Planning-Aware Code Infilling via Horizon-Length Prediction

Yifeng Ding[1*] **Hantian Ding**[2] **Shiqi Wang**[3*] **Qing Sun**[2] **Varun Kumar**[2] **Zijian Wang**[3*]
[1]University of Illinois Urbana-Champaign [2]AWS AI Labs [3]Meta
yifeng6@illinois.edu {dhantian,qinsun,kuvrun}@amazon.com
{tcwangshiqi,zijianwang}@meta.com

Abstract

Fill-in-the-Middle (FIM), or infilling, has become integral to code language models, enabling generation of missing code given both left and right contexts. However, the current FIM training paradigm which performs next-token prediction (NTP) over reordered sequence often leads to models struggling to generate content that aligns well with the surrounding context. We hypothesize that NTP alone is insufficient for models to *learn effective planning* conditioned on the distant right context, a critical factor for successful code infilling. To overcome this, we propose **Horizon-Length Prediction** (**HLP**), a novel training objective that teaches models to predict the number of remaining middle tokens at each step. HLP advances FIM with lookahead planning, enabling models to inherently learn infilling boundaries for arbitrary left and right contexts without relying on dataset-specific post-processing. Our evaluation across different model families and sizes shows that HLP significantly improves FIM performance by up to 24% relatively on diverse benchmarks, across file-level and repository-level. Furthermore, the enhanced planning capability gained through HLP boosts model performance on code reasoning. Importantly, HLP incurs negligible training overhead and no additional inference cost, ensuring its practicality for real-world scenarios.

1 Introduction

Fill-in-the-Middle (FIM), or infilling, has become essential for modern code development, where programmers frequently need to insert or modify code between existing sections rather than writing linearly from start to end (Bavarian et al., 2022; Fried et al., 2023). While large language models have shown remarkable capabilities in code generation, the FIM task poses unique challenges that go beyond traditional left-to-right generation. A model must not only generate code that follows from the preceding context (prefix), but also smoothly connect to the subsequent code (suffix) – a task that requires careful planning and foresight.

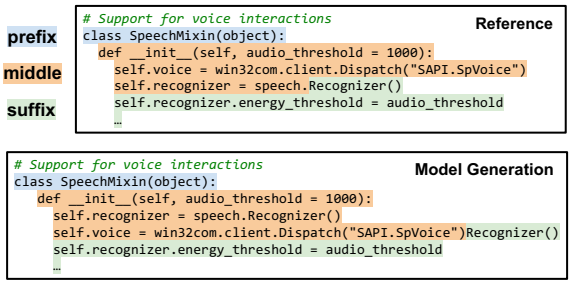


Figure 1: Successful FIM requires planning capabilities. Given prefix and suffix, the model is asked to infill the middle part. Compared with the ground truth, LLM fails to connect to suffix due to lack of planning capability: the last part of the generation needs to connect with the member function `Recognizer()`.

Current approaches to FIM typically reorder the input sequence and apply standard next-token prediction (NTP) training (Lozhkov et al., 2024; Guo et al., 2024; DeepSeek-AI et al., 2024; Hui et al., 2024). However, as illustrated in Figure 1, this methodology has a fundamental limitation: models struggle to maintain coherence over longer sequences and often fail to create smooth transitions to the right context (Bavarian et al., 2022). While existing benchmarks attempt to address this through rule-based post-processing (e.g., truncating generated code based on line count (Zhang et al., 2023; Wu et al., 2024) or program structure (Ding et al., 2023; Gong et al., 2024)), such methods rely on dataset-specific assumptions that do not generalize to real-world scenarios where both left and right contexts can be arbitrary.

We hypothesize that the core challenge lies in the model's limited ability to plan ahead while filling in the middle. Standard next-token prediction training only requires considering one token at a time, but successful code infilling demands awareness of the entire missing section to ensure both local coherence and proper connection to the right context. This planning capability is particularly crucial because generated code must satisfy both syntactic requirements and semantic constraints from both surrounding contexts.

To address this limitation, we propose Horizon-Length Prediction (HLP), a novel training objective that teaches

[*]Work done at AWS AI Labs.

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 32930–32942
November 4-9, 2025

	Post-processing Criteria
RepoEval (Zhang et al., 2023) CrossCodeLongEval (Wu et al., 2024)	Truncate generation to **the same number of lines as in ground truth**.
CrossCodeEval (Ding et al., 2023)	Truncate generation **at the first complete statement**.
SAFIM (Gong et al., 2024)	Stop when **the target program structure in ground truth** is generated.

Table 1: Post-processing criteria used in existing FIM benchmarks. Text in bold denotes restrictive dataset-specific knowledge they employ in evaluation.

models to predict the number of remaining tokens needed to complete the middle section at each generation step. HLP advances NTP by encouraging models to develop awareness of the generation horizon and plan accordingly. Unlike post-processing approaches, HLP is generalizable as it does not rely on any task-specific knowledge. Instead, it strengthens models' innate ability to plan and execute coherent code completions. Our comprehensive evaluation across different models and model sizes demonstrates that HLP significantly improves FIM performance, achieving up to 24% relative improvement on diverse benchmarks at both file-level and repository-level without relying on any dataset-specific post-processing. Furthermore, the enhanced planning capability gained through HLP training also boosts model performance on code reasoning tasks. Importantly, HLP incurs negligible training overhead and no additional inference cost, making it practical for real-world deployment.

Our key contributions are as follows:

- We identify planning capability as a fundamental bottleneck in current FIM approaches and quantitatively demonstrate how post-processing methods in existing benchmarks obscure this critical limitation.
- We propose Horizon-Length Prediction (HLP), a novel training objective that advances fill-in-the-middle capability by teaching models to plan over arbitrarily long horizons, with negligible training and zero inference overhead.
- We demonstrate up to 24% improvement in FIM performance across multiple benchmarks and model families without relying on any post-processing.
- We show that HLP's benefits extend beyond FIM to improve performance on code reasoning tasks, and present various analyses that illuminate the underlying mechanism that enable these improvements.

2 Post-processing for Fill-in-the-Middle

Most existing FIM works rely on post-processing to truncate code completions generated by LLMs for infilling tasks (Gong et al., 2024; Zhang et al., 2023; Ding et al., 2023; Wu et al., 2024). While such post-processing can enhance the FIM performance, we argue that they fundamentally depend on dataset-specific assumptions that make them impractical for real-world scenarios (§2.1). Through evaluation, we show that the performance of FIM existing code models drops significantly without post-processing, suggesting that post-processing conceals the fundamental struggles of models with code filling (§2.2). Furthermore, we show that this limitation stems from models' inability to plan coherent completions given a fixed suffix – a challenge that post-processing alone cannot address.

2.1 Post-processing Requires Task-Specific Knowledge

Post-processing methods adopted by recent FIM benchmarks typically assume a certain completion type and perform rule-based truncation accordingly. Table 1 summarizes the post-processing criteria of four popular FIM benchmarks, highlighting the specific rule used for each dataset. These criteria do not transfer across datasets, nor are they generalizable to FIM in the real-world scenario where both left and right contexts can be arbitrary. Given the complexity of programming languages, developing universally applicable post-processing rules for infilling is infeasible. Instead, models need to learn the intrinsic patterns that make for good completions.

2.2 LLMs Fail to Plan Coherent Completions

To further demonstrate to what extent post-processing conceals LLMs' inability of connecting to suffix, we conduct a comprehensive experiment on SAFIM. We compare FIM performance of four different code LLMs, with or without post-processing. As shown in Table 2, removing post-processing leads to up to a substantial 13.8% Pass@1 drop across all models. This reveals that current models have not truly mastered the fundamental task of generating code that properly connects prefix and suffix contexts. Post-processing creates an illusion of competence by artificially "fixing" problematic generations rather than addressing the core limitation.

2.3 FIM Requires Planning Capability

The challenges in FIM stem from models' inability to plan coherent completions. Consider the example in Figure 1, where a model needs to implement a speech recognition initialization: while the model

	SAFIM				Avg
	Algo	Algo_{v2}	Control	API	
DS-1.3B					
w/ post	43.9	49.2	55.6	62.9	52.9
w/o post	39.8	42.4	52.4	56.1	47.7
rel. diff	**-9.3%**	**-13.8%**	**-5.8%**	**-10.8%**	**-9.9%**
DS-6.7B					
w/ post	54.9	58.9	68.1	71.0	63.2
w/o post	53.4	56.7	66.6	69.0	61.4
rel. diff	**-2.7%**	**-3.7%**	**-2.2%**	**-2.8%**	**-2.8%**
SC2-3B					
w/ post	48.1	53.5	60.1	68.4	57.5
w/o post	45.4	49.7	57.1	61.3	53.4
rel. diff	**-5.6%**	**-7.1%**	**-5.0%**	**-10.4%**	**-7.2%**
SC2-7B					
w/ post	50.4	55.8	62.3	70.3	59.7
w/o post	48.4	53.1	60.4	63.9	56.5
rel. diff	**-4.0%**	**-4.8%**	**-3.0%**	**-9.1%**	**-5.4%**

Table 2: Effect of post-processing techniques for different code LLMs on SAFIM, where "**w/ post**" refers to using post-processing, "**w/o post**" refers to not using post-processing, and "**rel. diff**" refers to the relative performance difference between the two. We follow the same settings used in §4.1.

demonstrates knowledge of the required components (using `Recognizer`), it fails to order them properly. Without careful planning, it prematurely places the `Recognizer` call, leading to both syntactic and semantic errors. Importantly, this type of failure cannot be fixed through post-processing, as truncating the generation would lose essential statements while keeping it results in invalid code.

Such an example illustrates that successful FIM requires not only knowledge of individual components but also the model's ability to plan coherent sequences that smoothly connect to both contexts. The model must reason about the entire completion considering both local coherence and global structure, and that points to a clear need for models to develop intrinsic planning capabilities to succeed at FIM.

3 Horizon-Length Prediction

Given a document $D = \{x_t\}_{t=1}^{T}$ that contains T tokens $x_1, x_2, \cdots, x_T$, existing FIM training scheme can be divided into three steps: (1) Split the document D into three parts: prefix-middle-suffix[1], (2) Construct a new FIM-style document D' by reordering the three parts as prefix-suffix-middle, and (3) Conduct next-token prediction (NTP) training on the document D'.

Specifically, we define the three parts in document D as prefix $= x_{1\cdots P}$, middle $= x_{P+1\cdots P+M}$, and suffix $= x_{P+M+1\cdots T}$. Then, the new document D' will be formatted as follows:

$$\begin{aligned} D' &= \texttt{<PRE>}\ \text{prefix}\ \texttt{<SUF>}\ \text{suffix}\ \texttt{<MID>}\ \text{middle}\ \texttt{<EOI>} \\ &= \texttt{<PRE>}\ x_{1\cdots P}\ \texttt{<SUF>}\ x_{P+M+1\cdots T}\ \texttt{<MID>} \\ &\qquad x_{P+1\cdots P+M}\ \texttt{<EOI>} \\ &\triangleq y_{1\cdots T-M+3}\ x_{P+1\cdots P+M}\ \texttt{<EOI>}, \end{aligned} \tag{1}$$

where the last step re-indexes the leading tokens up until <MID> to $y_{1\cdots T-M+3}$ to focus on the FIM part, as LLMs are expected to start infilling after <MID> token and to end generation with <EOI> token to connect to suffix accurately.

Next-token prediction (NTP) training is conducted on the document D', which aims to minimize the following cross-entropy loss (where P_θ refers to the LLMs being trained):

$$\begin{aligned} L_{NTP} = &- \sum_{t=1}^{T-M+2} \log P_\theta(y_{t+1}|y_{1\cdots t}) \\ &- \sum_{t=1}^{M-1} \log P_\theta(x_{P+t+1}|y_{1\cdots T-M+3}, x_{P+1\cdots P+t}) \\ &- \log P_\theta(\texttt{<EOI>}|y_{1\cdots T-M+3}, x_{P+1\cdots P+M}). \end{aligned} \tag{2}$$

While NTP provides basic FIM capabilities, it has a fundamental limitation: the model only learns to predict one token at a time without developing awareness of the overall horizon. This makes it difficult to plan coherent sequences that properly connect to both contexts.

Horizon-Length Prediction (HLP). To mitigate this issue, we propose horizon-length prediction (HLP) as an auxiliary training objective. As shown in Figure 2, the key idea is to teach models to predict the number of remaining tokens needed to complete the middle section at each generation step. This creates an explicit training signal for planning awareness.

Specifically, at each position t in middle, HLP predicts

$$y_t = \frac{M-t}{M} \in (0,1]. \tag{3}$$

where M is the total length of middle. This normalized value represents the portion of middle that remains to be generated and ensures the target is always within $(0,1]$ interval regardless of the model's context window size. HLP is implemented as a linear layer on top of the transformer model (*i.e.,* `hlp_head` in Figure 2) with weight w_{hlp}, whose input is the hidden state h_t from the last attention layer. The output $w_{hlp}^\top h_t$ is converted to a value between 0 and 1 through a sigmoid layer σ to represent the final prediction. We use L1 loss for HLP:

$$L_{HLP} = \sum_{t=1}^{M} |\sigma(w_{hlp}^\top h_t) - y_t|. \tag{4}$$

[1]We opt to use PSM setting in this work given our base models DeepSeek-Coder and StarCoder2 were both pre-trained with PSM. We expect that our method is generalizable to SPM setting as well.

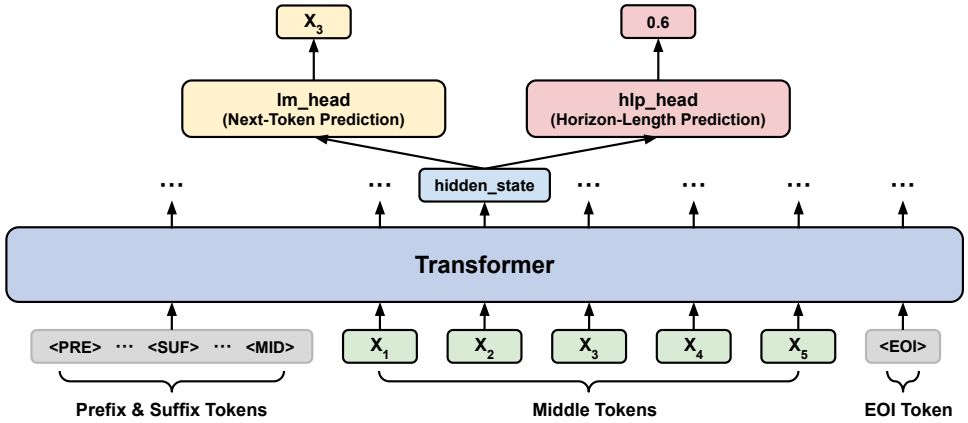


Figure 2: Overview of Horizon-Length Prediction (HLP). In this example, we set the length of middle to five tokens. Following the flow of arrows, we illustrate how the second token of middle (*i.e.,* "x_2") is processed through both next-token prediction objective and horizon-length prediction objective.

The full training objective is a weighted sum of NTP loss and HLP loss:

$$L = L_{NTP} + \lambda \cdot L_{HLP}, \tag{5}$$

where λ is the tunable weight. In experiments, we set $\lambda = 0.1$, which achieves good results across benchmarks empirically. We further study the relationship between the values of λ and the FIM performance of models trained with HLP. As discussed in Appendix A.1, the performance of HLP is robust to different values of λ.

Overhead Analysis. While HLP introduces the additional hlp_head during training, the number of added parameters is < 0.01% of the base model, which incurs *almost zero training overhead*. Furthermore, the additional head will not be used during inference, leading to *zero inference overhead*.

	SAFIM				Average
	Algo	Algo_{v2}	Control	API	
DS-1.3B	39.8	42.4	52.4	56.1	47.7
+ HLP	**41.3**	**46.1**	**53.4**	**59.0**	**50.0***
DS-6.7B	53.4	56.7	66.6	69.0	61.4
+ HLP	**53.5**	**57.4**	**66.9**	**69.7**	**61.9***
SC2-3B	45.4	49.7	57.1	61.3	53.4
+ HLP	**47.2**	**52.1**	**58.7**	**64.5**	**55.6***
SC2-7B	48.4	53.1	60.4	63.9	56.5
+ HLP	**49.4**	**54.5**	**61.8**	**65.8**	**57.9***

Table 3: Pass@1 results of training w/o and w/ HLP for different code LLMs on SAFIM (Gong et al., 2024) computed with greedy decoding. We perform statistical significance tests on "Average" and show that all results are significant. The same notation applies hereafter.

4 Experiments

Training. We conduct continual pre-training on a set of code LLMs of different model families and sizes to validate the effectiveness of HLP. Specifically, DeepSeek-Coder-Base 1.3B/6.7B (Guo et al., 2024) and StarCoder2 3B/7B (Lozhkov et al., 2024) are involved in our experiments. We use AdamW (Loshchilov and Hutter, 2019) as the optimizer with $\beta_1 = 0.9$ and $\beta_2 = 0.95$. We use a cosine learning rate scheduler with a peak learning rate equal to that at the pre-training end. All models are trained for 200K steps with a global batch size of 512.
Dataset. We use a subset of the-stack-v2 (Lozhkov et al., 2024) including Python, Java, C++, and C#. Following existing works (Guo et al., 2024; Lozhkov et al., 2024), FIM rate is set to 0.5. We employ Best-fit Packing (Ding et al., 2024) to group multiple files into each training sequence while masking out cross-file attention. The prefix-middle-suffix split is applied to each file independently rather than the whole training sequence. We conduct controlled experiments for all the studied code LLMs in our experiments. Specifically, we conduct two continual pre-training experiments for each model as follows:

- **NTP**: existing pre-training scheme with next-token prediction (NTP) objective only.
- **NTP + HLP**: our newly proposed objective that incorporates horizon-length prediction (HLP) objective with next-token prediction (NTP) objective.

Throughout this section, we determine the end of generation solely based on `<eoi>` generated by the model without any rule-based post-processing, unless otherwise specified (§4.4). We also conduct statistical analysis: results marked with * are statistically significant ($p < 0.05$) based on paired t-tests.

	CrossCodeEval / CrossCodeLongEval						Average	
	Line		Chunk		Function			
	EM	ES	EM	ES	EM	ES	EM	ES
DS-1.3B	15.23	49.64	22.48	56.40	4.58	33.96	14.10	46.67
+ HLP	**18.99**	**55.47**	**24.32**	**58.77**	**5.12**	**35.25**	**16.14***	**49.83***
DS-6.7B	26.23	62.07	28.90	62.37	**7.50**	**41.42**	20.88	55.29
+ HLP	**27.35**	**63.54**	**30.08**	**63.18**	7.22	40.99	**21.55***	**55.90***
SC2-3B	24.17	59.89	22.20	52.69	6.80	38.13	17.72	50.24
+ HLP	**25.67**	**62.62**	**30.66**	**62.01**	**7.18**	**39.42**	**21.17***	**54.68***
SC2-7B	26.00	61.68	27.14	56.52	7.66	39.54	20.27	52.58
+ HLP	**27.58**	**63.84**	**32.86**	**64.07**	**8.44**	**41.03**	**22.96***	**56.31***

	RepoEval						Average	
	Line		API		Function			
	EM	ES	EM	ES	EM	ES	EM	ES
DS-1.3B	24.50	50.42	18.81	58.15	3.96	29.73	15.76	46.10
+ HLP	**27.25**	**53.45**	**21.81**	**59.79**	**5.93**	**31.92**	**18.33***	**48.39***
DS-6.7B	26.62	52.59	22.69	61.94	7.47	36.24	18.93	50.26
+ HLP	**30.31**	**55.97**	**25.12**	**63.06**	**7.69**	**37.22**	**21.04***	**52.08***
SC2-3B	21.88	46.74	18.81	56.66	4.40	29.99	15.03	44.46
+ HLP	**26.56**	**50.56**	**23.06**	**61.02**	**7.25**	**33.79**	**18.96***	**48.46***
SC2-7B	27.94	51.60	21.56	58.98	6.81	32.80	18.77	47.79
+ HLP	**34.19**	**57.29**	**27.31**	**63.04**	**8.35**	**35.40**	**23.28***	**51.91***

Table 4: Exact Match (EM) and Edit Similarity (ES) results of training w/o and w/ HLP for different code LLMs on CrossCodeEval (Ding et al., 2023), CrossCodeLongEval (Wu et al., 2024), and RepoEval (Zhang et al., 2023) using greedy decoding, following the experimental setting of existing work (Wu et al., 2024). Our evaluation is conducted in "Retrieval" mode, where evaluation prompts are constructed by prepending the retrieved cross-file context to the current file, to show the performance of repository-level cross-file code completion.

4.1 Syntax-Aware Multilingual Code FIM

We use SAFIM (Gong et al., 2024), a syntax-aware and multilingual code Fill-in-the-Middle benchmark, to evaluate the effectiveness of HLP. SAFIM focuses on syntax-aware completions of program structures, covering algorithmic block (*i.e.,* Algo and Algo_{v2}), control-flow expression (*i.e.,* Control), and API function call (*i.e.,* API). It consists of 17,720 examples from four different programming languages, including Python, Java, C++, and C#. SAFIM employs execution-based evaluation and reports pass@1 as the evaluation metric. As shown in Table 3, compared with NTP only, adding HLP achieves up to 5% improvements on average across all the studied code LLMs. Importantly, the improvement is consistent across languages and program structures, demonstrating HLP's ability to enhance FIM capabilities regardless of language or completion context.

4.2 Repository-Level Cross-File Code FIM

In addition to single-file FIM evaluation with SAFIM, we also evaluate the effectiveness of HLP on repository-level code Fill-in-the-Middle in cross-file scenarios via CrossCodeEval (Ding et al., 2023), CrossCodeLongEval (Wu et al., 2024), and RepoEval (Zhang et al., 2023). CrossCodeEval (Python) and CrossCodeLongEval are two repository-level cross-file benchmarks that leverage more than 1500 raw Python repositories to construct 12,665 examples across line, chunk, and function completion tasks, which are used for a more rigorous evaluation. RepoEval is another repository-level cross-file code completion benchmark

consisting of 3,655 line, API, and function completion tasks created from 32 Python repositories. We follow existing work (Wu et al., 2024) to evaluate the model's FIM performance on these benchmarks and use Exact Match (EM) and Edit Similarity (ES) as our evaluation metrics. As shown in Table 4, adding HLP provides consistent improvements for all models across different benchmarks and completion tasks. Specifically, HLP achieves up to 24% improvements on EM and 9% improvements on ES relatively, showing its significant effectiveness.

	Code Repair	Code Reasoning	
	Defects4J	CRUXEval-I	CRUXEval-O
DS-1.3B	33	42.0	31.0
+ HLP	**39**	**44.7***	**31.8***
DS-6.7B	58	52.1	39.2
+ HLP	**59**	**52.4**	**39.6**
SC2-3B	39	42.8	32.1
+ HLP	**41**	**43.9***	**32.6***
SC2-7B	41	44.4	35.9
+ HLP	**47**	**45.5***	**36.1***

Table 5: Code fixing and reasoning performance of models trained w/o and w/ HLP on Defects4J and CRUXEval. On Defects4J, following the convention, we report the number of patches that passed test suites under greedy decoding. On CRUXEval, we follow the original setting to do sampling with $T=0.2$ and $n=10$ and to extract accurate input/output values from raw generation.

4.3 Code Repair via Fill-in-the-Middle

To assess HLP's impact on practical applications beyond code completion, we evaluate the performance of code repair using Defects4J (Just et al., 2014). Defects4J consists of open-source bugs found across 15 Java repositories. Following existing works (Xia et al., 2023; Xia and Zhang, 2023), we collect 313 single-hunk bugs from Defects4J that can be fixed by replacing or adding a continuous code hunk. Specifically, for each bug, models are prompted to generate the correct code hunk (*i.e.,* patch) given the left and right contexts of the buggy code hunk, with correctness verified by project test suites. As shown in the "Code Repair" section of Table 5, adding HLP during training results in relatively up to 18% more bugs fixed by the model[2], suggesting that enhanced planning capabilities translate directly to better bug-fixing performance.

4.4 Code Reasoning via Fill-in-the-Middle

We further examine whether HLP's planning benefits extend to code reasoning tasks. We use CRUXEval (Gu et al., 2024) which comprises 800 Python functions paired with two distinct tasks: CRUXEval-I, where LLMs need to predict the input from the known output, and CRUXEval-O, where LLMs are required to predict the output based on the given input.

We reformat prompts of CRUXEval-I into FIM style and leave CRUXEval-O as L2R generation, both of which are evaluated in zero-shot setting. Different from previous subsections where post-processing is not used, we follow the same pipeline as in the original CRUXEval paper to extract accurate input/output values from generation because we are focusing on evaluating the reasoning capability of LLMs rather than their capability of generating correct code[3]. As shown in the "Code Reasoning" section of Table 5, HLP demonstrates up to 6% improvements on both CRUXEval-I and CRUXEval-O tasks for all the code LLMs consistently, which shows that HLP also improves intrinsic code reasoning capabilities of LLMs.

5 Discussion

5.1 NTP Alone Cannot Yield Horizon Awareness

A key question of interest is whether standard next-token prediction (NTP) inherently provides models with awareness of generation horizons. Through systematic analysis, we demonstrate that hidden states of models trained with NTP alone do not capture meaningful information about required completion lengths.

To quantify this, we design a probing task that attempts to predict the remaining completion length from models' hidden states. We extract hidden states from 20K code snippets (approximately 7.8M tokens) using models trained with and without HLP and split the data to ensure no sequence-level overlap between train and test. Using these hidden states as input and normalized remaining token counts as targets, we fit linear regression models while keeping the underlying transformer parameters frozen.

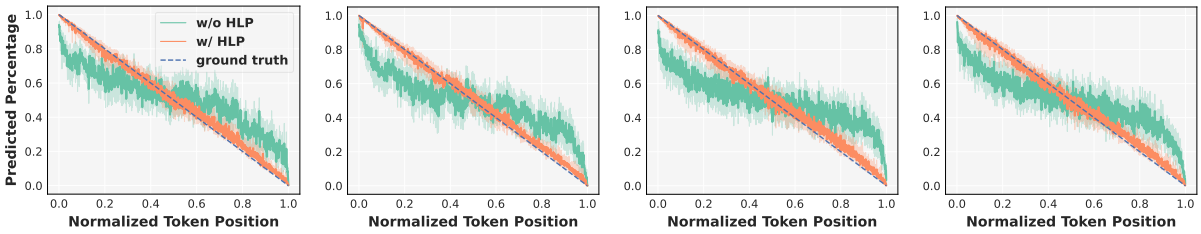
Test ↑	DS-1.3B	DS-6.7B	SC2-3B	SC2-7B
NTP	0.440	0.519	0.356	0.410
NTP+HLP	**0.915**	**0.913**	**0.932**	**0.932**

Table 6: Probing results of models trained w/o and w/ HLP. We report the coefficient of determination (R^2) of prediction, which is the higher the better.

Figure 3 shows predicted versus actual remaining token

[2]Note that Defects4J is a small dataset with only 313 examples and models can only solve 30-60 out of those, which makes it hard to obtain statistically significant differences with greedy decoding.

[3]In CRUXEval-I, we only want to evaluate the correctness of the input value infilled by LLMs in the given assertion. However, FIM-style prompts we use in the experiments does not restrict LLMs from writing multiple assertions before starting infilling the given assertion, which is useless in this task. So we use post-processing techniques to extract the input value infilled for the given assertion to better evaluate the reasoning capabilities.



(a) DS-Coder 1.3B (b) DS-Coder 6.7B (c) StarCoder2 3B (d) StarCoder2 7B

Figure 3: Predicted percentage of remaining future tokens (as defined in Eq. (3)) from models trained w/o and w/ HLP at different token positions, where the position of each token is normalized to the corresponding percentage over the sequence.

percentages at different positions, while Table 6 reports the coefficient of determination (R^2). Models trained with NTP alone show poor fit, indicating their hidden states lack horizon information. In contrast, models trained with HLP show much stronger correlations between hidden states and remaining lengths. This result demonstrates that horizon awareness must be explicitly trained rather than emerging naturally from NTP. We also carried out a non-linear probing experiment with the same inputs and targets by replacing linear regression models with an MLP regressor that uses logistic activation function (Appendix A.2), which further confirms our finding here.

5.2 Why Horizon-Length Prediction works?

The effectiveness of Horizon-Length Prediction (HLP) stems from its ability to address a critical limitation of standard next-token prediction (NTP) in autoregressive models: the absence of global planning awareness. While NTP optimizes for local token-level coherence by maximizing the likelihood of immediate next tokens, it inherently struggles to enforce long-horizon structural consistency between the generated middle and the given suffix in FIM. This limitation becomes pronounced when the model must align generated code with both preceding logic and subsequent constraints (e.g., API calls or variable dependencies defined in the suffix). HLP bridges this gap by teaching models to explicitly condition the model on the remaining generation horizon at every decoding step. By regressing the normalized count of future tokens required to complete middle, HLP effectively decomposes the infilling task into a sequence of length-aware subgoals, where each generated token is tied to a dynamically updated "budget" of remaining steps. This mechanism mirrors human problem-solving strategies, where progress estimation (e.g., "30% of steps remaining") guides iterative refinement of intermediate actions to meet global objectives. The success of HLP is rooted in its complementary role to NTP. While NTP ensures local fluency, HLP provides a global scaffold for planning. By unifying local token prediction with global horizon awareness, HLP bridges the gap between autoregressive decoding and holistic reasoning, enabling models to dynamically adapt their generation strategy to long-horizon constraints. This synergy is particularly critical in code infilling, where the correctness of middle depends on both preceding logic (*i.e.,* prefix) and subsequent context (*i.e.,* suffix), demanding a balance between immediate token likelihood followed by prefix and forward-looking structural coherence constrained by distant suffix.

Our in-depth attention analysis provides direct evidence of this enhanced planning capability. Figure 4 shows that models trained with HLP pay significantly more attention to suffix context, particularly at the start of the generation. This indicates that HLP enables proactive planning by considering long-horizon constraints from the beginning. Furthermore, the consistent improvements across FIM, bug-fixing, and reasoning tasks demonstrate that this enhanced planning capability generalizes beyond simple completion scenarios.

In summary, HLP transforms code generation from a myopic token-by-token process into a structured planning task. It enables models to dynamically adapt their generation strategy to long-horizon constraints, ensuring that each step not only satisfies local fluency

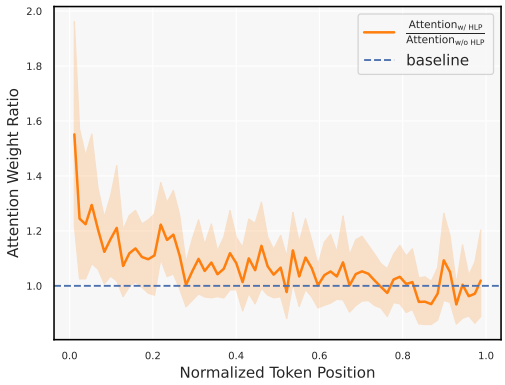


Figure 4: Attention analysis of DeepSeek-Coder-Base 1.3B on SAFIM, showing the ratio of attention paid to suffix between models trained with and without HLP. The X-axis shows the normalized position in the sequence, and the Y-axis shows the attention ratio. Values above 1 indicate that the HLP model pays more attention to suffix than the baseline model. We observe that the model trained with HLP generally pays more attention to suffix, especially at the beginning, demonstrating its lookahead planning behavior.

but also contributes to a globally coherent solution. This dual focus on immediate actions and overarching goals positions HLP as a pivotal advancement in code LLMs.

5.3 HLP Mitigates Planning Failures in FIM

We conduct a quantitative analysis to empirically demonstrate the frequency of planning failures in FIM and evaluate the effectiveness of HLP in addressing these failures. As presented in Figure 1 (§2), a prevalent manifestation of planning failures in FIM is the "correct code but incorrect order" pattern, wherein the model generates the final line of ground truth code accurately but positions it incorrectly within the completion. To maintain analytical tractability, our study focuses specifically on planning failures that result in this particular pattern. We establish the following criteria to identify instances of the "correct code but incorrect order" pattern in model's FIM completions:

- Completion fails to pass all the unit tests.
- Completion contains the last line from ground truth.
- Completion does not end with this last line.

Testing on SAFIM's Algorithmic and Algorithmic_v2 partitions which focus on multi-line FIM, we find that models trained without HLP exhibit the "correct code but incorrect order" pattern in 5-10% of problems across the benchmark, as shown in Table 7. With HLP training, this drops to 4-6%, demonstrating that HLP helps alleviate lack-of-planning failures effectively.

	SAFIM		Avg
	Algo	Algo_{v2}	
DS-1.3B	9.34%	10.48%	9.91%
+ HLP	**6.83%**	**6.52%**	**6.68%**
DS-6.7B	5.16%	5.93%	5.55%
+ HLP	**4.36%**	**4.44%**	**4.40%**
SC2-3B	7.25%	7.35%	7.30%
+ HLP	**5.65%**	**5.43%**	**5.54%**
SC2-7B	5.79%	6.26%	6.03%
+ HLP	**4.69%**	**4.29%**	**4.49%**

Table 7: Quantitative analysis on the frequency of lack-of-planning issue in models trained w/ and w/o HLP on SAFIM. We follow the same settings used in §4.1.

6 Related Work

Fill-in-the-Middle for Code Language Models Large language models trained on massive source code data have demonstrated great potential in various applications for software development. While early models such as Codex (Chen et al., 2021) and CodeGen (Nijkamp et al., 2023) only support Left-to-Right (L2R) generation, Fill-in-the-Middle (or infilling) has attracted increased attention because right context naturally carries an indispensable part of information for completing code in the middle (Fried et al., 2023; Bavarian et al., 2022). Subsequently, FIM training has become a common practice widely adopted by most code LLMs, such as StarCoder (Li et al., 2023; Lozhkov et al., 2024), DeepSeek-Coder (Guo et al., 2024; DeepSeek-AI et al., 2024), and Code Llama (Rozière et al., 2023).

Existing models generally tackle the infilling problem by breaking a code snippet into prefix-middle-suffix, and reordering them into prefix-suffix-middle (PSM) or suffix-prefix-middle (SPM), which are then used for next-token prediction (NTP) training. We point out that the infilling task cannot be effectively learned with NTP alone, as it requires planning capability for the model to fluently and meaningfully connect the middle completion to suffix through forward looking during auto-regressive decoding.

An alternative approach is to train two language models in different directions, with one generating from left to right and the other from right to left, and have the two generations meet in the middle (Nguyen et al., 2023). Nevertheless, the L2R model does not have access to the right context, and vice versa, which impedes holistic planning that takes into account the context from both sides.

Planning and Lookahead in Language Generation Standard decoder-only models are trained with next-token prediction and used to sequentially predict one token at a time, conditioned only on past tokens, in an auto-regressive manner. One drawback of this paradigm is that models are not aware of future tokens during decoding. The token that maximizes the conditional probability at current step may lead to suboptimal continuation, and consequently the model can fail to compose a fluent and sensible generation that meets human requirements. Various decoding techniques have been proposed to address the problem through tree search with lookahead heuristics, particularly for constrained generation problems (Lu et al., 2022; Huang et al., 2024). While these methods are training-free, they inevitably incur additional cost of inference complexity.

Apart from those, Gloeckle et al. (2024) proposed to predict multiple tokens from a single hidden state during both training and inference, which was shown to achieve stronger performance on coding tasks with no computation overhead. While multi-token prediction enhances models' planning capability within the n tokens predicted together ($n \leq 8$), we argue that with a small n,

the limited horizon is usually insufficient for planning in the case of infilling as the connection from middle to suffix only happens towards the end of the generation. In contrast, HLP adopts a global and arbitrary long horizon over all future tokens by counting the remaining generation budget, which more effectively helps models to close the generation fluently with early planning[4].

7 Conclusion and Future Work

Fill-in-the-Middle is ubiquitous in code completion, reflecting the iterative nature of software development where code is frequently inserted between existing sections, and thus has become an important consideration in the development of code language models. The current FIM training paradigm splits and reorders training sequences for next-token prediction. However, this approach frequently results in models struggling to generate content that smoothly aligns with the right context. While existing FIM benchmarks rely on different post-processing methods to circumvent this problem, we emphasize that such methods typically require dataset-specific assumptions, which are impractical in real-world scenarios.

To address this limitation and enhance the infilling capability of code language models, we propose Horizon-Length Prediction (HLP). HLP teaches models to predict the portion of remaining tokens at every step, enabling them to develop planning awareness over arbitrarily long horizons. Experiments across different model families and sizes show that HLP improves infilling performance on diverse FIM benchmarks, across file-level and repository-level, and without using any dataset-specific post-processing. Moreover, the enhanced planning capability acquired through HLP training also boosts models' performance on code reasoning tasks, suggesting broader benefits for language models' reasoning capabilities. Importantly, HLP achieves these improvements while maintaining efficiency, with negligible training overhead and no additional inference cost. Our work marks a significant advancement in developing more effective code language models for real-world applications. Future work could explore extending HLP to other generation tasks requiring long-horizon planning and investigate its potential for enhancing general reasoning capabilities.

Limitations

While HLP has proven effective through extensive evaluation in the paper, our pre-training experiments are restricted to models of $\leq$7B parameters due to the computation budget. It is prohibitively expensive to perform pretraining experiments for LLM, and unfortunately we do not have enough resources to demonstrate the impact of HLP on larger models. This highlights a broader challenge in open science for pre-training research. In addition, this work has mainly focused on the code domain, but the idea of horizon-length prediction can be broadly applicable for improving models' reasoning capability in natural language and mathematical domains as well, which we leave to future work.

[4]Please refer to Appendix §A.3 for more details.

References

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *ArXiv preprint*, abs/2108.07732.

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.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code.

DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, and 21 others. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence.

Hantian Ding, Zijian Wang, Giovanni Paolini, Varun Kumar, Anoop Deoras, Dan Roth, and Stefano Soatto. 2024. Fewer truncations improve language modeling.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

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, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Fabian Gloeckle, Badr Youbi Idrissi, Baptiste Rozière, David Lopez-Paz, and Gabriele Synnaeve. 2024. Better & faster large language models via multi-token prediction. *ArXiv preprint*, abs/2404.19737.

Linyuan Gong, Sida Wang, Mostafa Elhoushi, and Alvin Cheung. 2024. Evaluation of llms on syntax-aware code fill-in-the-middle tasks.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence.

James Y. Huang, Sailik Sengupta, Daniele Bonadiman, Yi an Lai, Arshit Gupta, Nikolaos Pappas, Saab Mansour, Katrin Kirchoff, and Dan Roth. 2024. Deal: Decoding-time alignment for large language models. *ArXiv preprint*, abs/2402.06147.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report.

René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 437–440, New York, NY, USA. Association for Computing Machinery.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, and 48 others. 2023. Starcoder: may the source be with you!

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, and 47 others. 2024. Starcoder 2 and the stack v2: The next generation.

Ximing Lu, Sean Welleck, Peter West, Liwei Jiang, Jungo Kasai, Daniel Khashabi, Ronan Le Bras, Lianhui Qin, Youngjae Yu, Rowan Zellers, Noah A. Smith, and Yejin Choi. 2022. NeuroLogic a*esque decoding: Constrained text generation with lookahead heuristics. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 780–799, Seattle, United States. Association for Computational Linguistics.

Anh Nguyen, Nikos Karampatziakis, and Weizhu Chen. 2023. Meet in the middle: A new pre-training paradigm. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2023. Code llama: Open foundation models for code.

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024. Repoformer: Selective retrieval for repository-level code completion.

Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2023. The plastic surgery hypothesis in the era of large language models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 522–534.

Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, Singapore. Association for Computational Linguistics.

Lin Zheng, Jianbo Yuan, Zhi Zhang, Hongxia Yang, and Lingpeng Kong. 2024. Self-infilling code generation. *Preprint*, arXiv:2311.17972.

A Appendix

A.1 Effect of λ Value Selection

Following the original experimental settings (but with 100k training steps given limited computational resources), we studied a series of λ values and report both HLP validation loss and SAFIM pass@1. As shown in Table 8, our method is robust to hyperparameter selection: all non-zero λ values improve over

the baseline ($\lambda = 0$). Moreover, we observe that HLP loss quickly decreases within the first few thousand training steps with all positive λ's, indicating that a large λ isn't necessary for models to learn HLP well. This also coincides with our observation on downstream performance: while lower HLP loss generally correlates with better SAFIM performance, the improvement plateaus beyond a certain threshold, suggesting that once the model acquires basic horizon awareness, further optimization of HLP loss provides diminishing returns for downstream performance.

λ	HLP Valid Loss	SAFIM (pass@1)
0	0.254	47.40%
0.01	0.060	48.98%
0.1	0.048	49.10%
0.2	0.045	48.74%
0.5	**0.043**	**49.34%**

Table 8: Study over the effect of λ value selection on HLP valudation loss and SAFIM performance.

A.2 Non-Linear Probing for Horizon Awareness

By replacing the linear regression models in the original linear probing experiment with an MLP regressor that uses logistic activation function, we further conduct a non-linear probing experiment to study whether NTP is able to yield horizon awareness. As shown in Table 9, even under the assumption of the non-linear relationship, models trained without HLP cannot effectively extract horizon information from their hidden states. This demonstrates that horizon awareness must be explicitly trained rather than emerging naturally from NTP.

Test ↑	DS-1.3B	DS-6.7B	SC2-3B	SC2-7B
NTP	0.392	0.431	0.180	0.226
NTP+HLP	**0.897**	**0.885**	**0.856**	**0.871**

Table 9: Non-linear probing results of models trained w/o and w/ HLP. We report the coefficient of determination (R^2) of prediction, which is the higher the better.

A.3 Comparing HLP with Multi-token Prediction

As discussed in §6, we argue that multi-token prediction (Gloeckle et al., 2024) is insufficient for planning in FIM, because multi-token prediction only enhances models' planning capability for a short and limited horizon, which does not suites FIM well as the connection from middle to suffix happens over a long horizon. Instead, HLP focuses on long-horizon planning and is more effective for FIM. We conduct an experiment following the same settings in §4 to compare HLP with multi-token prediction on DeepSeek-Coder-Base 1.3B by predicting the next 4 tokens (Gloeckle et al., 2024). We report their performance on SAFIM. As shown in Table 10, while adding HLP to NTP largely improves the model's performance on SAFIM, multi-token prediction fails to do so. These results provide empirical evidence that long-horizon planning capabilities brought by HLP is essential for advancing FIM performance.

A.4 HLP Improves Self-Infilling Performance

Recently, some strategies have been proposed to conduct L2R code generation based on FIM capabilities and self-infilling (Zheng et al., 2024) is one of the representative strategies of this kind. Given an initial input prompt prefix, self-infilling performs L2R code generation by using FIM decoding to (1) generate suffix based on prefix and then (2) generate middle based on prefix and previously generated suffix. Furthermore, self-infilling proposes a looping mechanism to improve the generated code iteratively, where it first uses L2R decoding to generate a new suffix′ based on prefix and previously generated middle and then uses FIM decoding to generate a new middle′ based on prefix and this newly generated suffix′. This looping procedure can be continued for multiple rounds to obtain greater improvements. Since self-infilling relies on FIM capabilities for L2R code generation, it is interesting to study whether HLP can also enhance its performance. To this end, we evaluate the self-infilling performance of DeepSeek-Coder-Base 1.3B trained with and without HLP on HumanEval using greedy decoding, following the setting of the original paper (Zheng et al., 2024). We report models' performance on HumanEval with N ranging from 0 to 4, where N denotes the number of times the decoding process goes through the loop and $N = 0$ represents that the looping mechanism is not activated. As shown in Table 11, the model trained with HLP consistently outperforms the model trained without HLP across different number of self-improving steps. Furthermore, with more self-improving steps, while the performance of the model trained without HLP gets stuck, the performance of the model trained with HLP continues to showcase steady improvements.

A.5 Effect of HLP on Left-to-Right Performance

While HLP have significantly improved the FIM performance of LLMs, we also study its impact on the L2R code completion. To this end, we evaluate L2R performance on HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) with DeepSeek-Coder-Base 1.3B. We further employ HumanEval+ and MBPP+ from EvalPlus (Liu et al., 2023) for more rigorous evaluation with better test coverage. As shown in Table 12, with HLP applied to FIM data only (*i.e.,* HLP_{FIM}), the perfor-

	SAFIM				Average
	Algo	Algo_{v2}	Control	API	
DS-1.3B	39.8	42.4	52.4	56.1	47.7
+ HLP	**41.3**	**46.1**	**53.4**	**59.0**	**50.0**
+ multi-token prediction	38.1	41.3	51.7	55.8	46.7

Table 10: Pass@1 results of training w/ HLP and w/ multi-token prediction for DeepSeek-Coder-Base 1.3B on SAFIM (Gong et al., 2024) computed with greedy decoding.

HumanEval	$N=0$	1	2	3	4
DS-1.3B	27.4	29.3	31.7	31.7	31.7
+ HLP	**30.5**	**31.7**	**31.7**	**32.3**	**32.9**

Table 11: Self-infilling HumanEval performance of models trained w/o and w/ HLP, with N ranging from 0 to 4. N denotes the number of times the decoding process goes through the loop and $N = 0$ represents that the looping mechanism is not activated. We report pass@1 results using greedy decoding.

mance on L2R tasks sometimes shows a slight degradation. We hypothesize that applying HLP to middle only causes unbalanced training on prefix and suffix parts. To mitigate such effect, we need to devise another HLP task that can be applied to L2R training (*i.e.,* HLP_{L2R}). However, the original design of HLP in §3 is not directly applicable to L2R data. While the end of middle in FIM data is strictly bounded by the beginning of suffix, the end of L2R data does not have any clear signals, as it is often possible to add additional contents (*e.g.,* another line of code or a new helper function) to the end of document fluently without any restrictions. Therefore, instead of taking the entire code file as the prediction horizon, we ask the model to predict the number of future tokens **required to complete current line** in L2R training, which is a natural semantic unit in code. Furthermore, to avoid conflicts between HLP_{FIM} and HLP_{L2R}, we use two independent hlp_heads to let the model learn HLP_{FIM} and HLP_{L2R} separately. As shown in Table 12, by applying HLP_{FIM} and HLP_{L2R} simultaneously, the performance degradation on L2R tasks is recovered, with the improvement on FIM tasks largely retained. These results demonstrate the generalizable effectiveness of HLP and shows the huge potential of applying the idea of HLP to more general training scenarios.

A.6 Additional Ablation Studies

We conduct several ablation studies to justify the design choices of HLP. In this section, we conduct experiments using DeepSeek-Coder-Base 1.3B, follow the same settings in §4, and report models' performance on SAFIM.

Complexity of hlp_head**.** We conduct an experiment to study the effect of the complexity of hlp_head by replacing the original linear layer (*i.e.,* "HLP (linear)") with a two-layer MLP with ReLU (*i.e.,* "HLP (mlp)"). As shown in Table 13, increasing the complexity of hlp_head does not bring significant improvements. We have also conducted a paired t-test between "HLP (linear)" and "HLP (mlp)" and did not see any clear directional statistical significance between them. Such results indicate that the complexity of hlp_head does not have a major impact on performance.

Applying HLP to all tokens *v.s.* first token only. While it is easy to see that knowing the HLP loss on the first token is sufficient to infer the horizon length in theory, having HLP loss on every token provides denser and more consistent supervision signals which makes learning easier (as discussed in §3). It also helps regularize the hidden representation of every subsequent token. To empirically show this, we conduct an experiment by applying HLP to the first token only (i.e., "HLP (first)") and compared its performance with our original HLP design (i.e., "HLP (all)"). As shown in Table 14, while applying HLP only to the first token performs better than NTP only, applying HLP loss for each token can achieve better performance than applying it to just the first token.

Normalized *v.s.* unnormalized targets. We use normalized targets in our original HLP design (*i.e.,* using $\frac{M-t}{M}$ rather than $M-t$) is that the scale of HLP loss will be otherwise in a huge range, e.g., some examples has single digit loss while some might have thousands. To further study the effect of normalization, we conduct an experiment by using $M-t$ as the target (*i.e.,* "HLP (raw)") rather than $\frac{M-t}{M}$ (*i.e.,* "HLP (normalized)"). To achieve this, we remove the sigmoid function from the original HLP. As shown in Table 15, setting the target as $M-t$ fails to improve FIM performance, likely due to the large-scale HLP loss after using $M-t$ as the target interferes with NTP pre-training.

	Left-to-Right		Fill-in-the-Middle
	HumanEval (+)	MBPP (+)	SAFIM
DS-1.3B	**26.3 (22.0)**	**45.8 (36.7)**	47.7
+ HLP $_{\text{FIM}}$	25.5 (21.3)	45.8 (36.5)	**50.0**
+ HLP $_{\text{FIM}}$ + HLP $_{\text{L2R}}$	**26.2 (22.0)**	**45.7 (36.6)**	**49.6**

Table 12: Effect of HLP_{FIM} only and $\text{HLP}_{\text{FIM}}+\text{HLP}_{\text{L2R}}$ for DeepSeek-Coder-Base 1.3B on L2R and FIM tasks. On L2R tasks including HumanEval (+) and MBPP (+), we do sampling with $T=0.8$ and $n=200$. We report pass@1 performance of all the models, where numbers outside and inside parenthesis "()" indicate `base` and `plus` versions of EvalPlus, respectively. For FIM experiments on SAFIM, we follow the same settings used in §4.1.

	SAFIM				Average
	Algo	Algo_{v2}	Control	API	
DS-1.3B	39.8	42.4	52.4	56.1	47.7
+ HLP (linear)	41.3	**46.1**	53.4	**59.0**	**50.0**
+ HLP (mlp)	**41.6**	45.9	**54.0**	57.4	49.7

Table 13: Pass@1 results of HLP w/ linear layer as `hlp_head` (*i.e.,* "HLP (linear)") and w/ MLP layer as `hlp_head` (*i.e.,* "HLP (mlp)") for DeepSeek-Coder-Base 1.3B on SAFIM (Gong et al., 2024) computed with greedy decoding.

	SAFIM				Average
	Algo	Algo_{v2}	Control	API	
DS-1.3B	39.8	42.4	52.4	56.1	47.7
+ HLP (all)	**41.3**	**46.1**	**53.4**	**59.0**	**50.0**
+ HLP (first)	40.0	44.4	52.3	57.4	48.5

Table 14: Pass@1 results of applying HLP to all tokens (*i.e.,* "HLP (all)") and first token only (*i.e.,* "HLP (first)") for DeepSeek-Coder-Base 1.3B on SAFIM (Gong et al., 2024) computed with greedy decoding.

	SAFIM				Average
	Algo	Algo_{v2}	Control	API	
DS-1.3B	39.8	42.4	52.4	56.1	47.7
+ HLP (normalized)	**41.3**	**46.1**	**53.4**	**59.0**	**50.0**
+ HLP (raw)	27.7	29.8	34.4	47.7	34.9

Table 15: Pass@1 results of using normalized targets (*i.e.,* "HLP (normalized)") and unnormalized targets (*i.e.,* "HLP (raw)") in HLP for DeepSeek-Coder-Base 1.3B on SAFIM (Gong et al., 2024) computed with greedy decoding.