, 2023. This selection criteria ensures the inclusion of recent code, avoiding overlap with major pretraining datasets like The Stack (Kocetkov et al., 2022) (cutoff at March 31, 2022) and the training data for GPT-3.5/GPT-4 (cutoff at September 2021), thus reducing the risk of data contamination.

In processing Codeforces data, we reevaluate each code solution by executing unit tests. We retain only those solutions that consistently pass all unit tests within 50% of the specified time limit, eliminating randomness and noise from external factors. We also filter out excessively lengthy (over twice the size of the shortest accepted solution) or near-duplicate solutions (exceeding a CodeBLEU (Ren et al., 2020) score threshold of 0.9 against previously added code), resulting in a curated set of 490 coding questions and 8,590 unique code solutions.

For GitHub, we first establish a list of widely-used API libraries for each programming language, detailed in Appendix A.1. We then extract code files that invoke APIs from such repositories with more than 10 stars to prioritize high-quality code. Files lacking natural language comments or documentation are excluded to avoid unsolvable examples. After thorough filtering and deduplication, our final GitHub corpus consists of 11,936 code files.

[1] https://codeforces.com/
[2] https://www.gharchive.org/

3.2. Generating and Filtering Completion Tasks

With our corpora ready, we parse each code file into an Abstract Syntax Tree (AST). This enables the creation of structured FIM tasks across three splits: algorithmic block completion, control-flow completion, and API function call completion. The first two are based on the Codeforces corpus, while the latter is based on the GitHub corpus as external API function calls are usually absent in competitive programming. In each split, we mask different code segments and ask the models to reconstruct these segments such that the original program functionality is maintained.

Algorithmic Block Completion. Here, we mask a code block critical for solving the coding question, evaluating the LLM's capability in interpreting natural language descriptions and designing algorithms. A "code block" refers to a contiguous list of statements, identified by indentations for Python or curly braces for C-family languages. We target the deepest block in the AST, often the innermost loop layer containing key operations or formulae, like a dynamic programming state transition equation (see Figure 1, Left). To avoid masking non-critical blocks (e.g., logging or debugging), we validate each block: if replacing a block with no-op causes unit test failures, it is included; otherwise, it is excluded. Such filtering ensures that only algorithmically significant blocks are included in the benchmark.

Control-Flow Completion. This category focuses on masking critical control expressions in the program, evaluating the LLM's understanding of code control flows. We mask conditional expressions in statements such as `for`, `while`, `do-while`, `for-each`, `if`, and `else-if`. For example, in Figure 1 (Middle), we mask `b % 2` in an `if` statement, as it determines when the `result` variable will be updated; we mask `b > 0` of the outer layer `if` in a different example. To ensure the relevance of each masked expression, we only retain cases where substituting the expression with

`false`, `true`, or an empty iterable would affect the unit test outcomes. Such filtering guarantees that only expressions critical to the program's control-flow are included in the benchmark.

API Function Call Completion. In this category, we mask calls to functions and object constructors from popular API libraries. This tests the LLM's API knowledge and the ability to integrate such knowledge with code context. Because this split is sourced from the inherently noisy GitHub corpus, we curate the dataset and add necessary hints as comments near each API call, ensuring each example is solvable by humans based on the given context. For example, in Figure 1 (Right), the LLM is expected to deduce the correct arguments `max_len` and `d_model` for a positional embedding layer defined by `nn.Embedding`.

The SAFIM benchmark has 17,720 examples across these three categories, with detailed statistics provided in Appendix A.2.

3.3. Evaluation Protocols

We evaluate completions generated by LLMs using *execution-based testing* and *syntactical matching*. The former applies to algorithmic block and control-flow completions, while the latter is used for API function call completion.

Execution-Based Evaluation is applied to examples with unit tests, covering 98.25% of our benchmark. A completion is considered correct if it passes all unit tests. We use the ExecEval framework (Khan et al., 2023) as our execution environment for this purpose.

Syntactical Match Evaluation is used where unit tests are impractical, which happens in the API function call completion split. This arises due to the potential side effects or dependencies on external environments inherent in external API function calls, which is difficult to check using only unit tests. In such instances, we use syntax matching to evaluate the model's output, comparing it against the ground truth. For instance, outputs like `func(a, b=1, c=2)` are considered equivalent to `func(a, c=2, b=1)`, focusing on syntactical equivalence rather than exact matches.

Our large dataset size of 17,720 examples enables robust evaluations without the need for multiple generations and averaging, as seen in smaller datasets like HumanEval (164 programs). Therefore, we only generate one completion for each LLM on each example and report the percentage of first-attempt passes, i.e., *Pass@1*, as our evaluation metric.

4. Prompts and Post-Processing

We now describe our prompt designs and post-processing techniques for the SAFIM benchmark. These aspects make huge impact in model evaluations but are often overlooked. We introduce our approach for creating prompts and our unique syntax-aware post-processing method, which refines model outputs for more accurate and fair benchmarking.

4.1. Prompts

LLMs' performance is heavily influenced by the design of the prompts (White et al., 2023; Sclar et al., 2023). Using only a limited range of prompt types can skew evaluation results. For instance, Fried et al. (2023) use the Prefix-Suffix-Middle (PSM) prompt for FIM-pretrained models and the Instructed Prefix Feeding (IPF) prompt for others, leading to direct comparisons across different prompt types. This method, however, might yield suboptimal performance for different types of LLMs, leading to inaccurate comparisons. We further discuss this in Section 6. We address these concerns by introducing a wider range of distinct prompts in our evaluations, as detailed in Figure 2:

Left-to-Right (L2R). This baseline consists of only the code's prefix and omits the suffix. It provides a foundation to assess the effectiveness of other prompt designs.

Prefix-Suffix-Middle (PSM). PSM uses a placeholder (a.k.a "sentinel token") to indicate the masked code segment, with the model tasked to generate the segment following the prompt. Effective use of this prompt type, however, requires that the model be pretrained with a FIM objective to recognize and appropriately respond to sentinel tokens.

Suffix-Prefix-Middle (SPM). SPM places the suffix at the beginning and the completion segment immediately after the prefix. This structure enables models, even those not pretrained on FIM objectives like CodeGen, to perform the completion task in a left-to-right manner. This adaptability to non-FIM pretrained makes SPM suitable for a wider range of models, although Rozière et al. (2023) reports SPM's inferior performance compared to PSM in the HumanEval-Infilling benchmark.

Instructed Prefix Feeding (IPF). IPF replaces the masked code with a placeholder, followed by an instruction, and then repeats the prefix. It allows non-FIM pretrained models to recognize and tackle completion tasks (Fried et al., 2023). Our experiments indicate a tendency in some models to erroneously output the placeholder token as part of their output. To address this, we introduce a logits masking technique to inhibit the generation of placeholder tokens, enhancing the effectiveness of IPF.

```
Calculate n-th fibonacci number

n = input()
a, b = 0, 1
for _ in range(n):
  a, b = b, a + b
print(a)
```

Original Code

```
Calculate n-th fibonacci number

n = input()
a, b = 0, 1
for _ in range(n):
  ◁
```

Left-to-Right (L2R)

```
Calculate n-th fibonacci number

n = input()
a, b = 0, 1
for _ in range(n):
  [MASK]
print(a)
[END] ◁
```

Prefix-Suffix-Middle (PSM)

```
[MASK]
print(a)
[END]

Calculate n-th fibonacci number

n = input()
a, b = 0, 1
for _ in range(n):
  ◁
```

Suffix-Prefix-Middle (SPM)

```
Calculate n-th fibonacci number

n = input()
a, b = 0, 1
for _ in range(n):
  [MASK]
print(a)
[END]
Complete the masked part:

n = input()
a, b = 0, 1
for _ in range(n):
  ◁
```

Instructed Prefix Feeding (IPF)

```
Calculate a + b
a, b = input()
[MASK]
print(c)
[END] c = a + b

Calculate n-th fibonacci number

n = input()
a, b = 0, 1
for _ in range(n):
  [MASK]
print(a)
[END] ◁
```

One-Shot (1S)

Figure 2. The original code is shown in the top-left, with the block `a, b = b, a + b` to be masked. The subsequent cells illustrate five distinct prompt types. The "◁" symbol indicates the end of the prompt, where model generation begins. The tokens `[MASK]` and `[END]` are model-specific, e.g., `<SUF>` and `<MID>` for CodeLLaMa, and `<|mask:0|>` and `<|mask:1|>` for InCoder.

One-Shot (1S). Tailored for non-FIM chat models, 1S uses a PSM-style prompt, supplemented with a simple input-output example, which provides the model with context about the task type and the expected input-output format.

4.2. Post-Processing

Post-processing is vital for automatic evaluation of LLMs in code generation, yet its importance is often underestimated. The raw output from LLMs is not immediately suitable for evaluation due to potential inclusions of irrelevant natural language or extra code beyond the targeted structure. SAFIM includes two stages of post-processing to address these challenges:

Code Extraction for Chat Models. We use regex-based heuristics to extract code from outputs of chat models like GPT-4, which often mix natural language with code in the Markdown-formatted outputs.

Truncation. An important challenge for models not fine-tuned for instruction following is their inability to determine the endpoint of their outputs. Often, such models generate the correct response but continue to produce extraneous content. A notable example is CodeGen (Nijkamp et al., 2023), which, due to its open-ended design, lacks the capability to signal an end-of-sequence (`<eos>`), resulting in unbounded output. Therefore, truncation is essential for the effective evaluation of code generation tasks.

However, inconsistencies in truncation methods across different models have led to skewed comparisons in prior work. For example, if the expected output is a Python expression and the truncation method retains only the first line of generated code, it may erroneously dismiss correct expressions that span multiple lines, as illustrated in Figure 1 (Right).

Syntax-Aware Truncation. In SAFIM, we introduce a syntax-aware truncation algorithm, replacing the conventional regex-based heuristics. This approach ensures the precise extraction of targeted code structures, thereby allowing for accurate and fair evaluations across different models.

For the algorithmic block completion task, which requires a code block as output, we use an iterative truncation process on the model's output. This involves sequentially removing the last line of the output until two key conditions are met: **(a)** the truncated output must fit into the AST as a "code block" subtree; and **(b)**, the AST of the remaining code—excluding the completion segment—must align with the AST of the original code, in terms of indentation level for Python or curly brace level for C-family languages. Once

Table 1. Summary of evaluated models, highlighting data cutoff dates, open-source status (OS), and pretraining objectives. Dates in red indicate overlap between the model's pretraining data and the SAFIM benchmark in date range (post-April 2022). Data cutoff dates for InCoder are estimated based on their initial paper draft publication dates. The OS column denotes open-source availability ($\surd$ for yes, $\times$ for no), and the FIM column indicates models pretrained with FIM objectives and support for sentinel tokens in FIM inference. *For CodeLLaMa, only 7B/13B versions support FIM inference, while the 34B version does not.

	#Params	Data Cutoff	OS	FIM
GPT-3.5	175B	Sept 2021	$\times$	$\times$
GPT-4	-	Sept 2021	$\times$	$\times$
CodeGen	350M/2B/6B/16B	Oct 2021	$\surd$	$\times$
InCoder	1.3B/6.7B	$\leq$ Mar 2022	$\surd$	$\surd$
CodeLLaMa	7B/13B/34B	Jul 2022	$\surd$	$\surd^*$
StarCoder	15.5B	Mar 2022	$\surd$	$\surd$
DeepSeekCoder	1.3B/6.7B/33B	Feb 2023	$\surd$	$\surd$

both conditions are satisfied, the truncated output is considered as the model's finalized completion.

For control-flow and API function call completions, our method incrementally adds characters to the output until it satisfies similar syntax matching criteria: the completed segment must form a valid "expression" node in the AST, and the rest of the code aligns precisely with the original code's AST structure.

5. Experimental Setup

We evaluate GPT-3.5 (Brown et al., 2020; Ouyang et al., 2022), GPT-4 (OpenAI, 2023), CodeGen (Nijkamp et al., 2023), InCoder (Fried et al., 2023), CodeLLaMa (Rozière et al., 2023), StarCoder (Li et al., 2023), and DeepSeekCoder (Guo et al., 2024) using SAFIM. As Table 1 shows, these models vary in terms of parameters, data cutoff dates, open-source availability, and pretraining objectives. Given the multilingual (Python, Java, C++, and C#) nature of SAFIM, our selection prioritizes models with multilingual capabilities, and exclude Python-only variants like CodeGen-Mono and StarCoder-Python. As we focus on code sources after April 2022, SAFIM guarantees that, with the exception of CodeLLaMa and DeepSeekCoder, all models are evaluated using clean, out-of-sample test cases. In Appendix A.9, we further discuss the impact of data contamination on our evaluation results.

For GPT-3.5 and GPT-4, we use the OpenAI API for generation. For the remaining models, generation is conducted via the Huggingface `transformers` library, following established practices in Fried et al. (2023), where we use top-p random sampling with $p = 0.95$ and a temperature of 0.2.

Table 2. Pass@1 of each model on algorithmic block completion, evaluated with various prompts and using syntax-aware truncation for post-processing. GPT-3.5, CodeGen-16B, and CodeLLaMa-34B cannot be evaluated with the Prefix-Suffix-Middle (PSM) prompt due to lack of support for FIM sentinel tokens, as discussed in Section 4.1. The most effective prompt type for each model is highlighted in **bold**.

	L2R	PSM	SPM	IPF	1S
GPT-3.5 (175B)	23.2	-	30.1	28.6	**31.2**
CodeGen-16B	24.6	-	**25.9**	15.2	0.4
InCoder-6B	18.1	**25.2**	24.1	12.2	23.2
CodeLLaMa-13B	32.3	10.2	**41.4**	30.9	16.1
CodeLLaMa-34B	35.5	-	**38.5**	35.4	19.6
StarCoder (15.5B)	29.3	44.0	**44.1**	20.8	42.4
DeepSeekCoder-33B	41.6	**60.8**	57.4	33.8	59.9

Model details for reproducibility, including the model identifiers used on OpenAI API and the Huggingface model hub, are provided in Appendix A.3.

6. Experimental Results

We now present the experimental results on our SAFIM benchmark, focusing on the effects of prompt designs, the efficacy of our syntax-aware truncation algorithm, and a comparative analysis of various LLMs across tasks. Given the inherent differences in model training environments and configurations, direct comparisons across different model families should be interpreted with caution. The primary value of our work is in establishing the SAFIM benchmark as a cornerstone for future experiments in this field.

6.1. Impact of Prompt Designs

Table 2 compares the effectiveness of different prompt designs by evaluating each model across various prompts with syntax-aware truncation in post-processing. This experiment reveals that:

Prompt Selection is Crucial for Fair Evaluation in Code FIM Tasks. A narrow selection of prompt types can lead to skewed evaluation results, as different models respond differently due to differences in their pretraining data and methods. A potentially skewed evaluation by Fried et al. (2023) highlights this by comparing FIM models using the PSM prompt against non-FIM models with the IPF prompt. Doing so suggests a misleading superiority of InCoder-6B (25.2%) over CodeGen-16B (15.2%) in Pass@1 on SAFIM. This comparison, however, overlooks that CodeGen-16B achieves a higher Pass@1 of 25.9% with the SPM prompt, a prompt not included in their evaluation setup. This example shows the necessity for a comprehensive prompt range to ensure fairness. Our work addresses this by reporting the

Table 3. Comparison of model performance with and without our syntax-aware truncation algorithm in the post-processing phase. This table presents two numbers for each model evaluated on algorithmic block completion tasks: **Pass@1** and **CErr%** (the percentage of unexecutable programs due to compile or syntax errors in the generated completions).

	No Trunc.		Syntax Trunc.	
	Pass@1	**CErr%**	**Pass@1**	**CErr%**
GPT-3.5 (175B)	28.7	25.3	31.2	17.0
GPT-4 (> 220B)	41.7	25.4	42.1	22.9
CodeGen-16B	0.0	99.9	25.9	17.9
InCoder-6B	21.8	25.7	25.2	13.2
CodeLLaMa-13B	16.4	64.6	41.4	10.9
CodeLLaMa-34B	1.0	94.5	38.5	14.7
StarCoder (15.5B)	42.7	14.3	44.1	9.5
DeepSeekCoder-33B	59.7	8.0	60.8	4.0

best-performing prompt for each model and includes an extensive result table in Appendix A.4 for thorough comparison.

FIM Pretraining Boosts *Both* FIM and L2R Performance. Pretraining LLMs with a FIM objective enhances their performance not only in FIM but also in left-to-right (L2R) generation. The advantage in FIM evaluation is highlighted by the results of CodeLLaMa models: the larger CodeLLaMa-34B, without FIM pretraining, is outperformed by the smaller, FIM+L2R pre-trained CodeLLaMa-13B. A more interesting observation emerges in the "L2R" column of Table 2: FIM-pretrained models like StarCoder outperform purely L2R-pretrained models like CodeGen-16B in L2R mode, despite similar sizes. This finding suggests that FIM pretraining does not harm, and actually enhances, a model's L2R performance, possibly by fostering a better understanding of code via contextually rich pretraining inputs. This supports similar improvements observed in FIM-pretrained GPT-3/Codex models in prior studies (Bavarian et al., 2022), and offer strong justification for the recent shift from pure L2R pretraining to FIM pretraining among code LLM developers (Li et al., 2023; Guo et al., 2024; Rozière et al., 2023).

6.2. Impact of Our Syntax-Aware Truncation

We assess the impact of our syntax-aware truncation algorithm through an ablation study, measuring model performance on the algorithmic block completion task with and without syntax-aware truncation. This analysis focuses on two key numbers: Pass@1 and the percentage of unexecutable programs due to compile or syntax errors in the generated completions. We treat empty outputs after truncation, typically indicative of a failure to identify any valid executable, as compilation errors. The results are shown in Table 3. These results show that:

Syntax-Aware Truncation Enhances FIM Output Quality. Table 3 shows that our syntax-aware truncation algorithm not only enhances the Pass@1 rates but also significantly reduces compilation errors across various models. This indicates a consistent improvement in the quality of FIM outputs, achieved without additional GPU overhead during model inference. We believe syntax-aware truncation holds promise for real-world code completion applications.

Syntax-Aware Truncation Enables Fair Comparison for Non-FIM Models. As shown in Table 3, syntax-aware truncation benefits non-FIM models much more than FIM models. For example, CodeLLaMa-13B's Pass@1 rate jumps from 16.4% to 41.4% with truncation, changing its comparative performance against InCoder-6B, whose Pass@1 only increases marginally from 21.8% to 25.2%. This discrepancy stems from their distinct training approaches. InCoder, exclusively trained on FIM, naturally aligns with FIM-style prompts. In contrast, CodeLLaMa-13B, with a primary focus on L2R in its mixed FIM+L2R training, often produces unwanted extra code after completion. The extra code, while removable by syntax-aware truncation, obscures CodeLLaMa-13B's true effectiveness when such truncation is not applied. By precisely eliminating the extra code, syntax-aware truncation unveils the true coding proficiency of non-FIM or hybrid models like CodeLLaMa, ensuring fair comparisons with FIM-focused models. Additionally, syntax-aware truncation allows open-ended models to be evaluated in FIM tasks.

6.3. Comparative Performance Analysis of LLMs

After determining the most effective prompt for each model and verifying the benefits of syntax-aware truncation, we conduct comprehensive evaluations across the entire SAFIM benchmark. Table 4 shows model performances in each task category, and Figure 3 visualizes the average performance of models against their model sizes. These results offers insights into the capabilities and limitations of code LLMs:

Pretraining Method and Data Are More Important Than Sheer Model Size. Smaller models with sophisticated pretraining paradigms can match or even outperform larger counterparts. For example, StarCoder, with 15.5B parameters, achieves an average Pass@1 of 55.5%, comparable to GPT-4's 53.3%, despite GPT-4's vast size. This pattern recurs in models like CodeLLaMa-13B and DeepSeekCoder-1.3B. Notably, the comparison between StarCoder and GPT-4 is *not* subject to data contamination, as discussed in Table 1. This finding challenges the common belief that larger models automatically yield superior performance, even with

Table 4. Pass@1 of various models on the SAFIM benchmark, showing their performance in algorithmic block completion (Algo.), control-flow completion (Control), and API function call completion (API). The table also reports the average performance, indicating each model's overall effectiveness on SAFIM.

	Algo.	Control	API	Avg
GPT-3.5 (175B)	31.2	37.5	53.9	40.9
GPT-4 (> 220B)	42.1	55.2	62.6	53.3
CodeGen-350M	16.3	26.1	26.5	22.9
CodeGen-2B	23.5	32.9	32.3	29.5
CodeGen-6B	23.6	34.8	27.7	28.7
CodeGen-16B	25.9	35.7	31.3	31.0
InCoder-1B	21.1	22.9	43.9	29.3
InCoder-6B	25.2	28.2	48.1	33.8
CodeLLaMa-7B	34.7	53.6	46.8	45.0
CodeLLaMa-13B	41.4	57.2	59.7	52.8
CodeLLaMa-34B	38.5	54.0	56.5	49.7
StarCoder (15.5B)	44.1	54.5	68.1	55.5
DeepSeekCoder-1.3B	41.2	54.1	62.6	52.6
DeepSeekCoder-6.7B	54.7	65.8	69.7	63.4
DeepSeekCoder-33B	**60.8**	**71.1**	**75.2**	**69.0**

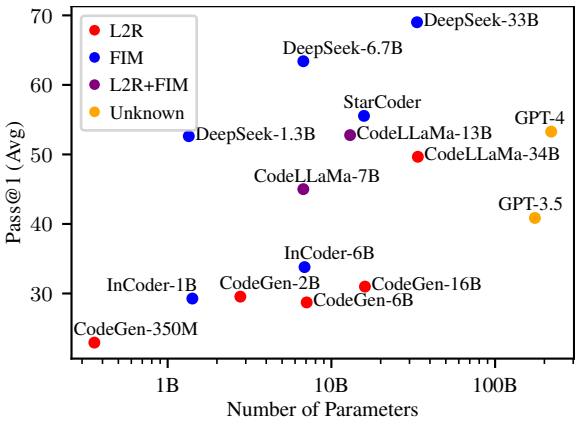

Figure 3. Average performance of different models relative to their sizes on the SAFIM benchmark. Each model is represented by a dot, with the x-axis showing model size (number of parameters) and the y-axis showing average performance across three task categories. Dot colors signify pretraining paradigms: red for Left-to-Right (L2R), blue for FIM, purple for a combination of L2R and FIM, and orange for proprietary models with undisclosed pretraining methods.

basic pretraining methods (Brown et al., 2020). Our study suggests that this may not hold true for coding tasks: within the same model family, performance gains from increased size are only modest, while models from different families exhibit substantial performance variations. For example, the weakest CodeLLaMa model surpasses the strongest CodeGen model by 14 points, a far more significant margin than the 7.8-point spread within CodeLLaMa models.

Pretraining Method and Data Influence Task-Specific Performance. We have discussed in Section 6.1 that FIM pretraining enhances performance on both FIM evaluation and L2R completion. Dissecting model performance across SAFIM's three splits sheds further light on this impact:

- For API function call completion, repository-level information is key. StarCoder and DeepSeekCoder, which excel in this task, both incorporate repository context into their pretraining data. StarCoder enriches its training input with GitHub issues and commit messages, while DeepSeekCoder organize code files according to their topological ordering based on API dependencies. These techniques significantly enhance their ability to understand API contexts.
- For control-flow completion, CodeLLaMa's relatively strong performance is attributed to its use of execution-based feedback in its self-instruct training method. By executing generated code and applying the results as rewards or penalties, CodeLLaMa learns to avoid generating unexecutable code or infinite loops, thereby gaining a more refined understanding of control flows.

These findings highlight the pivotal role of the pretraining paradigm in the performance of LLMs on coding tasks.

7. Conclusion and Future Work

We introduced the Syntax-Aware Fill-in-the-Middle (SAFIM) benchmark, the first large-scale, multilingual Fill-in-the-Middle (FIM) benchmark equipped with executable unit tests for evaluating code-centric Large Language Models (LLMs). To mitigate data contamination, SAFIM adopts a strict cutoff date for code sources. Moreover, SAFIM uniquely categorizes tasks into three syntax-driven splits: algorithmic block completion, control-flow expression completion, and API function call completion. These splits provide a comprehensive assessment of LLMs' coding capabilities across multiple dimensions. SAFIM's suite of prompts and its novel syntax-aware truncation algorithm for post-processing enable fair comparisons among various types of models, including those not explicitly pretrained on FIM tasks.

The results of our large-scale evaluation highlight the significant impact of pretraining paradigms on LLMs' performance, emphasizing the importance of training method and data quality over sheer model size. We found that FIM pretraining can enhance, rather than harm, Left-to-Right (L2R) inference capabilities, supporting a shift towards FIM

as a primary pretraining objective for code LLMs. We acknowledge a key limitation in our study: our conclusions are drawn from comparisons across various model families trained with different paradigms, rather than from controlled experiments altering pretraining paradigms within the same model. Yet, SAFIM establishes a foundational framework for future research into pretraining paradigms and the development of better LLMs for coding tasks.

Acknowledgements

We thank the reviewers at ICML for their constructive feedback and support. This work is supported in part by gift from Meta, the U.S. National Science Foundation through grants IIS-1955488, IIS-2027575, ARO W911NF2110339, ONR N00014-21-1-2724, and DOE awards DE-SC0016260, DE-SC0021982. We thank Chenyan Xiong and Carlton Shen for providing computational resources that enabled us to conduct urgent experiments during the rebuttal period.

Impact Statement

In this paper, we introduce Syntax-Aware Fill-in-the-Middle (SAFIM), a benchmark aimed at enhancing the capabilities of Large Language Models (LLMs) in code generation tasks. The advancement of LLMs in code generation raises concerns about automated code production's security, privacy, and potential misuse. There is a risk that improved code generation capabilities could be exploited for malicious purposes, such as automating the creation of software vulnerabilities or facilitating the development of harmful software. Our research emphasizes the importance of responsible AI development and use, advocating for continuous monitoring, ethical guidelines, and safeguards to mitigate these risks.

References

Agashe, R., Iyer, S., and Zettlemoyer, L. JuICe: A large scale distantly supervised dataset for open domain context-based code generation. (arXiv:1910.02216), October 2019. doi: 10.48550/arXiv.1910.02216. URL http://arxiv.org/abs/1910.02216. arXiv:1910.02216 [cs].

Ahmad, W. U., Chakraborty, S., Ray, B., and Chang, K.-W. Unified pre-training for program understanding and generation. Apr 2021. doi: 10.48550/arXiv.2103.06333. URL http://arxiv.org/abs/2103.06333. arXiv:2103.06333 [cs].

Allal, L. B., Li, R., Kocetkov, D., Mou, C., Akiki, C., Ferrandis, C. M., Muennighoff, N., Mishra, M., Gu, A., Dey, M., Umapathi, L. K., Anderson, C. J., Zi, Y., Poirier, J. L., Schoelkopf, H., Troshin, S., Abulkhanov, D., Romero, M., Lappert, M., De Toni, F., del Río, B. G., Liu, Q., Bose, S., Bhattacharyya, U., Zhuo, T. Y., Yu, I., Villegas, P., Zocca, M., Mangrulkar, S., Lansky, D., Nguyen, H., Contractor, D., Villa, L., Li, J., Bahdanau, D., Jernite, Y., Hughes, S., Fried, D., Guha, A., de Vries, H., and von Werra, L. SantaCoder: don't reach for the stars! (arXiv:2301.03988), February 2023. doi: 10.48550/arXiv.2301.03988. URL http://arxiv.org/abs/2301.03988. arXiv:2301.03988 [cs].

Athiwaratkun, B., Gouda, S. K., Wang, Z., Li, X., Tian, Y., Tan, M., Ahmad, W. U., Wang, S., Sun, Q., Shang, M., Gonugondla, S. K., Ding, H., Kumar, V., Fulton, N., Farahani, A., Jain, S., Giaquinto, R., Qian, H., Ramanathan, M. K., Nallapati, R., Ray, B., Bhatia, P., Sengupta, S., Roth, D., and Xiang, B. Multi-lingual evaluation of code generation models. (arXiv:2210.14868), March 2023. doi: 10.48550/arXiv.2210.14868. URL http://arxiv.org/abs/2210.14868. arXiv:2210.14868 [cs].

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. Program synthesis with large language models. Aug 2021. doi: 10.48550/arXiv.2108.07732. URL http://arxiv.org/abs/2108.07732. arXiv:2108.07732 [cs].

Bavarian, M., Jun, H., Tezak, N., Schulman, J., McLeavey, C., Tworek, J., and Chen, M. Efficient training of language models to fill in the middle. (arXiv:2207.14255), July 2022. doi: 10.48550/arXiv.2207.14255. URL http://arxiv.org/abs/2207.14255. arXiv:2207.14255 [cs].

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. (arXiv:2005.14165), July 2020. doi: 10.48550/arXiv.2005.14165. URL http://arxiv.org/abs/2005.14165. arXiv:2005.14165 [cs].

Cassano, F., Gouwar, J., Nguyen, D., Nguyen, S., Phipps-Costin, L., Pinckney, D., Yee, M.-H., Zi, Y., Anderson, C. J., Feldman, M. Q., Guha, A., Greenberg, M., and Jangda, A. MultiPL-E: A scalable and extensible approach to benchmarking neural code generation. (arXiv:2208.08227), December 2022. doi: 10.48550/arXiv.2208.08227. URL http://arxiv.org/abs/2208.08227. arXiv:2208.08227 [cs].

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D.,

Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code. Jul 2021a. doi: 10.48550/arXiv.2107.03374. URL http://arxiv.org/abs/2107.03374. arXiv:2107.03374 [cs].

Chen, X., Gong, L., Cheung, A., and Song, D. PlotCoder: Hierarchical decoding for synthesizing visualization code in programmatic context. In Zong, C., Xia, F., Li, W., and Navigli, R. (eds.), *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 2169–2181, Online, August 2021b. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.169. URL https://aclanthology.org/2021.acl-long.169.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., Garcia, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A. M., Pillai, T. S., Pellat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Diaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K., Eck, D., Dean, J., Petrov, S., and Fiedel, N. PaLM: Scaling language modeling with pathways. Oct 2022. doi: 10.48550/arXiv.2204.02311. URL http://arxiv.org/abs/2204.02311. arXiv:2204.02311 [cs].

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. May 2019. doi: 10.48550/arXiv.1810.04805. URL http://arxiv.org/abs/1810.04805. arXiv:1810.04805 [cs].

Ding, Y., Wang, Z., Ahmad, W. U., Ramanathan, M. K., Nallapati, R., Bhatia, P., Roth, D., and Xiang, B. CoCoMIC: Code completion by jointly modeling in-file and cross-file context. (arXiv:2212.10007), May 2023. doi: 10.48550/arXiv.2212.10007. URL http://arxiv.org/abs/2212.10007. arXiv:2212.10007 [cs].

Du, Z., Qian, Y., Liu, X., Ding, M., Qiu, J., Yang, Z., and Tang, J. GLM: General language model pretraining with autoregressive blank infilling. (arXiv:2103.10360), March 2022. doi: 10.48550/arXiv.2103.10360. URL http://arxiv.org/abs/2103.10360. arXiv:2103.10360 [cs].

Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W.-t., Zettlemoyer, L., and Lewis, M. InCoder: A generative model for code infilling and synthesis. (arXiv:2204.05999), April 2023. doi: 10.48550/arXiv.2204.05999. URL http://arxiv.org/abs/2204.05999. arXiv:2204.05999 [cs].

Gong, L., Elhoushi, M., and Cheung, A. AST-T5: Structure-aware pretraining for code generation and understanding. (arXiv:2401.03003), January 2024a. doi: 10.48550/arXiv.2401.03003. URL http://arxiv.org/abs/2401.03003. arXiv:2401.03003 [cs].

Gong, L., Wang, J., and Cheung, A. ADELT: Transpilation between deep learning frameworks. (arXiv:2303.03593), May 2024b. doi: 10.48550/arXiv.2303.03593. URL http://arxiv.org/abs/2303.03593. arXiv:2303.03593 [cs].

Gunasekar, S., Zhang, Y., Aneja, J., Mendes, C. C. T., Del Giorno, A., Gopi, S., Javaheripi, M., Kauffmann, P., de Rosa, G., Saarikivi, O., Salim, A., Shah, S., Behl, H. S., Wang, X., Bubeck, S., Eldan, R., Kalai, A. T., Lee, Y. T., and Li, Y. Textbooks are all you need. (arXiv:2306.11644), October 2023. doi: 10.48550/arXiv.2306.11644. URL http://arxiv.org/abs/2306.11644. arXiv:2306.11644 [cs].

Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., Chen, G., Bi, X., Wu, Y., Li, Y. K., Luo, F., Xiong, Y., and Liang, W. DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence. (arXiv:2401.14196), January 2024. doi: 10.48550/arXiv.2401.14196. URL http://arxiv.org/abs/2401.14196. arXiv:2401.14196 [cs].

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., and Steinhardt, J. Measuring coding challenge competence with APPS. (arXiv:2105.09938), November 2021. doi: 10.48550/arXiv.2105.09938. URL http://arxiv.org/abs/2105.09938. arXiv:2105.09938 [cs].

Jain, N., Vaidyanath, S., Iyer, A., Natarajan, N., Parthasarathy, S., Rajamani, S., and Sharma, R. Jigsaw: Large language models meet program synthesis. (arXiv:2112.02969), December 2021. doi: 10.48550/arXiv.2112.02969. URL http://arxiv.org/abs/2112.02969. arXiv:2112.02969 [cs].

Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., Casas, D. d. l., Hanna, E. B., Bressand, F., Lengyel, G., Bour, G., Lample, G., Lavaud, L. R., Saulnier, L., Lachaux, M.-A., Stock, P., Subramanian, S., Yang, S., Antoniak, S., Scao, T. L., Gervet, T., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E. Mixtral of experts. (arXiv:2401.04088), January 2024. doi: 10.48550/arXiv.2401.04088. URL http://arxiv.org/abs/2401.04088. arXiv:2401.04088 [cs].

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. SWE-Bench: Can language models resolve real-world Github issues? (arXiv:2310.06770), October 2023. doi: 10.48550/arXiv.2310.06770. URL http://arxiv.org/abs/2310.06770. arXiv:2310.06770 [cs].

Khan, M. A. M., Bari, M. S., Do, X. L., Wang, W., Parvez, M. R., and Joty, S. xCodeEval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. (arXiv:2303.03004), November 2023. doi: 10.48550/arXiv.2303.03004. URL http://arxiv.org/abs/2303.03004. arXiv:2303.03004 [cs].

Kocetkov, D., Li, R., Allal, L. B., Li, J., Mou, C., Ferrandis, C. M., Jernite, Y., Mitchell, M., Hughes, S., Wolf, T., Bahdanau, D., von Werra, L., and de Vries, H. The Stack: 3 TB of permissively licensed source code. (arXiv:2211.15533), November 2022. doi: 10.48550/arXiv.2211.15533. URL http://arxiv.org/abs/2211.15533. arXiv:2211.15533 [cs].

Lai, Y., Li, C., Wang, Y., Zhang, T., Zhong, R., Zettlemoyer, L., Yih, S. W.-t., Fried, D., Wang, S., and Yu, T. DS-1000: A natural and reliable benchmark for data science code generation. (arXiv:2211.11501), November 2022. doi: 10.48550/arXiv.2211.11501. URL http://arxiv.org/abs/2211.11501. arXiv:2211.11501 [cs].

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

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A. D., Hubert, T., Choy, P., d'Autume, C. d. M., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., Molloy, J., Mankowitz, D. J., Robson, E. S., Kohli, P., de Freitas, N., Kavukcuoglu, K., and Vinyals, O. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, December 2022. ISSN 0036-8075, 1095-9203. doi: 10.1126/science.abq1158. URL http://arxiv.org/abs/2203.07814. arXiv:2203.07814 [cs].

Liu, T., Xu, C., and McAuley, J. RepoBench: Benchmarking repository-level code auto-completion systems. (arXiv:2306.03091), October 2023. doi: 10.48550/arXiv.2306.03091. URL http://arxiv.org/abs/2306.03091. arXiv:2306.03091 [cs].

Luo, Z., Xu, C., Zhao, P., Sun, Q., Geng, X., Hu, W., Tao, C., Ma, J., Lin, Q., and Jiang, D. WizardCoder: Empowering code large language models with Evol-Instruct. (arXiv:2306.08568), June 2023. doi: 10.48550/arXiv.2306.08568. URL http://arxiv.org/abs/2306.08568. arXiv:2306.08568 [cs].

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. Codegen: An open large language model for code with multi-turn program synthesis. (arXiv:2203.13474), February 2023. doi: 10.48550/arXiv.2203.13474. URL http://arxiv.org/abs/2203.13474. arXiv:2203.13474 [cs].

OpenAI. GPT-4 technical report. (arXiv:2303.08774), December 2023. doi: 10.48550/arXiv.2303.08774. URL http://arxiv.org/abs/2303.08774. arXiv:2303.08774 [cs].

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J., and Lowe, R. Training language models to follow instructions with human feedback. (arXiv:2203.02155), March 2022. doi: 10.48550/arXiv.2203.02155. URL http://arxiv.org/abs/2203.02155. arXiv:2203.02155 [cs].

Patil, S. G., Zhang, T., Wang, X., and Gonzalez, J. E. Gorilla: Large language model connected with massive apis. (arXiv:2305.15334), May 2023. doi: 10.48550/arXiv.2305.15334. URL http://arxiv.org/abs/2305.15334. arXiv:2305.15334 [cs].

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. Jul 2020. doi: 10.48550/arXiv.1910.10683. URL http://arxiv.org/abs/1910.10683. arXiv:1910.10683 [cs, stat].

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., and Ma, S. CodeBLEU: a method for automatic evaluation of code synthesis. (arXiv:2009.10297), September 2020. doi: 10.48550/arXiv.2009.10297. URL http://arxiv.org/abs/2009.10297. arXiv:2009.10297 [cs].

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., and Synnaeve, G. Code Llama: Open foundation models for code. (arXiv:2308.12950), August 2023. doi: 10.48550/arXiv.2308.12950. URL http://arxiv.org/abs/2308.12950. arXiv:2308.12950 [cs].

Sclar, M., Choi, Y., Tsvetkov, Y., and Suhr, A. Quantifying language models' sensitivity to spurious features in prompt design or: How i learned to start worrying about prompt formatting. (arXiv:2310.11324), October 2023. doi: 10.48550/arXiv.2310.11324. URL http://arxiv.org/abs/2310.11324. arXiv:2310.11324 [cs].

Shrivastava, D., Larochelle, H., and Tarlow, D. Repository-level prompt generation for large language models of code. (arXiv:2206.12839), June 2023. doi: 10.48550/arXiv.2206.12839. URL http://arxiv.org/abs/2206.12839. arXiv:2206.12839 [cs].

Team, G., Anil, R., Borgeaud, S., Wu, Y., Alayrac, J.-B., Yu, J., Soricut, R., Schalkwyk, J., Dai, A. M., Hauth, A., Millican, K., Silver, D., Petrov, S., Johnson, M., Antonoglou, I., Schrittwieser, J., Glaese, A., Chen, J., Pitler, E., Lillicrap, T., Lazaridou, A., Firat, O., Molloy, J., Isard, M., Barham, P. R., Hennigan, T., Lee, B., Viola, F., Reynolds, M., Xu, Y., Doherty, R., Collins, E., Meyer, C., Rutherford, E., Moreira, E., Ayoub, K., Goel, M., Tucker, G., Piqueras, E., Krikun, M., Barr, I., Savinov, N., Danihelka, I., Roelofs, B., White, A., Andreassen, A., von Glehn, T., Yagati, L., Kazemi, M., Gonzalez, L., and Others. Gemini: A family of highly capable multimodal models. (arXiv:2312.11805), December 2023. doi: 10.48550/arXiv.2312.11805. URL http://arxiv.org/abs/2312.11805. arXiv:2312.11805 [cs].

Wang, Y., Wang, W., Joty, S., and Hoi, S. C. H. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. Sep 2021. doi: 10.48550/arXiv.2109.00859. URL http://arxiv.org/abs/2109.00859. arXiv:2109.00859 [cs].

Wang, Y., Le, H., Gotmare, A. D., Bui, N. D. Q., Li, J., and Hoi, S. C. H. CodeT5+: Open code large language models for code understanding and generation. May 2023a. doi: 10.48550/arXiv.2305.07922. URL http://arxiv.org/abs/2305.07922. arXiv:2305.07922 [cs].

Wang, Z., Zhou, S., Fried, D., and Neubig, G. Execution-based evaluation for open-domain code generation. (arXiv:2212.10481), May 2023b. doi: 10.48550/arXiv.2212.10481. URL http://arxiv.org/abs/2212.10481. arXiv:2212.10481 [cs].

Wei, Y., Wang, Z., Liu, J., Ding, Y., and Zhang, L. Magicoder: Source code is all you need. (arXiv:2312.02120), December 2023. doi: 10.48550/arXiv.2312.02120. URL http://arxiv.org/abs/2312.02120. arXiv:2312.02120 [cs].

White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., and Schmidt, D. C. A prompt pattern catalog to enhance prompt engineering with chatgpt. (arXiv:2302.11382), February 2023. doi: 10.48550/arXiv.2302.11382. URL http://arxiv.org/abs/2302.11382. arXiv:2302.11382 [cs].

Xu, F. F., Alon, U., Neubig, G., and Hellendoorn, V. J. A systematic evaluation of large language models of code. (arXiv:2202.13169), May 2022. doi: 10.48550/arXiv.2202.13169. URL http://arxiv.org/abs/2202.13169. arXiv:2202.13169 [cs].

Yang, S., Chiang, W.-L., Zheng, L., Gonzalez, J. E., and Stoica, I. Rethinking benchmark and contamination for language models with rephrased samples. (arXiv:2311.04850), November 2023. doi: 10.48550/arXiv.2311.04850. URL http://arxiv.org/abs/2311.04850. arXiv:2311.04850 [cs].

Yin, P., Li, W.-D., Xiao, K., Rao, A., Wen, Y., Shi, K., Howland, J., Bailey, P., Catasta, M., Michalewski, H., Polozov, A., and Sutton, C. Natural language to code generation in interactive data science notebooks. (arXiv:2212.09248), December 2022. doi: 10.48550/arXiv.2212.09248. URL http://arxiv.org/abs/2212.09248. arXiv:2212.09248 [cs].

Zhang, F., Chen, B., Zhang, Y., Keung, J., Liu, J., Zan, D., Mao, Y., Lou, J.-G., and Chen, W. RepoCoder: Repository-level code completion through iterative retrieval and generation. (arXiv:2303.12570), October 2023a. doi: 10.48550/arXiv.2303.12570. URL http://arxiv.org/abs/2303.12570. arXiv:2303.12570 [cs].

Zhang, K., Zhang, H., Li, G., Li, J., Li, Z., and Jin, Z. ToolCoder: Teach code generation models to use api search tools. (arXiv:2305.04032), September 2023b. doi: 10.48550/arXiv.2305.04032. URL http://arxiv.org/abs/2305.04032. arXiv:2305.04032 [cs].

Zheng, Q., Xia, X., Zou, X., Dong, Y., Wang, S., Xue, Y., Wang, Z., Shen, L., Wang, A., Li, Y., Su, T., Yang, Z., and Tang, J. CodeGeeX: A pre-trained model for code generation with multilingual evaluations on humaneval-x. (arXiv:2303.17568), March 2023. doi: 10.48550/arXiv.2303.17568. URL http://arxiv.org/abs/2303.17568. arXiv:2303.17568 [cs].

A. Appendix

A.1. Details about the API Function Call Completion Task

We consider the following API libraries for each programming language when we construct the API function call completion split of SAFIM:

- **Python:** NumPy, Pandas, Statsmodels, Sci-kit Learn, Matplotlib, NLTK, Gensim, XGBoost, PyTorch, Huggingface Transformers
- **Java:** GSON, Caffeine, Apache Commons, Google HTTP Client, Joda-Time, JavaParser,
- **C++:** GMP, Boost, JSON, QT, Eigen, OpenGL, Tree-Sitter
- **C#:** Newtonsoft.Json, SignalR, RestSharp, LiteDB, BCrypt.Net

A.2. Statistics of the SAFIM Benchmark

Statistics of each split of the SAFIM benchmark is presented in Table 5.

Table 5. Statistics of each task category of the SAFIM benchmark, including number of examples, total uncompressed disk size of code contexts, average length of code contexts in bytes, and average length of ground truth completions in bytes.

	Source	# Examples	Disk Size	Avg Code Len	Avg Completion Len
Algorithmic Block	Codeforces	8,781	29.4M	3346B	67B
Control-Flow	Codeforces	8,629	29.5M	3415B	16B
API Function Call	GitHub	310	713K	2302B	40B
Total	-	17,720	59.6M	3364B	42B

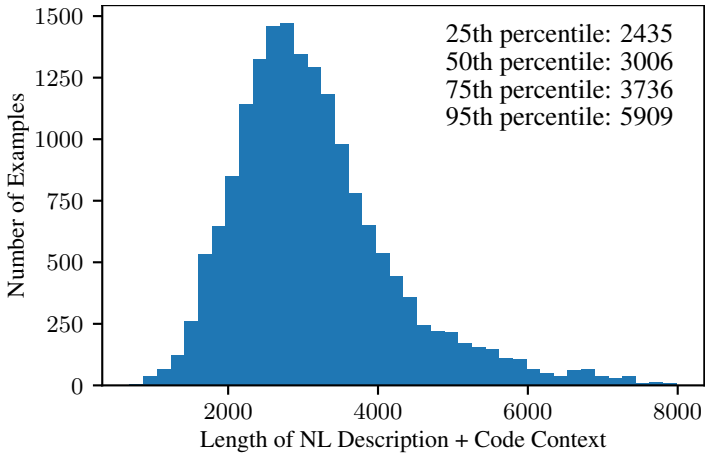

Figure 4. Histogram of the total number of **characters** of the natural language problem description and the code context. 424 example longer than 8,000 characters are excluded from this histogram for clarity but counted towards the displayed quantiles.

Figure 4 shows the distribution the total lengths of problem descriptions plus code contexts of examples measured in characters. A majority of the dataset has less than 6k characters. On average, one BPE token corresponds to 3 to 4 characters in the code domain. This ensures that the evaluated models, all with context windows of at least 2,048, accurately reflect performance without bias from input size.

Table 6 shows statistics per programming language of examples in SAFIM. This distribution of PLs in SAFIM mirrors the prevalence in our source corpus, especially in Codeforces where most contestants use C++. Table 6 also highlights variations

Table 6. Statistics of examples in each programming language of the SAFIM benchmark, including number of examples, total uncompressed disk size of code contexts, average length of code contexts in bytes, average length of ground truth completions in bytes, and average length of identifiers in bytes. The *identifiers* refer to the names of variables, functions, and classes.

	# Examples	Disk Size	Avg Code Len	Avg Completion Len	Avg Identifier Len
Python	9,901	30.0M	3026B	44B	2.73B
Java	4,999	17.3M	3454B	44B	4.14B
C++	1,736	5.14M	2962B	27B	3.65B
C#	1,084	7.24M	6675B	42B	5.79B
Total	17,720	59.6M	3364B	42B	3.59B

in coding style across languages. For instance, C++ and Python programmers favor succinct coding (less code, shorter variable names), while Java and C# users lean towards verbosity. Subsequent sections will discuss how these differences in coding style influence evaluation results.

A.3. Details of Model Implementations

Table 1 shows the implementations used for evaluating each LLM, including additional models we will discuss in Appendix A.6. For GPT-3.5 and GPT-4, we use the OpenAI API[3] for generation. For the remaining models, generation is conducted via the Huggingface `transformers` library[4].

Table 7. The code enviroment for evaluating each LLM and the model identifier on its respective platform.

	Codebase	Model Identifier
GPT-3.5	OpenAI API	`gpt-3.5-turbo-0301`
GPT-4	OpenAI API	`gpt-4-1106-preview`
CodeGen-350M	Huggingface Transformers	`Salesforce/codegen-350M-multi`
CodeGen-2B	Huggingface Transformers	`Salesforce/codegen-2B-multi`
CodeGen-6B	Huggingface Transformers	`Salesforce/codegen-6B-multi`
CodeGen-16B	Huggingface Transformers	`Salesforce/codegen-16B-multi`
InCoder-1B	Huggingface Transformers	`facebook/incoder-1B`
InCoder-6B	Huggingface Transformers	`facebook/incoder-6B`
CodeLLaMa-7B	Huggingface Transformers	`codellama/CodeLlama-7b-hf`
CodeLLaMa-13B	Huggingface Transformers	`codellama/CodeLlama-13b-hf`
CodeLLaMa-34B	Huggingface Transformers	`codellama/CodeLlama-34b-hf`
StarCoder (15.5B)	Huggingface Transformers	`bigcode/starcoderbase`
DeepSeekCoder-1.3B	Huggingface Transformers	`deepseek-ai/deepseek-coder-1.3b-base`
DeepSeekCoder-6.7B	Huggingface Transformers	`deepseek-ai/deepseek-coder-6.7b-base`
DeepSeekCoder-33B	Huggingface Transformers	`deepseek-ai/deepseek-coder-33b-base`
Mixtral-8x7B	Huggingface Transformers	`mistralai/Mixtral-8x7B-v0.1`
Phi-1.5 (1.3B)	Huggingface Transformers	`microsoft/phi-1_5`
Phi-2 (2.7B)	Huggingface Transformers	`microsoft/phi-2`
WizardCoder-1B	Huggingface Transformers	`WizardLM/WizardCoder-1B-V1.0`
WizardCoder-3B	Huggingface Transformers	`WizardLM/WizardCoder-3B-V1.0`
WizardCoder-15B	Huggingface Transformers	`WizardLM/WizardCoder-15B-V1.0`
WizardCoder-33B	Huggingface Transformers	`WizardLM/WizardCoder-33B-V1.1`
Magicoder-S-DS-6.7B	Huggingface Transformers	`ise-uiuc/Magicoder-S-DS-6.7B`

A.4. Results of All Models on All Prompts

Table 8, Table 9, and Table 10 show experimental results of all models using all types of prompts, where each table shows the results on one task category of SAFIM.

[3] https://openai.com/blog/openai-api

[4] https://github.com/huggingface/transformers

Table 8. The performance of each model with each type of prompts on algorithmic block completion. Syntax-aware truncation is used for post-processing. The most effective prompt type for each model is highlighted in **bold**.

	L2R	PSM	SPM	IPF	1S
GPT-3.5 (175B)	23.2	-	30.1	28.6	**31.2**
GPT-4	-	-	-	-	**42.1**
CodeGen-350M	15.4	-	**16.3**	6.8	0.1
CodeGen-2B	22.5	-	**23.5**	13.9	0.0
CodeGen-6B	23.2	-	**23.6**	14.6	0.0
CodeGen-16B	24.6	-	**25.9**	15.2	0.4
InCoder-1B	14.1	**21.1**	19.2	9.0	17.6
InCoder-6B	18.1	**25.2**	24.1	12.2	23.2
CodeLLaMa-7B	30.7	8.8	**34.7**	24.4	7.5
CodeLLaMa-13B	32.3	10.2	**41.4**	30.9	16.1
CodeLLaMa-34B	35.5	-	**38.5**	35.4	19.6
StarCoder (15.5B)	29.3	44.0	**44.1**	20.8	42.4
DeepSeekCoder-1.3B	28.0	**41.2**	38.7	6.5	38.0
DeepSeekCoder-6.7B	36.2	**54.7**	51.3	27.1	52.9
DeepSeekCoder-33B	41.6	**60.8**	57.4	33.8	59.9

Table 9. The performance of each model with each type of prompts on control-flow completion. Syntax-aware truncation is used for post-processing. The most effective prompt type for each model is highlighted in **bold**.

	L2R	PSM	SPM	IPF	1S
GPT-3.5 (175B)	-	-	-	-	**37.5**
GPT-4	-	-	-	-	**55.2**
CodeGen-350M	25.0	-	**26.1**	17.6	-
CodeGen-2B	32.4	-	**32.9**	25.1	-
CodeGen-6B	33.1	-	**34.8**	25.9	-
CodeGen-16B	34.7	-	**35.7**	27.9	-
InCoder-1B	19.6	22.9	**24.4**	11.5	-
InCoder-6B	23.6	28.2	**29.0**	14.9	-
CodeLLaMa-7B	43.1	25.8	**53.6**	40.6	-
CodeLLaMa-13B	45.1	27.3	**57.2**	46.2	-
CodeLLaMa-34B	48.0	-	**54.0**	51.5	-
StarCoder (15.5B)	43.4	**54.5**	53.7	37.4	-
DeepSeekCoder-1.3B	42.6	**54.1**	52.5	35.1	-
DeepSeekCoder-6.7B	50.4	**65.8**	63.8	51.4	-
DeepSeekCoder-33B	55.7	**71.1**	69.8	58.6	-

A.5. Further Results about Syntax-Aware Truncation

Section 6.2 explores the benefits of syntax-aware truncation with algorithmic block completion tasks. This section extends the results in Table 3 to encompass all tasks in SAFIM. Additionally, we also show the selected prompt for each model, determined by the highest pass@1 rate post-truncation. The results are shown in Table 11, Table 12, and Table 13.

We find that syntax-aware truncation consistently improves the pass@1 rate and reduces compilation errors in both algorithmic block completion and control-flow expression completion, highlighting the effectiveness of syntax-aware truncation.

However, in API function call completion, which involves generation of function invocation expressions or statements, LLMs typically produce error-free code without truncation, as these code segments are typically short and naturally segmented

Table 10. The performance of each model with each type of prompts on API function call completion. Syntax-aware truncation is used for post-processing. The most effective prompt type for each model is highlighted in **bold**.

	L2R	PSM	SPM	IPF	1S
GPT-3.5 (175B)	-	-	-	-	**53.9**
GPT-4	-	-	-	-	**62.6**
CodeGen-350M	23.5	-	**26.5**	9.7	-
CodeGen-2B	30.3	-	**32.3**	10.3	-
CodeGen-6B	25.5	-	**27.7**	13.5	-
CodeGen-16B	**31.3**	-	31.3	16.8	-
InCoder-1B	38.4	**43.9**	43.9	13.5	-
InCoder-6B	41.0	**48.1**	47.1	16.5	-
CodeLLaMa-7B	**48.7**	37.1	46.8	21.6	-
CodeLLaMa-13B	50.3	39.0	**59.7**	39.0	-
CodeLLaMa-34B	50.6	-	47.7	**56.5**	-
StarCoder (15.5B)	50.6	**68.1**	65.2	44.5	-
DeepSeekCoder-1.3B	45.8	**62.6**	51.9	11.9	-
DeepSeekCoder-6.7B	52.3	**69.7**	60.0	52.3	-
DeepSeekCoder-33B	45.5	**75.2**	64.5	50.6	-

Table 11. Comparison of model performance with and without our syntax-aware truncation algorithm in the post-processing phase. This table presents two numbers for each model evaluated: **Pass@1** and **CErr%**, as well as the prompt selected to evaluate each model.

	No Trunc.		Syntax Trunc.		
	Pass@1	CErr%	Pass@1	CErr%	Prompt
GPT-3.5 (175B)	28.7	25.3	31.2	17.0	1S
GPT-4 (> 220B)	41.7	25.4	42.1	22.9	1S
CodeGen-16B	0.0	99.9	25.9	17.9	SPM
InCoder-6B	21.8	25.7	25.2	13.2	PSM
CodeLLaMa-13B	16.4	64.6	41.4	10.9	SPM
CodeLLaMa-34B	1.0	94.5	38.5	14.7	SPM
StarCoder (15.5B)	42.7	14.3	44.1	9.5	SPM
DeepSeekCoder-33B	59.7	8.0	60.8	4.0	PSM

Table 12. Comparison of model performance with and without our syntax-aware truncation algorithm in the post-processing phase on control-flow expression completion. This table presents two numbers for each model evaluated: **Pass@1** and **CErr%**, as well as the prompt selected to evaluate each model.

	No Trunc.		Syntax Trunc.		
	Pass@1	CErr%	Pass@1	CErr%	Prompt
GPT-3.5 (175B)	37.4	19.7	37.5	19.5	1S
GPT-4 (> 220B)	55.2	21.8	55.2	21.9	1S
CodeGen-16B	0.0	99.9	35.7	14.6	SPM
InCoder-6B	10.4	62.1	28.2	11.0	PSM
CodeLLaMa-13B	27.8	54.8	57.2	2.3	SPM
CodeLLaMa-34B	0.3	98.6	54.0	2.7	SPM
StarCoder (15.5B)	51.8	9.8	54.5	6.0	PSM
DeepSeekCoder-33B	70.3	2.8	71.1	1.1	PSM

Table 13. Comparison of model performance with and without our syntax-aware truncation algorithm in the post-processing phase on API function call completion. This table presents two numbers for each model evaluated: **Pass@1** and **CErr%**, as well as the prompt selected to evaluate each model.

	No Trunc.		**Syntax Trunc.**		
	Pass@1	**CErr%**	**Pass@1**	**CErr%**	**Prompt**
GPT-3.5 (175B)	44.2	0.0	53.9	0.0	1S
GPT-4 (> 220B)	57.4	0.0	62.6	0.0	1S
CodeGen-16B	0.0	99.9	31.3	1.9	SPM
InCoder-6B	23.9	0.0	48.1	0.0	PSM
CodeLLaMa-13B	33.5	0.0	59.7	0.0	SPM
CodeLLaMa-34B	11.9	0.0	56.5	0.6	IPF
StarCoder (15.5B)	65.8	0.3	68.1	0.3	PSM
DeepSeekCoder-33B	72.3	0.0	75.2	0.0	PSM

Table 14. Summary of evaluated models, highlighting data cutoff dates, open-source status (OS), and pretraining objectives. Dates in red indicate overlap between the model's pretraining data and the SAFIM benchmark in date range (post-April 2022). Data cutoff dates for InCoder are estimated based on their initial paper draft publication dates. The OS column denotes open-source availability ($\surd$ for yes, $\times$ for no), and the FIM column indicates models pretrained with FIM objectives and support for sentinel tokens in FIM inference.

	#Params	**Data Cutoff**	**OS**	**FIM**
GPT-4	-	Sept 2021	$\times$	$\times$
CodeGen	350M/2B/6B/16B	Oct 2021	$\surd$	$\times$
InCoder	1.3B/6.7B	$\leq$ Mar 2022	$\surd$	$\surd$
CodeLLaMa	7B/13B	Jul 2022	$\surd$	$\surd$
CodeLLaMa	34B	Jul 2022	$\surd$	$\times$
StarCoder	15.5B	Mar 2022	$\surd$	$\surd$
DeepSeekCoder	1.3B/6.7B/33B	Feb 2023	$\surd$	$\surd$
Mixtral	46.7B (8x7B)	$\leq$ Sep 2023	$\surd$	$\times$
Phi	1.3B/2.7B	Mar 2022	$\surd$	$\times$
WizardCoder	1B/3B/15B	Mar 2022	$\surd$	$\surd$
WizardCoder	33B	Feb 2023	$\surd$	$\surd$
Magicoder	6.7B	Feb 2023	$\surd$	$\surd$

with line breaks. That said, syntax-aware truncation becomes crucial for models and prompts lacking explicit stop signals, such as SPM for CodeGen and IPF for CodeLLaMa-34B. In these scenarios, our truncation method allows fair comparisons across various models by standardizing the completion endpoint.

A.6. Evaluation Results of More LLMs

In this section, we expand our evaluation to include additional LLMs: Mixtral (Jiang et al., 2024), Phi (Gunasekar et al., 2023), WizardCoder (Luo et al., 2023), and Magicoder (Wei et al., 2023).

Table 14 provides the details of the additional models. Mixtral-8x7B, a sparse mixture-of-experts (MoE) model, uses a router to select two expert 7B models for each inference. The Phi models are small LLMs pretrained using distilled data (synthetic data generated by a teacher LLM). WizardCoder and Magicoder are initialized with base models and then finetuned on distilled data. Specifically, WizardCoder variants (15B and 33B) use StarCoder and DeepSeekCoder-33B as their respective base models, while Magicoder-S-DS-6.7B is finetuned from DeepSeekCoder-6.7B. Notably, these models are classified as FIM models because their inherited vocabulary supports FIM special tokens, despite the finetuning process not directly engaging with FIM tasks.

Table 15 shows our experimental results, which yield the following insights:

- Mixtral-8x7B achieves performance comparable to CodeLLaMa-7B. Given that Mixtral is not specialized for coding,

Table 15. Pass@1 of various models on the SAFIM benchmark, showing their performance in algorithmic block completion (Algo.), control-flow completion (Control), and API function call completion (API).

	Algo.	Control	API	Avg
GPT-3.5 (175B)	31.2	37.5	53.9	40.9
GPT-4 (> 220B)	42.1	55.2	62.6	53.3
CodeGen-350M	16.3	26.1	26.5	22.9
CodeGen-2B	23.5	32.9	32.3	29.5
CodeGen-6B	23.6	34.8	27.7	28.7
CodeGen-16B	25.9	35.7	31.3	31.0
InCoder-1B	21.1	22.9	43.9	29.3
InCoder-6B	25.2	28.2	48.1	33.8
CodeLLaMa-7B	34.7	53.6	46.8	45.0
CodeLLaMa-13B	41.4	57.2	59.7	52.8
CodeLLaMa-34B	38.5	54.0	56.5	49.7
StarCoder (15.5B)	44.1	54.5	68.1	55.5
DeepSeekCoder-1.3B	41.2	54.1	62.6	52.6
DeepSeekCoder-6.7B	54.7	65.8	69.7	63.4
DeepSeekCoder-33B	**60.8**	**71.1**	**75.2**	**69.0**
Mixtral-8x7B	33.7	50.3	58.4	47.5
Phi-1.5 (1.3B)	19.0	29.9	27.7	25.5
Phi-2 (2.7B)	23.8	34.8	22.3	26.9
WizardCoder-1B	28.1	40.0	57.4	41.8
WizardCoder-3B	34.4	46.3	65.2	48.6
WizardCoder-15B	41.0	52.6	71.0	54.8
WizardCoder-33B	49.5	66.3	74.5	63.4
Magicoder-S-DS-6.7B	41.5	62.3	65.5	56.4

its comparable performance to 7B code LLMs shows the effectiveness of MoE. Typically, general-purpose LLMs like GPT-3.5, GPT-4, and Mixtral need more parameters to match the performance of specialized code LLMs.

- Models Pretrained on Distilled Data (Phi-1.5 and Phi-2) exhibit good performance considering their tiny sizes, but they don't reach the high standards set by their HumanEval results. This difference underscores the SAFIM benchmark's diversity and complexity compared to HumanEval.
- Models Finetuned on Distilled Data shows a slight drop in performance compared to their FIM-pretrained base models (e.g., WizardCoder-15B vs. StarCoder, WizardCoder-33B vs. DeepSeekCoder-33B, Magicoder-S-DS-6.7B vs. DeepSeekCoder-6.7B). The performance drop stems from the left-to-right finetuning on distilled data, which lacks FIM objectives, thereby harming the models' proficiency in FIM tasks.

These additional findings further reinforce our original conclusion: the pretraining methodology significantly influences the performance of code LLMs.

A.7. Result Analysis by Programming Languages

Table 16 shows the average pass@1 rate for each LLM in our SAFIM benchmark, broken down by programming language and averaged on three completion tasks. Our analysis reveals that:

LLMs exhibit higher success rates in Java and C#, likely due to the verbosity of these languages, which leads to more predictable coding patterns. Conversely, completion in C++ and Python is more challenging, due to the more concise and less predictable coding styles prevalent among developers. As we discussed in Appendix A.2, the SAFIM benchmark consist of different programming languages written by different developers, so the results are affected by intrinsic variability in coding styles across PLs.

Table 16. Average pass@1 of various models on the three tasks in SAFIM, showing their results in different programming languages.

	C++	Java	Python	C#	Avg
GPT-3.5 (175B)	39.3	54.2	29.5	40.5	40.9
GPT-4 (> 220B)	49.4	63.3	42.7	54.6	53.3
CodeGen-350M	23.1	33.6	18.7	19.9	22.9
CodeGen-2B	27.9	43.4	24.1	28.9	29.5
CodeGen-6B	30.3	44.6	21.2	26.4	28.7
CodeGen-16B	35.5	46.5	20.7	30.2	31.0
InCoder-1B	21.3	35.9	35.3	32.2	29.3
InCoder-6B	26.2	41.4	40.5	32.4	33.8
CodeLLaMa-7B	33.6	56.1	40.6	47.9	45.0
CodeLLaMa-13B	45.8	60.2	52.5	64.7	52.8
CodeLLaMa-34B	43.3	59.9	49.0	62.3	49.7
StarCoder (15.5B)	52.0	63.9	59.5	54.7	55.5
DeepSeekCoder-1.3B	44.7	61.3	57.7	55.5	52.6
DeepSeekCoder-6.7B	57.6	70.3	67.5	70.4	63.4
DeepSeekCoder-33B	65.8	75.1	72.5	74.7	69.0
Mixtral-8x7B	42.0	58.0	43.0	56.9	47.5
Phi-1.5 (1.3B)	18.8	30.1	26.7	18.9	25.5
Phi-2 (2.7B)	22.6	36.3	23.8	23.3	26.9
WizardCoder-1B	34.1	48.7	44.5	39.3	41.8
WizardCoder-3B	43.3	57.0	53.8	49.8	48.6
WizardCoder-15B	51.7	61.7	59.2	52.1	54.8
WizardCoder-33B	61.1	70.8	67.2	55.0	63.4
Magicoder-S-DS-6.7B	50.8	65.4	57.3	61.4	56.4

Despite the language-dependent variability, the relative rankings of LLMs stay mostly consistent. This underscores the robustness of the SAFIM benchmark and supports our decision to report *micro-averaged* performance metrics in our study.

A.8. Case Study

This section presents a case study of the algorithmic block completion task from SAFIM (`task_id: block_completion_008121`). Two similar-performing models, InCoder-6B and CodeCen-16B (achieving pass rates at 25.2% and 25.9% respectively), are compared.

The case originates from problem 1678B1 on Codeforces[5]; given the problem description, the task is to fill in the `# TODO: Your code here` part of the provided code:

```
t=int(input(""))
for z in range(t):
    n=int(input(""))
    a=input("")
    s=[]
    for i in range(0,len(a)-1,2):
        # TODO: Your code here
    b=s.count('10')
    c=s.count('01')
    print(b+c)
```

[5] https://codeforces.com/problemset/problem/1678/B1

The ground truth involves appending two characters from the string `a` to the list `s` in each loop iteration (we added 8 spaces at the beginning for clarity):

```
        ab=a[i]+a[i+1]
        s.append(ab)
```

Using the PSM prompt, InCoder-6B successfully generates a valid Python completion using Python string slicing:

```
        s.append(a[i:i+2])
```

CodeGen-16B uses the SPM prompt. Note that CodeGen-16B lacks EOS token support, leading to generation of extra code followed by infinite output generation unless truncated:

```
        s.append(a[i])
    s.append(a[len(a)-1])
    s=s[::-1]
    b=s.count('10')
    c=s.count('01')
    print(b+c)

# [infinite empty lines]
```

Applying syntax-aware truncation, we keep only the relevant block completion `s.append(a[i])`. Unfortunately, this still yields an incorrect solution, leading to an outcome of "wrong answer" in evaluation.

This case study underscores the significance of syntax-aware truncation and highlights the behavior of different models.

A.9. Further Analysis on Data Contamination

SAFIM is sourced from Codeforces contests and Github code commits created between April 1, 2022 and January 1, 2023. This period, unfortunately, overlaps with the pretraining data of CodeLLaMa and DeepSeekCoder. To analyze the potential influence of data contamination on our evaluation results, we create an new dataset for the algorithmic block completion task based on Codeforces contests from April 1, 2023, to January 31, 2024, without any overlap with the training data of these models. Then we evaluate each of these models and StarCoder, on both datasets. The findings are shown in Table 17. We also visualize each model's performance across various months in the new test dataset in Figure 5.

Table 17. Pass@1 of each model on two versions of algorithmic block completion, including the **original** version (Apr 2022 - Jan 2023) and the **new** version (Apr 2023 - Jan 2024). Numbers in red indicate overlap between the model's pretraining data and the test dataset in date range. The Δ column shows the pass@1 change between the original and the new test datasets.

	Data Cutoff	**Original**	**New**	Δ
StarCoder	Mar 2022	44.1	46.7	+2.56
CodeLLaMa-7B	Jul 2022	34.7	32.7	-1.95
CodeLLaMa-13B	Jul 2022	41.4	45.8	+4.40
CodeLLaMa-34B	Jul 2022	38.5	43.8	+5.29
DeepSeekCoder-1.3B	Feb 2023	41.2	46.1	+4.87
DeepSeekCoder-6.7B	Feb 2023	54.7	58.4	+3.65
DeepSeekCoder-33B	Feb 2023	60.8	61.7	+0.91

Based on Table 17, for the new test data, without any overlap with the models' training date ranges, no significant performance decrease is noticed compared to the original dataset, which had a date range overlap. Figure 5 also shows stable performance across the timeline for all models, without a noticeable decline on newer questions for CodeLLaMa or DeepSeekCoder. These findings suggest that while vigilance against data contamination is prudent, the difference in cutoff dates has a negligible impact on our current evaluation results.

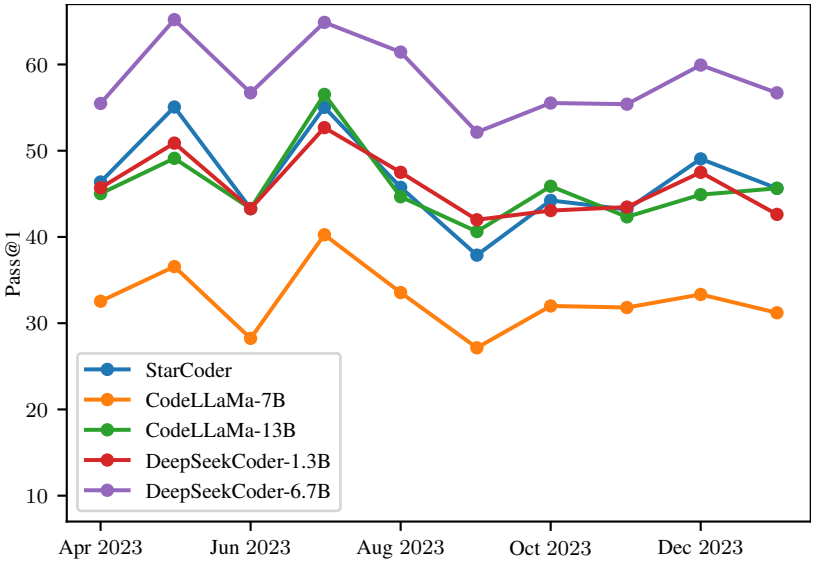

Figure 5. Pass@1 scores for each model on algorithmic block completion across various months in the **new** test dataset.