From Output to Evaluation: Does Raw Instruction-Tuned Code LLMs Output Suffice for Fill-in-the-Middle Code Generation?

Anonymous ACL submission

Abstract

Post-processing is crucial for the automatic evaluation of LLMs in fill-in-the-middle (FIM) code generation due to the frequent presence of extraneous code in raw outputs. This extraneous generation suggests a lack of awareness regarding output boundaries, requiring truncation for effective evaluation. The determination of an optimal truncation strategy, however, often proves intricate, particularly when the scope includes several programming languages. This study investigates the necessity of post-processing instruction-tuned LLM outputs. Our findings reveal that supervised fine-tuning significantly enhances FIM code generation, enabling LLMs to generate code that seamlessly integrates with the surrounding context. Evaluating our fine-tuned `Qwen2.5-Coder` (base and instruct) models on HumanEval Infilling and SAFIM benchmarks demonstrates improved performances without post-processing, especially when the *middle* consists of complete lines. However, post-processing of the LLM outputs remains necessary when the *middle* is a random span of code.

1 Introduction

The iterative process of coding, involving frequent edits and insertions (Bavarian et al., 2022; Fried et al., 2023), establishes Fill-in-the-Middle (FIM) code generation a prevalent task in code completion. Models tackling this must generate the missing code segment conditioned on both the preceding (left) and succeeding (right) context. A key challenge in FIM lies in seamlessly integrating the generated *middle* with the subsequent code while maintaining both structure and meaning – a non-trivial learning objective for models. Consequently, raw model outputs often undergo rule-based post-processing to remove extraneous content. As shown in Table 1, two widely used FIM code generation evaluation benchmarks employ specific truncation rules that may not generalize to real-world FIM scenarios with arbitrary left and right contexts. Furthermore, such truncation strategies often fail to account for alternative, yet valid, ways of generating the missing code. For instance, as illustrated in Figure 1, a single-line infilling task might expect one line as a solution, but an LLM could generate five lines that perfectly match the surrounding context. In this case, truncating the generated middle to a single line would incorrectly mark it as a failure. Given the advancements in code LLMs, a crucial question emerges: do modern code LLMs naturally know when to stop generating given any arbitrary left and right context, thereby eliminating the need for post-processing techniques like truncation?

The existing body of work (Bavarian et al., 2022; Fried et al., 2023; Nguyen et al., 2023; Zheng et al., 2024) predominantly examines *base* LLMs, trained on massive amounts of data to understand language patterns and generate consistent output. These models acquire Fill-in-the-Middle (FIM) capabilities by learning from reordered prefix-middle-suffix sequences, created via random splits of the training data. The purpose of this reordering is to allow the LLM to auto-regressively predict the middle segment, conditioned on both the left and right contexts as past information. In contrast to base LLMs, we posit that *instruction-tuned* LLMs are better equipped for FIM generation due to their customized nature and their inherent capacity to adhere to instructions. Our primary motivation for focusing on instruction-tuned LLMs stems from the objective to avoid the expensive pre-training (or their continuation) required by models like those in (Bavarian et al., 2022), which demonstrated that fine-tuning with FIM does not achieve the same performance as pre-training with FIM.

This study investigates the necessity of post-processing instruction-tuned LLM outputs for FIM code generation. Our empirical analysis reveal that the raw outputs of off-the-shelf instruction-

Dataset	FIM Type	Truncation Rule for Output
HumanEval Infilling	Single-line	Truncate output to one line.
	Multi-line	Truncate output to match ground truth line count.
	Random-span	Truncate output if overlaps with prefix and suffix.
SAFIM	Algorithm-block	Truncate output to one line.
	Control-flow	Truncate output to match ground truth program structure.
	API function call	Truncate output after first closing parenthesis.

Table 1: Truncation strategy used in two popular FIM benchmarks.

Canonical solution

```python
def even_odd_count(num):
    """[docstring truncated]"""
    even_count = 0
    odd_count = 0
    for i in str(abs(num)):
        if int(i)%2==0:
            even_count +=1
        else:
            odd_count +=1
    return (even_count, odd_count)
```

FIM by our finetuned model

```python
def even_odd_count(num):
    """[docstring truncated]"""
    even_count = 0
    odd_count = 0
    if num < 0:
        num = str(num)[1:]
    else:
        num = str(num)
    for i in num:
        if int(i)%2==0:
            even_count +=1
        else:
            odd_count +=1
    return (even_count, odd_count)
```

Figure 1: **Left**: An example of a single-line infilling task (highlighted in red) from the HumanEval Infilling benchmark. **Right**: A fill-in-the-middle generation produced by our fine-tuned Qwen2.5-Coder-7B-Instruct model.

tuned LLMs often require editing. Consequently, we fine-tuned both base and instruct versions of `Qwen2.5-Coder`. Our findings demonstrate that these fine-tuned models can produce outputs that do not require any post-processing when the middle code segments consist of whole lines. In fact, applying any preset, heuristic-based post-processing in such cases actually leads to incorrect middle outputs. However, when middle segments comprise partial lines, it becomes necessary to truncate overlapping code segments. Based on our findings, we offer straightforward post-processing recommendations for LLM-generated middle code segments.

In summary, we contribute the followings:

1. We show that off-the-shelf instruction-tuned LLMs require post-processing for effective FIM code generation and exhibit suboptimal performance due to a lack of task-specific fine-tuning or optimization.
2. We demonstrate that lightweight fine-tuning significantly boosts LLM performance for FIM generation. Interestingly, when the *middle* code consists of complete lines, the raw outputs from these fine-tuned models achieve better automatic evaluation scores than post-processed outputs, meaning no further editing is needed. However, if the *middle* includes partial lines, post-processing is still required.

2 Instruction-tuning of LLMs for Fill-in-the-Middle Code Generation

We investigate the FIM code generation accuracy of state-of-the-art instruction-tuned code LLMs by prompting them with instructions, as illustrated in Figure 3. This prompting method is consistent with their standard usage for code generation. Our findings in subsection 3.3 reveal that instruction-tuned LLMs perform suboptimally, even after their outputs undergo dataset-specific post-processing. Building on this observation, we further investigate if lightweight supervised fine-tuning can empower code LLMs for improved FIM generation.

To achieve this, we created a training dataset of instruction-response pairs using an LLM. First, we collected a set of Python functions from GitHub, following the data collection pipeline detailed in Wei et al. (2024). This involved a rigorous filtering process: type checking with Pyright, removal of benchmark items, elimination of poorly documented functions, and deduplication. Using these

collected functions, we employed a straightforward approach to generate instruction-response pairs. Specifically, we prompted Mixtral-8x22B (Jiang et al., 2024) with the template shown in Figure 2, asking it to split each function into prefix, middle, and suffix according to one of five strategies outlined in the prompt. After generating the prefix, middle, and suffix, we verified that their concatenation reconstructs the original function. At the end, we collected ≈1M instruction-response pairs that we used to finetune code LLMs.

3 Experiments

3.1 Setup

Training & Inference Setup We fine-tuned the 7B, 14B, and 32B parameter base and instruct versions of `Qwen2.5-Coder`. The finetuning spanned 5000 steps on NVIDIA H100-80GB GPUs, leveraging the AdamW optimizer (Kingma and Ba, 2015) with a batch size of 256 and a maximum sequence length of 4096 tokens. We initialized the learning rate at 5e-6 and applied a CosineAnnealing scheduler with a 10% warmup. We utilized tensor parallelism and BF16 precision to accelerate the training process. For evaluation, we utilized the final training checkpoint, and during inference, we employed greedy decoding.

Evaluation Benchmarks and Metrics We evaluated models using two FIM code generation benchmarks: HumanEval Infilling (Bavarian et al., 2022) and SAFIM (Gong et al., 2024). The HumanEval Infilling benchmark features three distinct tasks: Single-line, Multi-line, and Random span infilling. We provide the post-processing functions for these tasks in Figure 4. In contrast, SAFIM is a syntax-aware FIM benchmark, consisting of tasks focused on algorithm block, control flow, and API function call completion. For both benchmarks, we present results based on the standard pass@1 metric.[1]

3.2 Research Questions

We aim to address the following questions.

1. What is the out-of-the-box effectiveness of instruction-tuned code LLMs for fill-in-the-middle (FIM) code generation?
2. Can supervised fine-tuning significantly improve the FIM generation accuracy of code LLMs? How does finetuning impact *base* and *instruct* version of LLM? Furthermore, what are the effects of such fine-tuning on *base* vs. *instruct*-tuned LLMs?
3. Are the raw outputs of fine-tuned LLMs sufficiently effective for automatic evaluation?

3.3 Results

The results are presented in Table 2. We consistently observed a few performance trends.

Instruction-tuned LLMs are not ready out-of-the-box The `Qwen2.5-Coder-Instruct` models consistently perform poorly on both benchmarks, particularly on the SAFIM and random-span HumanEval infilling tasks. Their low accuracies clearly indicate that these models cannot be effectively used off-the-shelf in FIM generation.

Supervised finetuning (SFT) is a major leap for FIM instruction-tuning The overall results clearly indicate a significant performance boost with SFT of `Qwen2.5-Coder-Instruct` models. The average performance of the 7B and 14B models doubled, while the 32B models saw an impressive 40-50% improvement compared to their off-the-shelf counterparts.

Sample efficiency of base vs. instruct LLMs The average pass@1 accuracies across both benchmarks suggest that tuning instruction-following LLMs yields slightly better performance.

Raw outputs of finetuned LLMs are effective From Table 2, we observe that post-processing consistently lowers accuracies for single-line and multi-line infilling tasks in HumanEval benchmark as shown in Figure 1. However, for random-span infilling, raw LLM outputs do require editing, which is evident from the resulting improved performance after post-processing. We see a similar pattern for the SAFIM benchmark.

Based on our observations and experiment results, we recommend *post-processing to remove overlapping code segments found between the prefix and the generated middle, and similarly between the middle and the suffix*. This is our standard approach for all infilling tasks in this work.

3.4 Other Findings

Our experiments showed that generating multiple FIM samples from a single Python function (resulting in 5M instruct-response pairs) didn't significantly improve supervised fine-tuning. Thus, we

[1]For SAFIM evaluation, we used the authors released code, available at `https://github.com/gonglinyuan/safim`.

Model	Post-Proc.	HumanEval Infilling				SAFIM (Python)			
		SL	ML	RS	Avg.	Algo.	Control	API	Avg.
Qwen2.5-Coder-7B-Instruct	✗	47.1	22.4	0.9	23.5	3.5	0	0	1.2
	✓	53.3	24.3	12.4	30.0	3.98	14.4	22.1	13.5
Qwen2.5-Coder-7B	✗	89.3	68.2	31.9	63.1	28.5	29.1	69.6	42.4
	✓	85.7	61.3	**43.7**	63.6	28.3	**36.7**	69.1	44.7
Qwen2.5-Coder-7B-Instruct	✗	**91.6**	**67.4**	34.2	**64.4**	30.7	30.1	**72.4**	44.4
	✓	88.7	61.0	43.0	64.2	**30.8**	36.0	69.6	**45.5**
Qwen2.5-Coder-14B-Instruct	✗	43.9	27.8	3.4	25.0	7.2	0	2.2	3.1
	✓	46.7	27.4	12.2	28.8	12.7	15.1	43.7	23.8
Qwen2.5-Coder-14B	✗	87.0	72.7	36.4	65.4	23.7	29.6	74.0	42.4
	✓	84.9	63.9	46.3	65.0	23.7	35.5	72.9	44.0
Qwen2.5-Coder-14B-Instruct	✗	**91.7**	**73.4**	37.6	**67.6**	29.4	33.5	**76.8**	46.6
	✓	88.8	64.3	**46.8**	66.6	**29.8**	**38.7**	75.1	**47.9**
Qwen2.5-Coder-32B-Instruct	✗	74.7	47.7	5.9	42.8	19	1.7	10	10.2
	✓	77.0	56.9	22.7	52.2	19.5	25.3	45.9	30.2
Qwen2.5-Coder-32B	✗	93.9	75.3	36.6	68.6	31.4	36.1	74.6	47.4
	✓	89.7	66.8	47.9	68.1	**31.8**	**41.7**	75.1	**49.5**
Qwen2.5-Coder-32B-Instruct	✗	**94.8**	**76.5**	37.6	**69.6**	31.6	36.7	**76.2**	48.2
	✓	91.4	68.7	**48.0**	69.4	31.6	**41.7**	74.6	49.3

Table 2: Performance comparison of Qwen2.5-Coder-Instruct models across three different sizes. SL, ML, and RS indicate "single-line", "multi-line", and "random-span" infilling tasks, respectively. Highlighted rows show our finetuned models' performances. Bold indicates the highest performances for each model groups.

suggest future work prioritize diversity in Python functions over generating many samples from one.

Additionally, fine-tuning models for more than roughly one epoch degraded performance on downstream FIM tasks. Therefore, we recommend using a larger collection of training samples, but with only a single training iteration over them.

4 Related Work

Bavarian et al. (2022) presented a foundational approach to training large language models (LLMs) for FIM code generation, marking a significant first step in this area. Their core innovation involved segmenting unlabeled code into three distinct parts and rearranging those segments to create training sequences. This pioneering strategy proved highly influential, shaping nearly all subsequent research in FIM code generation (Fried et al., 2023; Zheng et al., 2024; Wu et al., 2024; Sagtani et al., 2025).

In contrast to this dominant paradigm, Nguyen et al. (2023) introduced an alternative method. They trained two separate language models, each generating code in an opposing direction: one from left-to-right and the other from right-to-left. The FIM task was then solved by having these independently generated segments converge and "meet" in the middle. More recently, Ding et al. (2024) departed from these approaches, showing improvements by adopting a planning and lookahead based approach to language generation.

To the best of our knowledge, the existing body of work in FIM code generation has primarily focused on either pre-training base LLMs or exploring alternative architectures and training methodologies. A significant gap in the existing literature is the lack of focused investigation into the intrinsic FIM capabilities of instruction-tuned LLMs – models already adapted for following instructions. Our work aims to bridge this gap by specifically evaluating and enhancing the FIM performance of models that have already been fine-tuned for instruction following, offering a novel perspective on leveraging these readily available and powerful models for this crucial code completion task.

5 Conclusion

Supervised fine-tuning considerably enhances the generation of code that can be evaluated directly, significantly diminishing the reliance on intricate post-processing. Our fine-tuned `Qwen2.5-Coder` models achieve substantial performance gains on the HumanEval Infilling and SAFIM benchmarks. This underscores targeted fine-tuning as a route to directly utilize raw LLM outputs.

6 Limitations

First, our evaluation is primarily focused on the Python programming language, as reflected in the HumanEval Infilling and SAFIM benchmarks. The generalizability of our findings to other programming languages, which may exhibit different syntactic structures and code completion patterns, remains an open question. Future work should explore the application of our fine-tuning approach and the resulting reduction in post-processing needs across a more diverse set of languages.

Second, the instruction fine-tuning data we created, while effective, was generated using a specific LLM (`Mixtral-8x22B`) and a defined set of splitting strategies. The quality and diversity of this synthetic data directly influence the performance of our fine-tuned models. Exploring alternative data generation methods, incorporating human-annotated FIM examples, or scaling the size and diversity of the training data could potentially lead to further improvements in FIM generation and a more robust elimination of post-processing requirements.

Finally, our evaluation focused on specific benchmarks designed for FIM code generation. While these benchmarks are widely used, they represent a specific type of FIM task. The performance of our fine-tuned models and the necessity of post-processing might vary in more complex or less constrained FIM scenarios encountered in real-world code editing environments. Further investigation into the applicability of our findings to such diverse scenarios is warranted.

References

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*.

Yifeng Ding, Hantian Ding, Shiqi Wang, Qing Sun, Varun Kumar, and Zijian Wang. 2024. Horizon-length prediction: Advancing fill-in-the-middle capabilities for code generation with lookahead planning. *arXiv preprint arXiv:2410.03103*.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations*.

Linyuan Gong, Sida Wang, Mostafa Elhoushi, and Alvin Cheung. 2024. Evaluation of LLMs on syntax-aware code fill-in-the-middle tasks. In *Forty-first International Conference on Machine Learning*.

Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of experts.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Anh Tuan Nguyen, Nikos Karampatziakis, and Weizhu Chen. 2023. Meet in the middle: A new pre-training paradigm. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Hitesh Sagtani, Rishabh Mehrotra, and Beyang Liu. 2025. Improving fim code completions via context & curriculum based learning. In *Proceedings of the Eighteenth ACM International Conference on Web Search and Data Mining*, pages 801–810.

Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro Von Werra, Arjun Guha, and LINGMING ZHANG. 2024. Selfcodealign: Self-alignment for code generation. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024. Repoformer: Selective retrieval for repository-level code completion. In *Forty-first International Conference on Machine Learning*.

Lin Zheng, Jianbo Yuan, Zhi Zhang, Hongxia Yang, and Lingpeng Kong. 2024. Self-infilling code generation. In *Forty-first International Conference on Machine Learning*.

Supervised Finetuning Prompt for Fill-in-the-Middle Code Generation

Split the provided Python code into three parts: (1) prefix, (2) middle, and (3) suffix. The split can be made at any character position. The "middle" section should be from one of the following categories.

1. A random span
2. An algorithmic block
3. A control-flow expression
4. An API function call
5. An assignment expression

Note that, when combined, the prefix, middle, and suffix must recreate the original code in its entirety.

The input code is as follows.
```python
{content}
```

Provide 5 examples of prefix, middle, and suffix in the following format. Additionally, label the middle span as one of the five categories listed above.

Example: example_number

Prefix
```python
# your code here
```

Suffix
```python
# your code here
```

Middle
```python
# your code here
```

Label

Figure 2: Prompt template to generate fill-in-the-middle training samples.

Supervised Finetuning Prompt for Fill-in-the-Middle Code Generation

You are given an incomplete code with a prefix and suffix. Your task is to generate the middle section.

Prefix
```python
{prefix}
```

Suffix
```python
{suffix}
```

Middle
```python
# your code here
```

Middle section generation guidelines:

1. The middle section must, when combined with the prefix and suffix, form a complete code without syntax errors. Ensure that the end of the middle section does not overlap with the start of the suffix.
2. Do not include any explanations or notes.

Figure 3: Prompt template for supervised finetuning for fill-in-the-middle code generation.

```python
def single_line_infill_postprocess(completion):
  lines = completion.splitlines()
  for line in lines:
    current_line = line.strip()
    if not current_line:
      continue
    if current_line.startswith("#"):
      continue
    return line
  return ""

def multi_line_infill_postprocess(completion, num_lines):
  assert num_lines > 0
  l = 0
  completion_lines = []
  for line in completion.split("\n"):
    completion_lines.append(line)
    current_line = line.strip()
    if current_line and not current_line.startswith("#"):
      l += 1
      if l == num_lines:
        break
  completion = "\n".join(completion_lines)
  return completion


def remove_overlap_prefix_middle(prefix, middle):
  prefix_len = len(prefix)
  middle_len = len(middle)
  for i in range(min(prefix_len, middle_len), 0, -1):
    if middle.startswith(prefix[-i:]):
      return middle[i:]
  return middle


def remove_overlap_middle_suffix(middle, suffix):
  suffix_len = len(suffix)
  middle_len = len(middle)
  for i in range(min(middle_len, suffix_len), 0, -1):
    if middle.endswith(suffix[:i]):
      return middle[:-i]
  return middle

def random_span_infill_postprocess(completion, prefix, suffix):
  completion = remove_overlap_prefix_middle(prefix, completion)
  completion = remove_overlap_middle_suffix(completion, suffix)
  return completion
```

Figure 4: Post-processing functions for different HumanEval infilling tasks.