Self-Infilling Code Generation

Lin Zheng [1 *] **Jianbo Yuan** [2] **Zhi Zhang** [2] **Hongxia Yang** [2] **Lingpeng Kong** [1]

Abstract

In this work, we introduce self-infilling code generation, a general framework that incorporates infilling operations into auto-regressive decoding. Our approach capitalizes on the observation that recent infilling-capable code language models can perform *self-infilling*: whereas conventional infilling is designed to fill in the middle based on a predefined prefix and suffix, self-infilling sequentially generates both such surrounding context and the infilled content. We utilize self-infilling to introduce novel interruption and looping mechanisms in conventional decoding, evolving it into a non-monotonic process. Interruptions allow for postponing the generation of specific code until a definitive suffix is established, enhancing control during decoding. Meanwhile, the looping mechanism, which leverages the complementary nature of self-infilling and left-to-right decoding, can iteratively update and synchronize each piece of generation cyclically. Extensive experiments across a variety of code generation benchmarks demonstrate that decoding with self-infilling not only improves the output quality but also regularizes the overall generation, which effectively mitigates potential degeneration and scaffolds code to be more consistent with intended functionality.

1. Introduction

Contemporary large language models have achieved excellent performance in tasks related to code generation and understanding (Austin et al., 2021; Chen et al., 2021; Fried et al., 2023; Nijkamp et al., 2023b; Li et al., 2022; Anil et al., 2023; OpenAI, 2023; Touvron et al., 2023a; Li et al., 2023; Touvron et al., 2023b; Muennighoff et al., 2023; Nijkamp et al., 2023a; Roziere et al., 2023; Xie et al., 2023; Xu et al., 2024). Building upon this success, an active line of research aims to endow these language models with enhanced free-form generation capacities (Fried et al., 2023; Bavarian et al., 2022; Allal et al., 2023; Nijkamp et al., 2023a; Li et al., 2023; Roziere et al., 2023), where models learn to *infill* content considering both preceding and subsequent contexts. Such capabilities are instrumental for numerous downstream code-related tasks that require a bidirectional context, including but not limited to partial code completion, docstring generation, and type prediction (Fried et al., 2023; Li et al., 2023; Roziere et al., 2023).

Established practices for fostering infilling capabilities in language models involve training with an explicit infilling objective (Raffel et al., 2020; Bavarian et al., 2022; Tay et al., 2023; Anil et al., 2023; Fried et al., 2023; Nijkamp et al., 2023a), which directs the model to predict the missing span of sequence tokens given the surrounding context. Despite a substantial allocation of computational resources dedicated to the specialized infilling training (Bavarian et al., 2022; Li et al., 2023; Roziere et al., 2023), most code generation systems still persist in a strictly left-to-right fashion. It remains unclear how the acquired proficiency in infilling could aid in synthesizing *complete* code beyond narrowly tailored partial infilling tasks (Bavarian et al., 2022; Fried et al., 2023; Allal et al., 2023).

In this work, we investigate the integration of infilling capabilities into the generation process of code language models. We start with a seemingly straightforward yet overlooked observation: recent open-source code language models, specifically those trained with fill-in-the-middle objectives (Bavarian et al., 2022) (e.g., StarCoder (Li et al., 2023) and Code Llama (Roziere et al., 2023)), inherently possess *self-infilling* capabilities (§2.1). Unlike regular infilling operations that demand a partial surrounding context as input, self-infilling autonomously generates *both* the surrounding context and the infilled content.

We leverage this finding to develop an *interruption* mechanism (§2.2) in conventional left-to-right decoding, transforming it into a non-monotonic process. This allows the language model to temporarily halt the decoding process when necessary, craft a suffix, and then return to the interruption point to infill the bypassed context (Figure 1c). To handle the variable length of the skipped context and

[*]This work was done during an internship at ByteDance. [1]The University of Hong Kong [2]ByteDance Inc. Correspondence to: Lin Zheng <lzheng2@cs.hku.hk>.

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

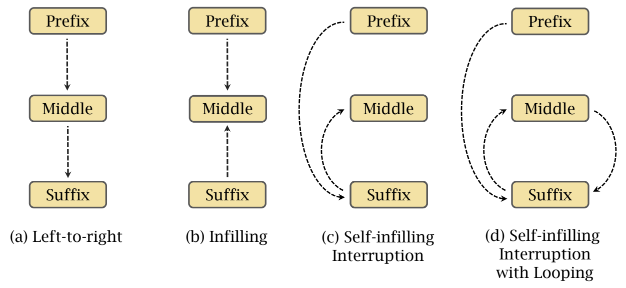

Prefix
Middle
Suffix
(a) Left-to-right
(b) Infilling
(c) Self-infilling Interruption
(d) Self-infilling Interruption with Looping

Figure 1. Schematic illustrations of various decoding approaches for code generation. **(a)** and **(b)** represent standard left-to-right decoding and infilling operations, respectively. Whereas infilling requires the *user-provided* prefix and suffix, self-infilling interruption **(c)** autonomously generates these segments. **(d)** further expands on self-infilling by incorporating a looping mechanism.

enforce an appropriate suffix, we introduce *suffix prompting* by initiating the suffix with a prespecified yet common prompt. The interruption mechanism is particularly useful in high-entropy situations, where conventional decoding methods falter due to uncertainty in predicting the next token and might cause propagation of errors through the context, known as the exposure bias (Ranzato et al., 2015; Bengio et al., 2015). Instead, self-infilling interruption offers an *easy-first* alternative by deferring the generation of more difficult tokens and crafting a well-definitive suffix. This suffix works as an anchor to scaffold the code and produce a more structured context for the subsequent infilling, which ensures the final generation is logically consistent with the intended functionality and enjoys improved regularity.

Besides, self-infilling implies a *looping* mechanism (§2.3) to further enhance the decoding process. Leveraging the complementary nature of left-to-right and self-infilling methods, it iteratively updates code snippets with broader contexts, wherein the output from one mode seamlessly informs the input to the other (Figure 1d). Specifically, the suffix in self-infilling is generated based solely on the prefix with a rather limited context. This can be addressed by chaining a subsequent left-to-right decoding phase, enabling the suffix to be re-generated within an expanded context that includes the infilled middle. Similarly, we can continue by appending the next self-infilling to synchronize the middle and the prefix based on the latest suffix information. Such piecewise synchronization allows each snippet to be recurrently updated with more informative contexts.

Intuitively, the developed interruption and looping mechanism mirrors human coding practices, which typically do not conform strictly to a linear, left-to-right progression. Instead, code is usually written via a dynamic process of continuous refinement, editing, and reconnection of fragments. While the replication of human coding strategies may not necessarily be the optimal approach for language models, our extensive experiments (§3) on various code benchmarks verify the effectiveness of self-infilling generation. We demonstrate that our framework brings significant improvements in the quality and regularity of code generation compared to conventional left-to-right approaches.

2. Self-infilling Code Generation

In this section, we introduce self-infilling, the built-in capability of models trained with *fill-in-the-middle* (§2.1; Bavarian et al., 2022). We then detail our integration of self-infilling into the decoding process, including the developed interruption (§2.2) and looping (§2.3) mechanisms.

2.1. FIM Training Entails Self-infilling

Fill-in-the-middle (FIM) Training. Throughout this work, we focus on code language models with causal decoder-only Transformers (Vaswani et al., 2017; Radford et al., 2018), which are currently the dominant paradigm at large scale. Most of these language models are trained with conventional left-to-right next token prediction (Radford et al., 2018; 2019). Despite compelling performance (Austin et al., 2021; Chen et al., 2021; Nijkamp et al., 2023b), such training objective only permits left-to-right generation for downstream applications and restricts more versatile generation tasks like infilling, which necessitates a non-causal (or bidirectional) context. A recent practice to address this limitation is training with the *Fill-In-the-Middle* (FIM) objective (Bavarian et al., 2022), which is extensively employed in recent code models (Allal et al., 2023; Li et al., 2023; Roziere et al., 2023). FIM randomly splits input raw code into three pieces (`prefix`, `middle`, `suffix`) and rearranges them to form a permuted sequence `[<PRE>;prefix;<SUF>;suffix;<MID>;middle;<EOT>]`, where `;` denotes concatenation and particular sentinel tokens are interleaved to mark the boundary of each piece.[1] The model is then optimized to maximize the factorized likelihood $p(\texttt{prefix})p(\texttt{suffix} \mid \texttt{prefix})p(\texttt{middle} \mid \texttt{prefix}, \texttt{suffix})$ with each modeled in a standard left-to-right auto-regressive manner.

FIM Entails Self-infilling. Our key observation is that FIM (Bavarian et al., 2022) trains the model to predict the "next" token irrespective of the specific pieces (i.e., `prefix`, `middle`, or `suffix`) these tokens belong to. Consequently, the model learns not only to predict the `middle` conditioned on both `prefix` and `suffix`, but also to predict `suffix` only based on `prefix`. While the

[1]An alternative format, known as the SPM mode (Bavarian et al., 2022), structures the pieces as `[<PRE>;<SUF>;suffix;<MID>;prefix;middle;<EOT>]`. However, this permutation is not as useful in our setting, because we would like the suffix to be conditioned on the prefix context.

former is extensively studied in the literature as infilling (Donahue et al., 2020; Bavarian et al., 2022; Du et al., 2022; Aghajanyan et al., 2022; Tay et al., 2022; 2023; Fried et al., 2023), the latter is rarely explored, which characterizes the distribution of `suffix` given `prefix` marginalized over possible outcomes of `middle`. This leads to *self-infilling*, where the model can first generate a `suffix` based on `prefix` and then return to fill in the `middle` given the *self-generated* context. We leverage this aspect to develop a non-monotonic decoding process, as detailed subsequently.

2.2. Self-infilling Interruption

Self-infilling introduces a dynamic decoding process with the *interruption* mechanism. Specifically, given an initial input prompt denoted as `prefix`, the language model performs next-token prediction similar to regular decoding,

$$\texttt{prefix} \sim p(\texttt{prefix}). \tag{1}$$

Our approach enhances decoding by incorporating *interruptions*. On invocation, the current decoding is suspended to generate a `suffix`, followed by filling in the `middle`:

$$\texttt{suffix} \sim p(\texttt{suffix} \mid \texttt{prefix}), \tag{2}$$

$$\texttt{middle} \sim p(\texttt{middle} \mid \texttt{prefix}, \texttt{suffix}). \tag{3}$$

In general, interruptions can be triggered by various indicators, such as low likelihood or the occurrence of specific tokens. We implement a simple heuristic to signal interruptions by employing a probability threshold τ. If the maximum probability over the next token falls below τ, indicating high uncertainty, self-infilling is invoked. Contrasting with left-to-right decoding, self-infilling interruption adopts an *easy-first* methodology and dynamically defers the generation of potentially difficult snippets. It is thus useful to mitigate divergent issues due to producing an error-prone context, known as the exposure bias (Bengio et al., 2015; Ranzato et al., 2015).

Suffix Prompting. Accurately crafting a proper `suffix` from only `prefix` is often challenging due to the indeterminate nature of the skipped segment `middle` during infilling pretraining. To guide suffix generation, we propose *suffix prompting* that enforces `suffix` to start with specific tokens. For instance, in Python function generation (Chen et al., 2021; Austin et al., 2021), a common ending is a return statement containing the keyword **`return`**. Such common keywords can be used as the *suffix prompt* (denoted as $\texttt{suffix}^{\text{p}}$) to shape `suffix`.

Technically, we represent the whole input sequence as a quadruple in the following form,

$$[\texttt{prefix};\ \texttt{middle};\ \texttt{suffix}^{\text{p}};\ \texttt{suffix}^{\text{c}}], \tag{4}$$

where $\texttt{suffix} \coloneqq [\texttt{suffix}^{\text{p}}; \texttt{suffix}^{\text{c}}]$ and $\texttt{suffix}^{\text{c}}$ is the *suffix completion* of $\texttt{suffix}^{\text{p}}$. Given `prefix` and a predefined $\texttt{suffix}^{\text{p}}$, we can further expand upon Equation 2 as follows,

$$\begin{aligned} \texttt{suffix}^{\text{c}} &\sim p(\texttt{suffix}^{\text{c}} \mid \texttt{prefix}, \texttt{suffix}^{\text{p}}), \\ \texttt{suffix} &\coloneqq [\texttt{suffix}^{\text{p}}; \texttt{suffix}^{\text{c}}]. \end{aligned} \tag{5}$$

The use of suffix prompting helps generate an appropriate `suffix`, which serves as a contextual anchor to scaffold the overall generation and ensure the output structure is logically consistent with the intended functionality.

In practice, self-infilling interruption is implemented through the manipulation of sentinel tokens. For instance, the language model is programmed to produce `<SUF>` as the next token upon interruption activation. Our detailed implementation is outlined in Algorithm 2 (Appendix A).

2.3. Decoding through a Looping Mechanism

In this section, we introduce a *looping mechanism* to improve decoding, which interweaves self-infilling with left-to-right conditional generation to recurrently update snippets. Note that during self-infilling, `suffix` is generated based solely on `prefix`, a narrower context that excludes `middle`. This can be enhanced by chaining a subsequent left-to-right decoding phase to re-generate `suffix` as

$$\begin{aligned} \texttt{suffix}^{\text{p}} &\sim p(\texttt{suffix}^{\text{p}} \mid \texttt{prefix}, \texttt{middle}), \\ \texttt{suffix}^{\text{c}} &\sim p(\texttt{suffix}^{\text{c}} \mid \texttt{prefix}, \texttt{middle}, \texttt{suffix}^{\text{p}}), \end{aligned}$$

allowing $\texttt{suffix} \coloneqq [\texttt{suffix}^{\text{p}}; \texttt{suffix}^{\text{c}}]$ to integrate information from both `prefix` and `middle`. Notably, the suffix prompt $\texttt{suffix}^{\text{p}}$ is also updated through looping to become fully contextual instead of being specified *a priori*.

This looping procedure can be continued to synchronize `middle` with the latest suffix information through the next self-infilling call. Instead of restarting self-infilling with `prefix` generation (Equation 1), which would ignore the information from the last left-to-right decoding and repeat the first iteration, hereafter self-infilling begins with Equation 5 to directly incorporate $\texttt{suffix}^{\text{p}}$.[2] To update the generated tokens of `prefix` (Equation 1), we prepend them into `middle` and reset `prefix` to the original fixed input. This design choice corroborates prior research (Bavarian et al., 2022), suggesting that while the infilled `middle` sometimes struggles to join `suffix`, it could adeptly continue `prefix`. During looping, transferring the output from self-infilling to left-to-right decoding is straightforward due to explicit sentinel tokens. However, to continue the looping

[2]It is also feasible to pass the entire generated `suffix` to initiate self-infilling with `middle` generation (Equation 3); however, this approach often yields inferior results, as shown in Table 4 and discussed in §3.4.

Algorithm 1 Looping Mechanism

```
Input: prompt, the language model, suffix prompt tokens $\texttt{suffix}^{\texttt{p}}$, and number of iterations $N$.
Output: Generated code $y$.

Set prefix ← prompt and $x \leftarrow$ [<PRE>;prefix];

for $n = 1, 2, \ldots, N$ do
    Invoke self-infilling generation (Algorithm 2) with
     ↪ input $x$ to output $x'$;
    Parse $x'$ into
     ↪ (prefix', middle', $\texttt{suffix}^{\texttt{p}\prime}$, $\texttt{suffix}^{\texttt{c}\prime}$);
    for p ∈ (prefix, middle, $\texttt{suffix}^{\texttt{p}}$, $\texttt{suffix}^{\texttt{c}}$) do
        ▷ Update each piece to its latest version.
        Update p ← p';

    Set $x \leftarrow$ [prefix;middle];

    Invoke left-to-right generation (Algorithm 3) with
     ↪ input $x$ to output $x'$;
    Parse $x'$ via l2r_parser() (Function 5) into
     ↪ (prefix', middle', $\texttt{suffix}^{\texttt{p}\prime}$, $\texttt{suffix}^{\texttt{c}\prime}$);
    for p ∈ (prefix, middle, $\texttt{suffix}^{\texttt{p}}$, $\texttt{suffix}^{\texttt{c}}$) do
        Update p ← p';

    if $n \neq N$ then
        ▷ Construct new input $x$ for the next cycle.
        $x \leftarrow$ [<PRE>;prefix;<SUF>;$\texttt{suffix}^{\texttt{p}}$];
    else
        ▷ Prepare the final output $y$.
        $y \leftarrow$ [prefix;middle;$\texttt{suffix}^{\texttt{p}}$;$\texttt{suffix}^{\texttt{c}}$];
Return output $y$.
```

mechanism from the left-to-right decoding phase, we have to parse the output into the quadruple (prefix, middle, suffix, $\texttt{suffix}^{\texttt{p}}$, $\texttt{suffix}^{\texttt{c}}$), as these segments are not explicitly delineated in left-to-right decodes. We explore several heuristic approaches for this segmentation in Function 5 and discuss them further in §3.4.

As outlined in Algorithm 1, the looping mechanism cyclically updates middle and suffix through self-infilling and left-to-right generation, respectively. Since each piece of the context is updated *in situ*, this looping process, akin to a rolling window over the context or piece-wise Gibbs sampling, leads to continuous synchronization of each section. In addition, this mechanism allows all tokens to be conditioned on a more informative bidirectional context, overcoming the causal limitations of traditional decoding.

3. Experiments

In this section, we present extensive experiments to evaluate self-infilling generation across various benchmarks. Please refer to Appendix B for an exhaustive overview of experimental details and Appendix C for additional results including illustrative generation samples.

3.1. Experimental Setup

Benchmarks. Our evaluation encompasses a range of code generation benchmarks, including HUMANEVAL (Chen et al., 2021), MBPP (Austin et al., 2021), and DS-1000 (Lai et al., 2023). In addition, we also extend our analysis to multilingual code generation with MULTIPL-E (Cassano et al., 2023) and mathematical reasoning with GSM8K (Cobbe et al., 2021), the detailed results of which can be found at Appendix C.1.

Code Language Models. For our experiments, we utilize the open-sourced STARCODER (Li et al., 2023) and CODE LLAMA (Roziere et al., 2023) models, which have been pre-trained with the FIM objective (Bavarian et al., 2022). Further model details are available in Appendix B.2.

Evaluation Protocols. Following Chen et al. (2021), we evaluate the performance of code language models with the pass@k rate, which estimates the probability of a code model generating a correct solution within k attempts. To facilitate a fair comparison to previous work (Lai et al., 2023; Li et al., 2023; Roziere et al., 2023), we report pass@1, pass@10, and pass@100 for the HUMANEVAL and MBPP benchmarks. We measure pass@1 for other tasks. Pass@1 rates are calculated via greedy decoding, while pass@10 and pass@100 are computed by generating 200 samples at temperature 0.8 using nucleus sampling (Holtzman et al., 2020) with top-$p = 0.95$. We set the maximum context length to 2048 tokens and limit the maximum number of generated tokens to 512, except for the HUMANEVAL benchmark, where we limit the context length to 640 for accelerating decoding. For self-infilling generation, τ and N are defaulted to 0.25 and 2, respectively, unless otherwise specified.

3.2. Results

Results on HUMANEVAL and MBPP. Table 1 displays the comparative results of self-infilling and traditional decoding approaches on HUMANEVAL and MBPP benchmarks. When self-infilling is solely equipped with its interruption functionality, without the looping mechanism (denoted as $N = 0$), its performance is marginally inferior to vanilla left-to-right completion baselines. This could be attributed to the inherent complexity of infilling tasks compared to next-token prediction, often resulting in difficulties in integrating the prefix with the suffix (Bavarian et al., 2022). Self-infilling further exacerbates these challenges by requiring the model to join the self-generated suffix with greater variety. Nonetheless, as indicated in §3.3, self-infilling contributes significantly to enhancing the structure of generated code and mitigating potential de-

Table 1. The pass@1(%), pass@10(%), and pass@100(%) rates on HUMANEVAL (zero-shot) and MBPP (three-shot) with different code language models. N denotes the number of times the decoding process goes through the loop, and $N = 0$ represents that the looping mechanism is not activated.[†] Results are taken from Roziere et al. (2023) and Xu et al. (2024).

Model	Size	Method	HUMANEVAL pass@1	HUMANEVAL pass@10	HUMANEVAL pass@100	MBPP pass@1	MBPP pass@10	MBPP pass@100
CODE LLAMA - INSTRUCT [†]	7B	Left-to-right	34.8	64.3	88.1	44.4	65.4	76.8
	13B		42.7	71.6	91.6	49.4	71.2	84.1
	34B		41.5	77.2	93.5	57.0	74.6	85.4
CODE LLAMA - PYTHON [†]	7B		38.4	70.3	90.6	47.6	70.3	84.8
	13B		43.3	77.4	94.1	49.0	74.0	87.6
	34B		53.7	82.8	94.7	56.2	76.4	88.2
Lemur[†]	70B		35.4	-	-	53.2	-	-
GPT-3.5 Turbo[†]	-		72.6	-	-	70.8	-	-
GPT-4 Turbo[†]	-		88.4	-	-	81.0	-	-
CODE LLAMA	7B	Left-to-right	34.1	59.6	86.5	42.8	66.8	82.3
		Self-infill ($N = 0$)	29.9	56.9	86.0	41.0	67.0	84.2
		Self-infill ($N = 1$)	34.1	61.3	87.0	43.8	67.6	83.9
		Self-infill ($N = 2$)	**39.0**	**62.5**	**88.5**	**44.8**	**67.9**	**84.5**
CODE LLAMA	13B	Left-to-right	35.4	69.7	90.0	47.2	71.9	87.3
		Self-infill ($N = 0$)	32.3	69.2	90.5	44.0	71.2	87.8
		Self-infill ($N = 1$)	38.4	70.7	**92.5**	47.2	72.6	88.3
		Self-infill ($N = 2$)	**40.8**	**72.1**	91.1	**49.0**	**73.0**	**89.2**
STARCODERBASE	15.5B	Left-to-right	31.7	56.3	80.3	43.8	68.7	85.3
		Self-infill ($N = 0$)	27.4	52.0	80.5	42.2	68.7	85.7
		Self-infill ($N = 1$)	33.5	56.6	82.3	44.6	**69.4**	**86.4**
		Self-infill ($N = 2$)	**36.0**	**59.4**	**84.6**	**46.6**	69.1	84.9
STARCODER	15.5B	Left-to-right	35.4	62.1	85.1	48.6	**71.5**	86.6
		Self-infill ($N = 0$)	29.2	58.0	85.9	46.4	70.7	85.6
		Self-infill ($N = 1$)	37.8	63.2	86.5	47.8	71.3	86.1
		Self-infill ($N = 2$)	**38.4**	**64.7**	**87.3**	**50.0**	71.1	**87.2**

generate issues. Besides, when integrated with the looping mechanism, self-infilling exhibits much higher code generation quality, scales effectively with increased iterations N, and consistently outperforms left-to-right baselines. The improvements even sometimes surpass those brought by specialized language training (e.g., CODE LLAMA - PYTHON) or instruction tuning (e.g., CODE LLAMA - INSTRUCT). These results suggest that the performance of *infill-capable* code models can be largely improved by integrating self-infilling capabilities into the inference phase, moving beyond the conventional left-to-right generation paradigm.

Results on DS-1000. The DS-1000 task supports code language models to generate in both *left-to-right* completion and *insertion* formats. In insertion mode, official suffixes are provided for each task (except for Matplotlib problems) to condition generation; while in left-to-right completion mode, these suffixes are translated into succinct natural language specifications and appended to problem descriptions to align with the causal formulation. In our evaluation, we follow the left-to-right format and use the given specifications to construct instance-wise suffix prompts for self-infilling. As shown in Table 2, our framework improves the performance over left-to-right completion and narrows the quality gap with the insertion mode even though without the use of official suffixes. Generation examples demonstrate that self-infilling more effectively adheres to the given specifications, such as allocating the final result to a specific variable for evaluation. However, we note that performance gains for STARCODER models are marginal, possibly due to their limited infilling training compared to CODE LLAMA series (Li et al., 2023; Roziere et al., 2023).

Results on Multilingual Code Generation. We further evaluate the multilingual performance of self-infilling generation across various programming languages. We recruit the MULTIPL-E benchmark (Cassano et al., 2023) and evaluate our approach for C++, Java, and PHP languages. Detailed

Table 2. Zero-shot pass@1(%) performance on DS-1000 with different code language models. † Results are taken from Li et al. (2023) and Luo et al. (2023).

Model	Method	Matplotlib	NumPy	Pandas	PyTorch	SciPy	Scikit-Learn	TensorFlow	Overall
CodeGen-16B-Mono†	Left-to-right	31.7	10.9	3.4	7.0	9.0	10.8	15.2	11.7
code-cushman-001†	Left-to-right	40.7	21.8	7.9	12.4	11.3	18.0	12.2	18.1
code-davinci-001†	Left-to-right	41.8	26.6	9.4	9.7	15.0	18.5	17.2	20.2
InCoder-6B†	Left-to-right	28.3	4.4	3.1	4.4	2.8	2.8	3.8	7.4
	Insertion	28.3	4.6	2.9	4.4	2.8	3.1	7.8	7.5
WizardCoder†	Left-to-right	55.2	33.6	16.7	26.2	24.2	24.9	26.7	29.2
	Insertion	55.2	35.1	20.4	30.4	28.9	32.3	37.8	32.8
code-davinci-002†	Left-to-right	57.0	43.1	26.5	41.8	31.8	44.8	39.3	39.2
	Insertion	57.0	46.5	30.1	47.7	34.8	53.7	53.4	43.3
CODE LLAMA 7B	Left-to-right	47.1	27.3	14.4	23.5	19.8	23.5	20.0	24.8
	Self-infilling	**48.4**	30.0	**17.2**	**27.9**	**22.6**	**38.3**	20.0	**28.7**
	Insertion	47.1	**30.9**	12.0	23.5	**22.6**	33.9	**35.6**	27.1
CODE LLAMA 13B	Left-to-right	49.0	33.2	18.6	30.9	**20.8**	37.4	31.1	30.3
	Self-infilling	49.7	33.2	**24.4**	32.4	**20.8**	47.0	26.7	33.1
	Insertion	**50.3**	**36.4**	21.6	**33.8**	**20.8**	**53.9**	**35.6**	**34.4**
STARCODERBASE	Left-to-right	**47.1**	**31.4**	9.6	26.5	**27.4**	38.3	17.8	26.9
	Self-infilling	45.8	29.1	10.0	23.5	25.5	**44.3**	20.0	26.7
	Insertion	45.8	**31.4**	**12.0**	**27.9**	26.4	42.6	**26.7**	**28.3**
STARCODER	Left-to-right	**51.6**	35.0	11.3	27.9	23.6	46.1	24.4	29.8
	Self-infilling	50.3	34.1	12.0	29.4	23.6	**47.0**	26.7	29.9
	Insertion	50.3	**37.7**	**13.7**	**35.3**	**24.5**	43.5	**31.1**	**31.5**

results are presented in Table 3, we observe a similar trend in improving conventional left-to-right generation approaches under the multilingual setting. These results indicate the versatility of self-infilling generation across various programming languages.

3.3. Analysis

Decoding Regularization. As discussed in §2.2, self-infilling facilitates decoding that respects specific constraints, particularly through interruption and suffix prompts. A significant advantage of this approach is its capacity to shape the generation and mitigate degeneracy, a prevalent issue where language models are prone to generating empty or repetitive programs (Holtzman et al., 2020; Zhang et al., 2023). We illustrate the effectiveness of such regularization in Figure 2, which depicts the frequency of *degenerate* samples for both vanilla and self-infilling decoding on HUMANEVAL. Notably, self-infilling significantly reduces the occurrence of degenerate outputs, while vanilla decoding produces more degenerate cases and displays a substantially heavier tail. This improvement is largely attributed to the interruption, which enforces the generation to end with a

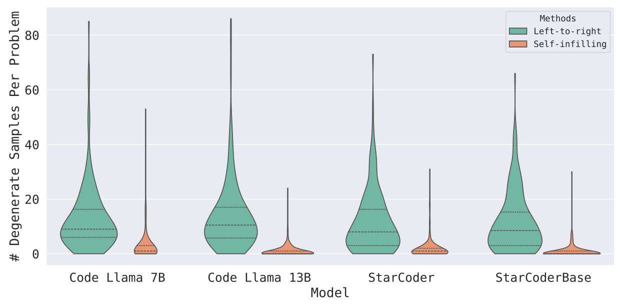


Figure 2. The distribution of degenerate solutions from self-infilling ($N = 2$) versus vanilla decoding on HUMANEVAL across various models. For each problem, 200 samples are generated using nucleus sampling with the temperature 0.8 and top-p 0.95.

fitting suffix, such as a return statement in function-level generation. Thanks to the suffix-first non-monotonic formulation, self-infilling effectively shapes the generation toward complete function programs. Our proposed approach makes it easy to implement such regularization while achieving higher performance, which is otherwise difficult to accomplish in conventional left-to-right decoding.

Table 3. Pass@1(%) rates on C++, Java, and PHP versions of HUMANEVAL problems from the MULTIPL-E benchmark (zero-shot). N denotes the number of times the decoding process goes through the loop, and $N=0$ represents that the looping mechanism is not activated.

Model	Size	Method	Language C++	Java	PHP
STARCODERBASE	15.5B	Left-to-right	30.4	27.8	25.5
		Self-infill ($N=0$)	31.1	26.6	26.1
		Self-infill ($N=1$)	**31.7**	**30.4**	26.1
		Self-infill ($N=2$)	30.4	**30.4**	**31.7**
STARCODER	15.5B	Left-to-right	31.1	27.8	24.8
		Self-infill ($N=0$)	31.7	27.2	25.5
		Self-infill ($N=1$)	31.1	**31.0**	26.1
		Self-infill ($N=2$)	**33.5**	29.7	**28.0**
CODE LLAMA	7B	Left-to-right	28.6	**33.5**	24.2
		Self-infill ($N=0$)	27.3	25.3	27.3
		Self-infill ($N=1$)	**31.7**	30.4	**29.8**
		Self-infill ($N=2$)	**31.7**	30.4	28.0
CODE LLAMA	13B	Left-to-right	38.5	34.8	35.4
		Self-infill ($N=0$)	36.0	34.2	29.2
		Self-infill ($N=1$)	39.1	35.4	**36.6**
		Self-infill ($N=2$)	**41.0**	**36.7**	35.4

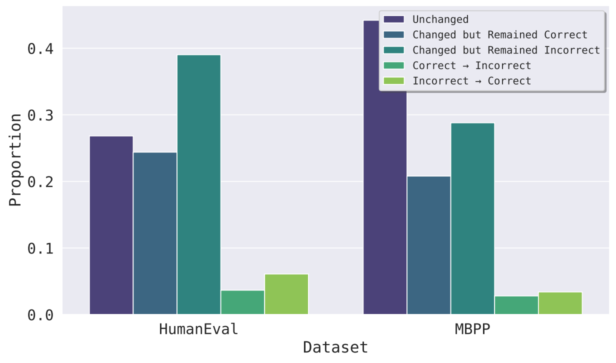


Figure 3. Proportional distribution of changes after a second iteration of the looping mechanism ($N = 2$) on HUMANEVAL and MBPP benchmarks with CODE LLAMA 13B. Categories illustrate the state changes of generated code: '*Unchanged*' denotes no change during the second time of looping, '*Changed but Remained Correct/Incorrect*' for changed snippets that stayed correct/incorrect, and '*Correct* $\rightarrow$ *Incorrect*' for snippets that changed from being correct to incorrect (vice versa).

Inspecting the Looping Mechanism. Different from research on self-improving frameworks (Madaan et al., 2023; Chen et al., 2023; Huang et al., 2023) in large language models, our looping mechanism (§2.3) operates independently of external tools or self-generated verbal feedback. Instead, the model dynamically modifies the generation *in-place* based on the most recent context. An in-depth examination of this mechanism is provided in Figure 3, which reveals that while overall task performance tends to improve with increasing iterations N, the looping mechanism does not intrinsically *improve* code quality. Rather, it simply updates code snippets with more informative contexts and broadens the decoding space, thereby increasing the likelihood of deriving correct solutions on average. It is possible that executing the loop can inadvertently introduce new bugs to initially correct solutions (e.g., the category '*Correct* $\rightarrow$ *Incorrect*') or fail to fix buggy programs (the '*Changed but Remained Incorrect*' category). We provide additional analyses in Appendix C.4 and Appendix C.6, including illustrative generated examples across various categories (Figures 18, 19, 20, 21 and 22).

3.4. Ablation Study

Additional ablation studies are deferred to Appendix C, including inspecting the effect of varying suffix prompts (Appendix C.2), examining the impact of removing self-infilling from the looping mechanism (Appendix C.3), and comparing looping with sample-and-rank approaches (Appendix C.4).

On the Effect of τ and N. τ and N are main hyperparameters in our self-infilling generation framework. The parameter τ determines the trigger point for the self-infilling interruption. Specifically, a larger τ value indicates a more aggressive approach towards activating self-infilling to regularize generation, and vice versa. Besides, the parameter N controls the duration of looped decoding execution. We conduct a detailed analysis to examine the influence of τ and N on the generation results, as presented in Figure 4 (as well as Figures 7, 8 and 9 in Appendix C.5). When the looping mechanism (§2.3) is inactive ($N = 0$), We observe that a larger τ typically correlates with slightly worse performance. This could be attributed to **1)** the complexity of infilling and **2)** the challenge in crafting an apt suffix according to a limited prefix; nevertheless, large τ values lead to better control over generation structures. For instance, nearly 13% of solutions generated by CODE LLAMA 7B on HUMANEVAL exhibit degeneracy at $\tau = 0.1$, which is reduced to 2.4% when increasing τ to 0.25. When equipped with the looping mechanism, most τ settings exhibit improved performance as N increases, despite some fluctuations at longer looping durations. Their performance generally surpasses the left-to-right completion baseline significantly. An exception is observed when $\tau = 0.0$, where self-infilling is not engaged in the first iteration. This case is equivalent to initializing the generation with left-to-right decoding, which is shown to benefit less from looping compared to other settings of τ. This trend may step from the inherently less structured nature of the left-to-right generation, narrowing the exploration space in subsequent iterations.

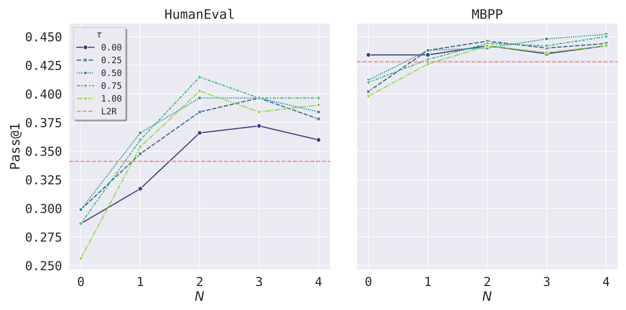


Figure 4. Results on HUMANEVAL and MBPP with different values of τ and N on CODE LLAMA 7B. $N = 0$ indicates that the looping mechanism is disabled, and the horizontal dashed line represents the performance of the vanilla left-to-right baseline (L2R).

On the Implementation of Looping Mechanisms. The introduced looping mechanism iteratively updates snippets of output to enhance decoding. A pivotal aspect of these iterations involves the transmission of updated contexts to the subsequent iteration (Algorithm 1). Unlike self-infilling, our mechanism's left-to-right decoding lacks a clear notion of $\texttt{suffix}^{\text{p}}$ and $\texttt{suffix}^{\text{c}}$, necessitating strategic output parsing to extract essential pieces for the next update cycle. We explore three distinct strategies for extraction: **1) Vanilla**, where the new $\texttt{suffix}^{\text{p}}$ starts at the beginning of updated `suffix` and ends at the occurrence of the default $\texttt{suffix}^{\text{p}}$; **2) Extended**, similar to Vanilla but starting from the midpoint of the whole generation to enlarge $\texttt{suffix}^{\text{p}}$; and **3) Half**, splitting the entire generation uniformly and employing the latter half as $\texttt{suffix}^{\text{p}}$, where the self-infilling reduces to standard infilling for Equation 3. These strategies reflect an increased amount of context information available for the subsequent iteration.

Our analysis (Table 4) indicates that CODE LLAMA models usually exhibit greater resilience to splitting strategy variations. Conversely, models like STARCODER generally benefit from a more informative suffix prompt, encompassing a broader context. This might step from less sufficient infilling training in STARCODER (only 50% of its pre-training time) compared to that in CODE LLAMA (90% of pre-training), which restricts its capability of (self-)infilling according to a limited context. In addition, the Half strategy, despite being more informative, often leads to slightly inferior results. This could be due to its complete suffix being overly specific and narrowing the solution space, compared to the other strategies that allow for more model-drive completion. Through our experiments, we utilize the Extended strategy for all code language models, except for CODE LLAMA 7B where strategy Vanilla is used.

Table 4. Results on HUMANEVAL and MBPP with different suffix splitting strategies in the looping mechanism.

Dataset	Strategy	Model			
		STARCODER		CODE LLAMA	
		15B-BASE	15B	7B	13B
HUMANEVAL	Vanilla	31.7	34.8	**37.2**	35.9
	Extended	36.0	37.8	32.3	**40.8**
	Half	**37.8**	**38.4**	31.7	39.6
MBPP	Vanilla	41.6	44.8	**44.6**	**49.0**
	Extended	**46.6**	**50.8**	43.8	**49.0**
	Half	43.8	48.8	43.8	46.8

4. Related Work

A variety of language models possess the capability to perform infilling (Sun et al., 2024). Encoder-only (Devlin et al., 2018; Liu et al., 2019; Joshi et al., 2020) and encoder-decoder architectures (Raffel et al., 2020; Lewis et al., 2020; Aghajanyan et al., 2021; Tay et al., 2023) are capable of conditioning on the bidirectional context, but they primarily focus on representation learning and fall short of generating coherent content. Standard left-to-right causal language models excel in generating high-quality text (Radford et al., 2018; 2019; Brown et al., 2020); however, their inherent causal formulation limits their effectiveness in infilling tasks that require bidirectional context. To circumvent these limitations, several approaches tailor the auto-regressive model architecture to generate tokens in a more flexible order beyond the standard left-to-right direction (Yang et al., 2019). Techniques such as integrating both left-to-right and right-to-left language models (Nguyen et al., 2023) have been effective in capturing bidirectional dependencies and thus facilitating infilling. Additionally, the ordering of generation can be made adaptive based on the model output or enhanced by learning a location predictor for each token (Stern et al., 2019; Gu et al., 2019; Chan et al., 2019; Welleck et al., 2019; Shen et al., 2020; Alon et al., 2020; Shen et al., 2023), further enhancing model flexibility in generation.

Another line of research enables infilling by transforming the input sequences while *retaining* the left-to-right auto-regressive formulation (Zhu et al., 2019; Donahue et al., 2020; Tay et al., 2022; Du et al., 2022; Aghajanyan et al., 2022; Bavarian et al., 2022; Fried et al., 2023). These methods reformat the input sequence by randomly selecting various spans and relocating them to the end of the sequence. The language model is then trained to predict tokens in the standard left-to-right auto-regressive manner but under this permuted sequence, which learns to infill considering both preceding and following content. This conceptually simple framework can be considered as extending span corruption objectives (Raffel et al., 2020; Tay et al., 2023; 2022; Anil et al., 2023), which are commonly used in training encoder-decoder Transformers, to the context of decoder-only language models. For instance, the causal masking objective (Aghajanyan et al., 2022; Yasunaga et al., 2022; Fried et al., 2023; Yu et al., 2023; Nijkamp et al., 2023a) samples a number of contiguous token spans at random, moves these spans to the end of the input sequence, and replaces the tokens at the original position with mask sentinel tokens. GLM (Du et al., 2022; Zeng et al., 2022) further generalizes the span corruption objective by randomly permuting the order among different spans to fully capture the inter-dependencies between different spans. Fill-In-the-Middle (FIM; Bavarian et al., 2022) employs a similar form as the causal masking objective but only samples a single span. These objectives are specially recruited in training several code language models, including recent versions of Codex (OpenAI et al., 2022), InCoder (Fried et al., 2023), SantaCoder (Allal et al., 2023), StarCoder (Li et al., 2023), StarCoder 2 (Lozhkov et al., 2024), CodeGen 2/2.5 (Nijkamp et al., 2023a), Code Llama (Roziere et al., 2023), DeepSeek-Coder (Guo et al., 2024), and CodeGemma (CodeGemma Team, 2024), facilitating numerous downstream tasks including partial code completion, docstring generation, return type prediction, and adaptive retrieval-augmented generation (Wu et al., 2024).

In this work, we extend the study of FIM objectives used in training language models and investigate their *built-in* self-infilling capability. Complementary to prior efforts in infilling training objectives, our work explores the advantages of imbuing infilling capabilities with the *decoding* process. While previous findings indicate that infilling can be learned in pre-training without (or slightly) compromising left-to-right generation quality (Bavarian et al., 2022; Li et al., 2023; Nijkamp et al., 2023a; Roziere et al., 2023), our findings suggest decoding can be much enhanced by incorporating the acquired infilling capability.

5. Conclusion

This work explores the built-in self-infilling capability of FIM-trained code language models, based on which we develop a code generation framework that integrates infilling with auto-regressive decoding. Our method extends traditional decoding to a non-monotonic process that supports interruption and looping mechanisms, allowing the model to defer the generation of some contexts and recursively update code snippets cyclically. Throughout extensive experiments, we demonstrate that self-infilling decoding significantly improves generation quality and regularity.

Limitations and Future Directions. Our findings suggest that the decoding behavior of language models can be effectively programmed and extended by harnessing their (self-)infilling abilities. This highlights the considerable potential of language models trained with diverse objectives like FIM (Bavarian et al., 2022), which not only maintain scalability (Bavarian et al., 2022; Tay et al., 2022; 2023; Anil et al., 2023) but also yield possibly improved generation quality. Besides, there are several interesting directions for future work, some of which we outline below:

- While our framework is primarily tailored for code generation tasks, its application to other domains, such as mathematical reasoning, offers an intriguing avenue for future exploration.
- The developed interruption and looping techniques present our initial attempts to exploit self-infilling. There exists potential to guide language models towards more structured generation, such as conforming to context-free grammars (Willard & Louf, 2023; Microsoft, 2023).
- Our self-infilling framework is limited to single-span infilling due to the formulation of FIM objectives. Extending self-infilling to accommodate multiple or nested spans is a compelling direction for future research.
- The looping mechanism developed in this work incurs additional computational overhead due to repeated context processing and decoding operations. Future work might include optimization of key-value caching and reuse across iterations to enhance efficiency.

Acknowledgements

We would like to thank the HKU NLP group and the anonymous reviewers for their valuable suggestions that greatly helped improve this work. Our manuscript especially benefited from insightful discussions with Yiheng Xu, Tianbao Xie, Chenxin An, and Hongjin Su. This work is partially supported by the joint research scheme of the National Natural Science Foundation of China (NSFC) and the Research Grants Council (RGC) under grant number N_HKU714/21.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none of which we feel must be specifically highlighted here.

References

Aghajanyan, A., Okhonko, D., Lewis, M., Joshi, M., Xu, H., Ghosh, G., and Zettlemoyer, L. Htlm: Hyper-text pre-training and prompting of language models. *arXiv preprint arXiv:2107.06955*, 2021.

Aghajanyan, A., Huang, B., Ross, C., Karpukhin, V., Xu, H., Goyal, N., Okhonko, D., Joshi, M., Ghosh, G., Lewis, M., et al. CM3: A causal masked multimodal model of the internet. *arXiv preprint arXiv:2201.07520*, 2022.

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

Alon, U., Sadaka, R., Levy, O., and Yahav, E. Structural language models of code. In *International conference on machine learning*, pp. 245–256. PMLR, 2020.

Anil, R., Dai, A. M., Firat, O., Johnson, M., Lepikhin, D., Passos, A., Shakeri, S., Taropa, E., Bailey, P., Chen, Z., et al. PaLM 2 Technical Report. *arXiv preprint arXiv:2305.10403*, 2023.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

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 preprint arXiv:2207.14255*, 2022.

Bengio, S., Vinyals, O., Jaitly, N., and Shazeer, N. Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in Neural Information Processing Systems*, volume 28, 2015.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901, 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

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 polyglot approach to benchmarking neural code generation. *IEEE Trans. Software Eng.*, 49(7):3675–3691, 2023.

Chan, W., Kitaev, N., Guu, K., Stern, M., and Uszkoreit, J. Kermit: Generative insertion-based modeling for sequences. *arXiv preprint arXiv:1906.01604*, 2019.

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

Chen, X., Lin, M., Schärli, N., and Zhou, D. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

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

CodeGemma Team, G. L. Codegemma: Open code models based on gemma, 2024. URL https://storage.googleapis.com/deepmind-media/gemma/codegemma_report.pdf.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Donahue, C., Lee, M., and Liang, P. Enabling language models to fill in the blanks. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 2492–2501, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.225. URL https://aclanthology.org/2020.acl-main.225.

Du, Z., Qian, Y., Liu, X., Ding, M., Qiu, J., Yang, Z., and Tang, J. Glm: General language model pretraining with autoregressive blank infilling. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 320–335, 2022.

Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, S., Zettlemoyer, L., and Lewis, M. InCoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=hQwb-lbM6EL.

Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., Callan, J., and Neubig, G. Pal: Program-aided language models. In *International Conference on Machine Learning*, pp. 10764–10799. PMLR, 2023.

Gu, J., Liu, Q., and Cho, K. Insertion-based decoding with automatically inferred generation order. *Transactions of the Association for Computational Linguistics*, 7:661–676, 2019.

Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., Chen, G., Bi, X., Wu, Y., Li, Y., et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

Holtzman, A., Buys, J., Du, L., Forbes, M., and Choi, Y. The curious case of neural text degeneration. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=rygGQyrFvH.

Huang, J., Chen, X., Mishra, S., Zheng, H. S., Yu, A. W., Song, X., and Zhou, D. Large language models cannot self-correct reasoning yet. *arXiv preprint arXiv:2310.01798*, 2023.

Jiang, S., Wang, Y., and Wang, Y. Selfevolve: A code evolution framework via large language models. *arXiv preprint arXiv:2306.02907*, 2023.

Joshi, M., Chen, D., Liu, Y., Weld, D. S., Zettlemoyer, L., and Levy, O. Spanbert: Improving pre-training by representing and predicting spans. *Transactions of the association for computational linguistics*, 8:64–77, 2020.

Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., and Iwasawa, Y. Large language models are zero-shot reasoners. In *Advances in Neural Information Processing Systems*, volume 35, pp. 22199–22213, 2022.

Lai, Y., Li, C., Wang, Y., Zhang, T., Zhong, R., Zettlemoyer, L., Yih, W.-t., Fried, D., Wang, S., and Yu, T. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pp. 18319–18345. PMLR, 2023.

Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., and Zettlemoyer, L. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 7871–7880, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.703. URL https://aclanthology.org/2020.acl-main.703.

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., et al. StarCoder: May the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022.

Liu, J., Xia, C. S., Wang, Y., and Zhang, L. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. RoBERTa: A robustly optimized BERT pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

Lozhkov, A., Li, R., Allal, L. B., Cassano, F., Lamy-Poirier, J., Tazi, N., Tang, A., Pykhtar, D., Liu, J., Wei, Y., et al. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

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 preprint arXiv:2306.08568*, 2023.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., et al. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*, 2023.

Microsoft. A guidance language for controlling large language models., 2023. URL https://github.com/microsoft/guidance.

Muennighoff, N., Liu, Q., Zebaze, A., Zheng, Q., Hui, B., Zhuo, T. Y., Singh, S., Tang, X., von Werra, L., and Longpre, S. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*, 2023.

Nguyen, A., Karampatziakis, N., and Chen, W. Meet in the middle: A new pre-training paradigm. *arXiv preprint arXiv:2303.07295*, 2023.

Ni, A., Iyer, S., Radev, D., Stoyanov, V., Yih, W.-t., Wang, S., and Lin, X. V. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pp. 26106–26128. PMLR, 2023.

Nijkamp, E., Hayashi, H., Xiong, C., Savarese, S., and Zhou, Y. CodeGen2: Lessons for training LLMs on programming and natural languages. *arXiv preprint arXiv:2305.02309*, 2023a.

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. In *The Eleventh International Conference on Learning Representations*, 2023b. URL https://openreview.net/forum?id=iaYcJKpY2B_.

OpenAI. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

OpenAI, Bavarian, M., Jiang, A., Jun, H., and Pondé, H. New GPT-3 Capabilities: Edit and Insert. *OpenAI blog*, 2022. URL https://openai.com/blog/gpt-3-edit-insert/.

Pan, L., Saxon, M., Xu, W., Nathani, D., Wang, X., and Wang, W. Y. Automatically correcting large language models: Surveying the landscape of diverse self-correction strategies. *arXiv preprint arXiv:2308.03188*, 2023.

Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al. Improving language understanding by generative pre-training. *OpenAI blog*, 2018.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language models are unsupervised multitask learners. *OpenAI blog*, 2019.

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. *Journal of Machine Learning Research*, 21: 140:1–140:67, 2020.

Ranzato, M., Chopra, S., Auli, M., and Zaremba, W. Sequence level training with recurrent neural networks. *arXiv preprint arXiv:1511.06732*, 2015.

Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Shen, T., Quach, V., Barzilay, R., and Jaakkola, T. Blank language models. *arXiv preprint arXiv:2002.03079*, 2020.

Shen, T., Peng, H., Shen, R., Fu, Y., Harchaoui, Z., and Choi, Y. Film: Fill-in language models for any-order generation. *arXiv preprint arXiv:2310.09930*, 2023.

Stern, M., Chan, W., Kiros, J., and Uszkoreit, J. Insertion transformer: Flexible sequence generation via insertion operations. In *International Conference on Machine Learning*, pp. 5976–5985. PMLR, 2019.

Sun, Q., Chen, Z., Xu, F., Cheng, K., Ma, C., Yin, Z., Wang, J., Han, C., Zhu, R., Yuan, S., et al. A survey of neural code intelligence: Paradigms, advances and beyond. *arXiv preprint arXiv:2403.14734*, 2024.

Tay, Y., Wei, J., Chung, H. W., Tran, V. Q., So, D. R., Shakeri, S., Garcia, X., Zheng, H. S., Rao, J., Chowdhery, A., et al. Transcending scaling laws with 0.1% extra compute. *arXiv preprint arXiv:2210.11399*, 2022.

Tay, Y., Dehghani, M., Tran, V. Q., Garcia, X., Wei, J., Wang, X., Chung, H. W., Bahri, D., Schuster, T., Zheng, S., Zhou, D., Houlsby, N., and Metzler, D. UL2: Unifying language learning paradigms. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=6ruVLB727MC.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. LLaMA: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023a.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023b.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837, 2022.

Welleck, S., Brantley, K., Iii, H. D., and Cho, K. Non-monotonic sequential text generation. In *International Conference on Machine Learning*, pp. 6716–6726. PMLR, 2019.

Willard, B. T. and Louf, R. Efficient guided generation for llms. *arXiv preprint arXiv:2307.09702*, 2023.

Wu, D., Ahmad, W. U., Zhang, D., Ramanathan, M. K., and Ma, X. Repoformer: Selective retrieval for repository-level code completion. *arXiv preprint arXiv:2403.10059*, 2024.

Xie, T., Zhao, S., Wu, C. H., Liu, Y., Luo, Q., Zhong, V., Yang, Y., and Yu, T. Text2reward: Automated dense reward function generation for reinforcement learning. *arXiv preprint arXiv:2309.11489*, 2023.

Xu, Y., Su, H., Xing, C., Mi, B., Liu, Q., Shi, W., Hui, B., Zhou, F., Liu, Y., Xie, T., Cheng, Z., Zhao, S., Kong, L., Wang, B., Xiong, C., and Yu, T. Lemur: Harmonizing natural language and code for language agents. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=hNhwSmtXRh.

Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R. R., and Le, Q. V. Xlnet: Generalized autoregressive pretraining for language understanding. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

Yasunaga, M., Aghajanyan, A., Shi, W., James, R., Leskovec, J., Liang, P., Lewis, M., Zettlemoyer, L., and Yih, W.-t. Retrieval-augmented multimodal language modeling. *arXiv preprint arXiv:2211.12561*, 2022.

Yu, L., Shi, B., Pasunuru, R., Muller, B., Golovneva, O., Wang, T., Babu, A., Tang, B., Karrer, B., Sheynin, S., et al. Scaling autoregressive multi-modal models: Pretraining and instruction tuning. *arXiv preprint arXiv:2309.02591*, 2023.

Zeng, A., Liu, X., Du, Z., Wang, Z., Lai, H., Ding, M., Yang, Z., Xu, Y., Zheng, W., Xia, X., et al. Glm-130b: An open bilingual pre-trained model. *arXiv preprint arXiv:2210.02414*, 2022.

Zhang, T., Yu, T., Hashimoto, T., Lewis, M., Yih, W.-t., Fried, D., and Wang, S. Coder reviewer reranking for code generation. In *International Conference on Machine Learning*, pp. 41832–41846. PMLR, 2023.

Zheng, W., Sharan, S. P., Jaiswal, A. K., Wang, K., Xi, Y., Xu, D., and Wang, Z. Outline, then details: Syntactically guided coarse-to-fine code generation. In *International Conference on Machine Learning*, pp. 42403–42419. PMLR, 2023.

Zhu, W., Hu, Z., and Xing, E. Text infilling. *arXiv preprint arXiv:1901.00158*, 2019.

Appendices

A. Pseudo-code

In practice, we implement our self-infilling interruption technique (§2.2; Algorithm 2) in a stateless manner through the manipulation of sentinel tokens. For completeness, we also include the pseudo-code of the conventional left-to-right generation process used in our looping mechanism (§2.3), as shown in Algorithm 3. Note that in both Algorithms 2 and 3 we omit detailed decoding setups, such as the use of temperature scaling and nucleus sampling.

Algorithm 2 Self-infilling Interruption

```
Input: Input sequence $x$, language model $f(\cdot;\boldsymbol{\theta})$, suffix prompt tokens $\texttt{suffix}^{\texttt{p}}$, stop condition $\mathbb{S}$, and probability threshold $\tau$.
Output: Self-infilled output.

for $t = 1, 2, \ldots$ do
    Calculate the probability distribution over the next token $p(x_t) \leftarrow f(x;\boldsymbol{\theta})$;
    if <PRE> $\in x$ and <SUF> $\notin x$ and <MID> $\notin x$ then
        ▷ Prefix generation.
        if $\max_v p(v) < \tau$ then
            ▷ In case the model becomes uncertain, interrupt and pivot to suffix generation.
            Set $p(x_t) = \delta_{x_t}($<SUF>$)$ to place all probability mass on sentinel token <SUF>;
    else if <PRE> $\in x$ and <SUF> $\in x$ and <MID> $\notin x$ then
        ▷ Suffix generation.
        Calculate the length of the current suffix $l_{\texttt{suffix}}$ and the suffix prompt $l_{\texttt{suffix}^{\texttt{p}}}$;
        if $l_{\texttt{suffix}} < l_{\texttt{suffix}^{\texttt{p}}}$ then
            ▷ Override the next token logit so that the suffix has to start with $\texttt{suffix}^{\texttt{p}}$.
            Set $p(x_t) = \delta_{x_t}(\texttt{suffix}^{\texttt{p}}[l_{\texttt{suffix}}])$;
    ▷ Middle generation (i.e., infilling) does not require extra post-processing.

    if stop condition $\mathbb{S}$ are met then
        if <PRE> $\in x$ and <SUF> $\in x$ and <MID> $\notin x$ then
            ▷ If the suffix is being generated, populate <MID> to signify the start of infilling.
            Set $p(x_t) = \delta_{x_t}($<MID>$)$;
        else
            ▷ Otherwise, the generation process is terminated.
            break
    Draw the next token $x_t \sim p(x_t)$ and append $x_t$ to the current input $x \leftarrow [x; x_t]$;
Return the updated sequence $x$.
```

Algorithm 3 Left-to-right Generation

```
Input: Input sequence $x$, language model $f(\cdot;\boldsymbol{\theta})$, and a list of stop tokens $\mathbb{S}$.
Output: Left-to-right decoded output.
for $t = 1, 2, \ldots$ do
    Calculate the probability distribution over the next token $p(x_t) \leftarrow f(x;\boldsymbol{\theta})$;

    if stop tokens $\mathbb{S}$ are met then
        ▷ The generation process is terminated if the current generation contains stop tokens.
        break
    Draw the next token $x_t \sim p(x_t)$ and append $x_t$ to the current input $x \leftarrow [x; x_t]$;
Return the updated sequence $x$.
```

Parsing Function for Left-to-right Outputs

```python
def l2r_parser(x, prompt, suffix_p, strategy):
    # x: The current output sequence
    # prompt: The original input prompt
    # suffix_p: The suffix prompt
    # strategy: The strategy used for splitting the output, as in Table 3 of Section 3.4.

    l_prompt = x.find(prompt) + len(prompt)
    # the index of midpoint of the whole generation excluding the original prompt
    l_half = l_prompt + (len(x) - l_prompt) // 2
    l_suffix_p_start = x.find(suffix_p, l_half)
    l_suffix_p_end = l_suffix_p_start + len(suffix_p)

    prefix = x[:l_prompt]
    if strategy == "Vanilla":
        middle   = x[l_prompt:l_suffix_p_start]
        suffix_p = x[l_suffix_p_start:l_suffix_p_end]
        suffix_c = x[l_suffix_p_end:]
    elif strategy == "Extended":
        middle   = x[l_prompt:l_half]
        suffix_p = x[l_half:l_suffix_p_end]
        suffix_c = x[l_suffix_p_end:]
    elif strategy == "Half":
        middle   = x[l_prompt:l_half]
        suffix_p = x[l_half:]
        suffix_c = ""

    return (prefix, middle, suffix_p, suffix_c)
```

Figure 5. Python pseudo-code implementation of the parsing function for the left-to-right generation.

B. Additional Experimental Details

B.1. Task Details

- **HUMANEVAL** (Chen et al., 2021) consists of 164 crafted programming challenges, each accompanied by multiple unit tests to evaluate the correctness of solutions generated by code models. We conduct evaluations in a zero-shot manner. Following previous practice (Li et al., 2023), for STARCODER series, we strip off the trailing newline symbol `\n` appearing at the end of the official prompt to align with their trained tokenizer (Microsoft, 2023); we adhere to the official prompt format in evaluation for the remaining models.
- **MULTIPL-E** (Cassano et al., 2023) is a multilingual benchmark extending HUMANEVAL to various programming languages. In this work, we report results of self-infilling generation for C++, Java, and PHP languages.
- **MBPP** (Austin et al., 2021) includes 500 crowd-sourced basic Python programming problems as the test set to evaluate code models. Following Austin et al. (2021), we use their provided 3-shot prompts for both STARCODER and CODE LLAMA model families.
- **DS-1000** (Lai et al., 2023) comprises 1,000 data science questions sourced from StackOverflow, aimed at benchmarking code models against real-world scenarios. The questions span various Python libraries commonly used in data science, including Matplotlib (155 questions), NumPy (220), Pandas (291), PyTorch (68), SciPy (106), Scikit-learn (115), and Tensorflow (45). We use the provided prompt to perform zero-shot evaluation. There are two distinct prompt formats in DS-1000 available for models considered in this work: the *left-to-right completion* format, which puts the entire instruction in the left context for regular left-to-right decoding; and the *insertion* format, which provides the instruction in both the left and right contexts. Our self-infilling mode follows the left-to-right format for generation, with further details elaborated below.
- **GSM8K** (Cobbe et al., 2021) comprises various grade school math word problems. We assess the performance of language models on the GSM8K test set, which includes 1,319 instances, using an 8-shot in-context example prompt.

B.2. Model Details

For code language models, our study mainly utilizes STARCODERBASE and STARCODER from the STARCODER family (Li et al., 2023), along with CODE LLAMA 7B and CODE LLAMA 13B from the CODE LLAMA series (Roziere et al., 2023),

Table 5. The pass@1(%), pass@10(%), and pass@100(%) rates on HUMANEVAL (zero-shot) with DeepSeek-Coder-Base models. N denotes the number of times the decoding process goes through the loop, and $N=0$ represents that the looping mechanism is not activated.

Model	Size	Method	HUMANEVAL		
			pass@1	pass@10	pass@100
DeepSeek-Coder-Base	1.3B	Left-to-right	31.1	52.0	78.3
		Self-infill ($N=0$)	27.4	48.4	72.7
		Self-infill ($N=1$)	32.3	52.9	79.1
		Self-infill ($N=2$)	**35.4**	**54.6**	**80.1**
DeepSeek-Coder-Base	6.7B	Left-to-right	47.0	75.2	92.1
		Self-infill ($N=0$)	36.6	70.5	89.6
		Self-infill ($N=1$)	46.3	76.2	92.1
		Self-infill ($N=2$)	**48.2**	**78.4**	**92.4**
DeepSeek-Coder-Base	33B	Left-to-right	49.4	81.4	93.1
		Self-infill ($N=0$)	49.4	78.9	92.2
		Self-infill ($N=1$)	**58.5**	83.0	94.1
		Self-infill ($N=2$)	**58.5**	**84.0**	**94.8**

Table 6. Zero-shot pass@1(%) performance on DS-1000 with DeepSeek-Coder-Base models. † Results are taken from Guo et al. (2024).

Model	**Method**	**Matplotlib**	**NumPy**	**Pandas**	**PyTorch**	**SciPy**	**Scikit-Learn**	**TensorFlow**	**Overall**
DeepSeek-Coder-Base 1.3B	Left-to-right†	32.3	21.4	9.3	8.8	8.5	16.5	8.9	16.2
	Left-to-right	35.5	22.3	9.6	7.4	7.5	27.8	11.1	18.2
	Self-infilling	36.8	21.4	9.6	7.4	11.3	27.0	11.1	18.5
	Insertion	35.5	20.9	5.2	7.4	13.2	13.0	15.6	15.7
DeepSeek-Coder-Base 6.7B	Left-to-right†	48.4	35.5	20.6	19.1	22.6	38.3	24.4	30.5
	Left-to-right	51.6	39.1	22.3	19.1	24.5	45.2	26.7	33.4
	Self-infilling	52.3	38.6	26.8	29.4	23.6	42.6	28.9	35.1
	Insertion	51.6	45.9	31.3	22.1	29.2	42.6	40.0	38.5
DeepSeek-Coder-Base 33B	Left-to-right†	56.1	49.6	25.8	36.8	36.8	40.0	46.7	40.2
	Left-to-right	60.6	51.8	28.2	35.3	34.9	48.7	46.7	42.8
	Self-infilling	61.9	52.3	28.9	44.1	34.0	47.8	46.7	43.7
	Insertion	61.3	49.5	32.0	47.1	29.2	47.0	60.0	44.1

respectively. Note that we do not evaluate CODE LLAMA 34B as it is not trained with the infilling objective (Roziere et al., 2023).

Self-infilling with DeepSeek-Coder-Base Models. We also evaluate the performance of DeepSeek-Coder-Base families (Guo et al., 2024), a series of recently released infilling-capable code language models. As detailed in Table 5 and Table 6, we observe significant improvements in performing self-infilling generation compared to conventional left-to-right approaches. Notably, this advantage scales well with the model size, where DeepSeek-Coder-Base 33B even demonstrates a 9% absolute increase in pass@1 rate on the HUMANEVAL benchmark. On DS-1000, the results are consistent with the trends observed for CODE LLAMA with DeepSeek-Coder-Base 6.7B and 33B models. In particular, our approach effectively narrows the gap between left-to-right completion and the insertion baseline (which provides more informative bidirectional contexts). However, DeepSeek-Coder-Base 1.3B exhibits a divergence from this trend, with the insertion baseline yielding slightly worse performance. This difference might be attributed to the model's smaller size, limiting its ability to generalize infilling capabilities to more diverse problems.

Self-infilling with Instruct Models. While these model families encompass other variants supporting infilling, such as CODE LLAMA - INSTRUCT, it is much more complicated to deal with the interplay between the instruction and infilling sentinel tokens, which might override the effect of one with the other. To examine this, we conduct evaluations of these instruct models, in particular Code Llama - Instruct and DeepSeek-Coder-Instruct, on HumanEval as well as HumanEval$^+$ (Liu et al., 2023) for a more rigorous and robust evaluation. Besides, we prepare the input prompt according to these models' respective formats. As detailed in Table 7, our findings reveal mixed results when applying self-infill generation to these language models. Generally, self-infilling does not significantly improve the performance of these instruct models, nor does increasing the looping time yield substantial benefits. This can be attributed to the *absence* of infilling training in the instruction fine-tuning stage, complicating the simultaneous application of instruction formatting and infilling patterns during decoding. This observation suggests a potential area for future research: exploring the usage of incorporating FIM during instruction fine-tuning to further improve self-infilling generation.

Self-infilling with More Flexible Infilling-capable Models. Notably, there are other open-sourced code models like InCoder (Fried et al., 2023) and CodeGen 2/2.5 (Nijkamp et al., 2023a) compatible with multi-span infilling. However, our preliminary experiments indicate these models are difficult to perform self-infilling, particularly in generating a coherent suffix from a given prefix prompt. This limitation may stem from their training paradigms, which involve more intricate span corruption objectives (Aghajanyan et al., 2022; Tay et al., 2023) as opposed to the FIM objective (Bavarian et al., 2022). In their input sequence construction, the same set of sentinel tokens marks both the suffix and middle segments. For example, a typical training instance might follow the format `[prefix; <Mask:0>; suffix; <Mask:0>; middle; <EOT>]`, with `<Mask:0>` indicating the start of both the suffix and middle segments. Consequently, these models may struggle to discern whether the ongoing generation pertains to the suffix or the infilled section. As a result, our study focuses on the StarCoder and Code Llama families, deferring an extensive analysis of more flexible infilling models to future work.

Table 7. The pass@1(%) rates on HumanEval and HumanEval$^+$ (both are evaluated zero-shot) benchmarks with different instruct code models.

Model	Size	Method	HumanEval	HumanEval$^+$
Code Llama - Instruct	7B	Left-to-right	43.9	38.4
		Self-infill ($N=0$)	42.7	37.2
		Self-infill ($N=1$)	45.1	39.6
		Self-infill ($N=2$)	**45.7**	**40.9**
Code Llama - Instruct	13B	Left-to-right	**43.3**	**36.6**
		Self-infill ($N=0$)	37.2	31.1
		Self-infill ($N=1$)	42.1	34.8
		Self-infill ($N=2$)	42.1	35.4
DeepSeek-Coder-Instruct	1.3B	Left-to-right	**68.3**	**63.4**
		Self-infill ($N=0$)	57.3	52.4
		Self-infill ($N=1$)	65.2	59.8
		Self-infill ($N=2$)	65.2	58.5
DeepSeek-Coder-Instruct	6.7B	Left-to-right	78.7	70.1
		Self-infill ($N=0$)	73.8	66.5
		Self-infill ($N=1$)	**79.3**	**72.0**
		Self-infill ($N=2$)	78.7	**72.0**
DeepSeek-Coder-Instruct	33B	Left-to-right	77.4	68.9
		Self-infill ($N=0$)	74.4	67.7
		Self-infill ($N=1$)	**79.3**	**71.3**
		Self-infill ($N=2$)	77.4	69.5

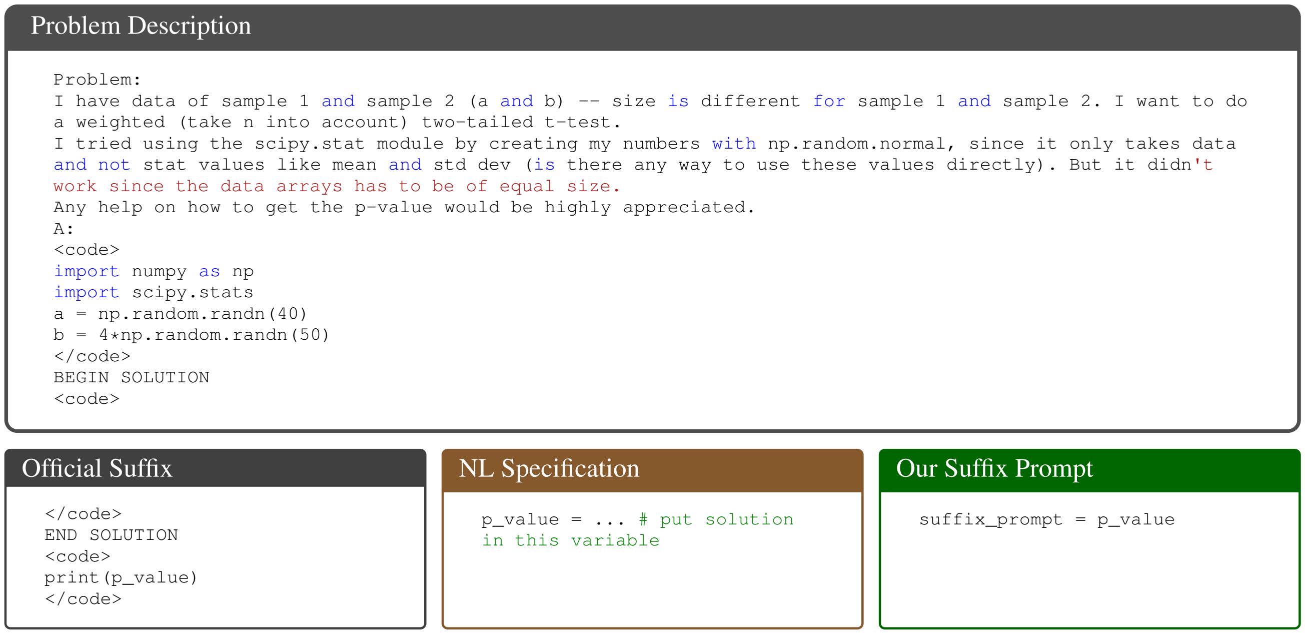
B.3. Self-infilling Implementation Details

In this section, we provide comprehensive details of our self-infilling implementation.

The Use of Stopping Criteria. Common code language models usually necessitate specific stopping conditions to appropriately terminate decoding, since they struggle to faithfully stop their generation as expected. This is often accomplished by monitoring for specific tokens in the generated code, such as incorporating markers like `<code>` and `</code>`; the decoding process is then terminated upon encountering the closing marker `</code>`. Another example is employing stop tokens indicative of the start of a new function, class implementation, or assertion during function-level generation. In our looping mechanism, where multiple rounds of decoding occur, we terminate both self-infilling and left-to-right generation upon these stop tokens and remove any extra content after the first met stop token.

Tokenization. Proper tokenization in infill-capable language models often poses challenges (Microsoft, 2023; Roziere et al., 2023), especially when different parts of the generation output (`prefix`, `middle`, and `suffix`) may break up tokens across piece boundaries. This can sometimes lead to irregular tokens and hurt performance due to out-of-distribution tokens. We apply a heuristic strategy to alleviate this by right-stripping all spaces from the current generation output, making the context more amenable to tokenization in the next iteration. However, this approach is sub-optimal to resolving irregular tokens and we leave a systematic investigation as future work.

Fallback for Infilling Failures. Sometimes self-infilling does not yield a coherent generation, such as failing to generate a well-defined suffix or producing an infilled middle without joining the suffix appropriately. To mitigate this issue and continue the looping mechanism, we employ a simple fallback strategy to reset the context when necessary. For self-infilling, if it fails to generate a proper suffix, we retain the prefix in the context and pass it to the subsequent left-to-right decoding; alternatively, if the middle fails to integrate with the given suffix, we truncate it to the last occurrence of the suffix prompt and use the resulting segment for left-to-right generation. Similarly, if the left-to-right decoding leads to degenerate outputs or fails to produce suffix prompt tokens, we revert to the context of the previous self-infilling call. Such fallback strategies ensure that even though some updating iterations encounter issues, we can still reset the context properly and continue the looping mechanism.



Problem Description

```
Problem:
I have data of sample 1 and sample 2 (a and b) -- size is different for sample 1 and sample 2. I want to do
a weighted (take n into account) two-tailed t-test.
I tried using the scipy.stat module by creating my numbers with np.random.normal, since it only takes data
and not stat values like mean and std dev (is there any way to use these values directly). But it didn't
work since the data arrays has to be of equal size.
Any help on how to get the p-value would be highly appreciated.
A:
<code>
import numpy as np
import scipy.stats
a = np.random.randn(40)
b = 4*np.random.randn(50)
</code>
BEGIN SOLUTION
<code>
```

Official Suffix

```
</code>
END SOLUTION
<code>
print(p_value)
</code>
```

NL Specification

```
p_value = ... # put solution
in this variable
```

Our Suffix Prompt

```
suffix_prompt = p_value
```

Figure 6. Illustration of suffix prompt construction for DS-1000 problems (Lai et al., 2023), where the official suffix for insertion mode is manually translated to a natural language (NL) specification and appended to the completion mode prompt. Self-infilling generation follows the completion format and utilizes the suffix prompt derived from the NL specification.

Suffix Prompts. For HUMANEVAL, MULTIPL-E, MBPP, and GSM8K benchmarks, which focus on function-level program synthesis, we set the default suffix prompt to **`return`**, controlling the structure of the generation while relying on minimal prior knowledge.

The DS-1000 benchmark covers a spectrum of code generation tasks. We adopt a simple heuristic to construct suffix prompts according to the provided input/output specification in a problem-wise manner. In particular, DS-1000 offers two prompt formats for evaluating code language models: **1)** insertion mode, where official prefixes and suffixes are given as input for each problem, except in Matplotlib tasks where a trailing context is absent; and **2)** left-to-right mode with only prefix prompts and manually translated natural language (NL) specifications for suffixes appended to prefixes. To evaluate the ability of self-infilling, we follow the left-to-right mode and devise suffix prompts from the translated NL specification for each problem, as illustrated in Figure 6. In general, our strategy proceeds as follows: if the problem requires completing a Python function, we set $\texttt{suffix}^{\text{p}}$ = "`return`"; if the problem requires storing the result in a particular variable named `var`, set $\texttt{suffix}^{\text{p}}$ = "`var`"[3]; if there is no such specification, we set $\texttt{suffix}^{\text{p}}$ = "`</code>`" that signifies the end of the generation; as for Matplotlib problems, we set $\texttt{suffix}^{\text{p}}$ = "`# SOLUTION END`" that indicates the end.

B.4. Evaluation Setup Details

Stop Criterion. Code language models usually require a list of stop tokens to signify the end of their generation. For HUMANEVAL, we recruit the same set of stop tokens following prior research (Chen et al., 2021; Li et al., 2023), including `["\nclass", "\ndef", "\n#", "\n@", "\nprint", "\nif", "\n```"]` that indicates the start of generation beyond the

[3]If there are multiple variables of interest, we simply adopt the last variable name as the suffix prompt.

Table 8. 8-shot accuracy on the GSM8K math-reasoning benchmark. Solutions are generated via chain-of-thought prompting (Wei et al., 2022; Kojima et al., 2022) with greedy decoding. † Results are taken from Li et al. (2023); Touvron et al. (2023a); Roziere et al. (2023).

Model	Size	Method	GSM8K PAL	GSM8K CoT
CodeGen-Multi†	16B	Left-to-right	8.6	3.2
CodeGen-Mono†	16B		13.1	2.6
StarCoderBase †	15.5B		21.5	8.4
StarCoderBase	15.5B	Left-to-right	21.5	6.1
		Self-infilling	21.3	6.4
StarCoder	15.5B	Left-to-right	23.9	5.8
		Self-infilling	24.9	7.0
Code Llama	7B	Left-to-right	27.4	9.7
		Self-infilling	29.4	10.2
Code Llama	13B	Left-to-right	37.7	17.7
		Self-infilling	39.8	21.6

current function scope. Following a similar spirit, [`"\nclass"`, `"\nassert"`, `'\n"""'`, `"\nprint"`, `"\nif"`, `"\n<|/"`, `"\n```"`, `"[DONE]"`] is used for MBPP. Default stop tokens are used for MultiPL-E (Cassano et al., 2023). For DS-1000, the default setup in Lai et al. (2023) adopts stop tokens [`"</code>"`, `"# SOLUTION END"`]. We extend the list to include [`"</code>"`, `"# SOLUTION END"`, `"\nEND SOLUTION"`], which more effectively truncates generated outputs and improves performance across both baselines and our method. For GSM8K, which employs a few-shot prompting format, the stop tokens are [`"\n\n\n"`, `"\nQ:"`].

On Evaluating Looping with Pass@k Metrics. When evaluating pass@k metrics for code generation, it is worth noting that the looping mechanism generates a single solution within an updated context iteratively, unlike multi-pass approaches where each pass yields an independent solution for subsequent selection and evaluation. The looping process is essentially part of a singular generative process and does not perform interim evaluations of functional correctness or involve selecting among different iterations. Consequently, our evaluation of pass@k metrics still accurately reflects functional correctness without skewing or biasing the results due to the looping process. This ensures our comparison to conventional left-to-right decoding on Pass@k remains valid.

C. Additional Experimental Results

C.1. Mathematical Reasoning with GSM8K

In this section, we extend our analysis to the GSM8K (Cobbe et al., 2021) mathematical reasoning task, employing methodologies from Program-Aided Language models (PAL; Gao et al., 2023) and Chain-of-Thought prompting (CoT; Wei et al., 2022). PAL solves mathematical reasoning problems by generating and executing Python programs to calculate answers, whereas CoT generates explicit intermediate steps in natural language to reach conclusions. For self-infilling in PAL, we set the default suffix prompt to **`return`**; while for CoT, we design the suffix prompt as follows,

$$\texttt{suffix}^{\texttt{p}} \coloneqq \text{“}\texttt{\textbackslash nTherefore, the answer is}\text{”}.$$

This formulation guides the suffix towards generating a conclusive response to the mathematical problem.

GSM8K results are presented in Table 8. We observe a consistent improvement in reasoning accuracy over left-to-right baselines with Code Llama models under the PAL format. However, similar to the DS-1000 benchmark, the benefits for StarCoder models are slight. In terms of CoT, which involves natural language instead of code, self-infilling decoding also leads to improved task accuracy in comparison to traditional left-to-right generation. These results not only highlight the versatility of self-infilling but also suggest its potential applicability in complicated reasoning tasks.

C.2. On the Effect of Different Suffix Prompts

Suffix prompts serve as another core component in our self-infilling framework, significantly influencing the decoding process and the structure of generation. To understand their effect, we conduct an experiment with varied suffix prompt

Table 10. Pass@1(%) Results on HUMANEVAL using different looping mechanisms. N denotes the number of iterations and t denotes the temperature for rejection sampling. SI denotes Self-Infilling.

Decoding Method	Setup	Model			
		STARCODER		CODE LLAMA	
		15B-BASE	15B	7B	13B
Rejection Sampling	$t=0.3$	32.3	35.4	34.8	37.8
	$t=0.5$	31.7	35.4	35.4	36.6
	$t=0.8$	32.9	37.2	34.1	38.4
Loop w/o SI	$N=0$	31.7	35.4	34.1	35.4
	$N=1$	31.1	34.1	31.7	34.8
	$N=2$	31.7	36.0	31.1	32.9
Loop w/ SI (ours)	$N=0$	27.4	29.2	29.9	32.3
	$N=1$	33.5	37.8	34.1	38.4
	$N=2$	36.0	38.4	39.0	40.8
Left-to-right	-	31.7	35.4	34.1	35.4

configurations, as detailed in Table 9. For these experiments, we use $N = 1$ for the looped mechanism to facilitate a direct comparison among different suffix prompts. For HUMANEVAL, we explore four suffix prompt variants, including 1) an empty string, 2) the default choice with the **`return`** keyword, 3) a dynamic strategy `$ARG_NAME` using the function's first argument name, as well as 4) a full return statement by specifying a variable `result`. We observe that the default suffix prompt choice yields the best performance across all model types, but choices 3) and 4) also demonstrate effectiveness. This highlights the potential benefits of enhancing self-infilling through refined suffix prompt design. We conduct a similar evaluation scheme for DS-1000 and find STARCODER models are robust to varying suffix prompts, while CODE LLAMA models benefit from more informative suffix prompts. Interestingly, CODE LLAMA models tend to produce empty suffixes with an empty suffix prompt, likely a consequence of the pre-training phase, where `<SUF>` might be commonly followed by `<MID>` and thus empty suffixes are preferred.

C.3. A Looping Mechanism without Self-infilling

In this section, we investigate the impact of removing the self-infilling component from the looping mechanism. In particular, we design a looping mechanism without self-infilling by initiating with left-to-right decoding, manually extracting the latter part of the completion as `suffix`, and then using the standard infilling operator to regenerate `middle` based on the extracted `suffix`. This roughly corresponds to a right-shifted version of looping in §2.3 (instead of starting with self-infilling and then performing left-to-right decoding, this variant starts from left-to-right decoding to self-infilling). As shown in Table 10, our findings indicate that looping under these conditions typically does not enhance coding performance, and extending the looping time does not yield improvements. Furthermore, this looping mechanism even leads to lower pass rates sometimes. This may be attributed to the specificity of `suffix` generated through left-to-right decoding, which potentially restricts the possible outcomes of `middle`. In addition, left-to-right decoding does not regularize the structure of generated output, which might be degenerate and thus lead to degenerate `suffix` as well. Self-infilling, on the other hand, effectively scaffolds the overall generation and regenerates `suffix` based on a succinct $\texttt{suffix}^{\texttt{p}}$, encouraging the model to explore a larger decoding space. This ablation study underscores the value of the self-infilling component in the looping mechanism.

Table 9. Results on HUMANEVAL and DS-1000 with different suffix prompts.

Dataset	Suffix Prompt	Model			
		STARCODER		CODE LLAMA	
		15B-BASE	15B	7B	13B
HUMANEVAL	""	29.9	34.1	33.5	35.4
	`"return"` **(default)**	33.5	37.8	34.1	38.4
	`"$ARG_NAME"`	26.8	31.1	29.2	37.8
	`"\nreturn result\n"`	32.9	38.4	30.5	37.2
DS-1000	""	27.0	28.6	24.7	27.6
	`"$ARG_NAME"` **(default)**	27.0	29.9	28.1	31.6
	`"\n</code>\n"`	26.9	30.0	23.6	27.1

We also compare our approach to another iterative variant with rejection sampling. This method initially uses greedy decoding and falls back to stochastic sampling if the initial generation degenerates (that is, lacking a return keyword or becoming empty). We limit the rejection process to a maximum of 10 trials to prevent infinite loops. The temperature settings were varied, with top_p set at 0.95. The results indicate that while rejection sampling reduces degenerate behaviors (typically requiring 2-3 sampling attempts for acceptance), it does not significantly improve pass rates compared to self-infilling.

Additionally, we did not find a significant correlation between pass rates, sampling times, and temperature settings. This evidence suggests that the advantages of self-infilling extend beyond simply addressing degeneration biases.

C.4. Comparison to Sample-and-Rank Baselines

The introduced looping mechanism (§2.3) in place updates pieces of the generation multiple times, which produces multiple possible generations along the iterative process. An alternative approach to accomplishing this is performing beam search, or simply sampling multiple generations in the conventional left-to-right way, followed by selection based on the highest probability or average token log-likelihood (Chen et al., 2021; Zhang et al., 2023). Our comparative analysis with these multi-sample baselines, as presented in Table 11 on HUMANEVAL, reveals that our looping mechanism, despite lacking an explicit selection or aggregation step, performs competitively with these baselines. Furthermore, when complemented by a ranking component — first generating multiple samples via self-infilling and then selecting based on the highest average likelihood — our method demonstrates superior performance over existing baselines. These results highlight the effectiveness of the looping mechanism, which encourages code language models to explore a broader decoding space.

Our looping mechanism is also relevant to multi-pass code generation approaches in the literature (Chen et al., 2023; Jiang et al., 2023; Pan et al., 2023). Unlike these approaches Chen et al. (2023), we demonstrate that base code models, even without instruction-following abilities, can exhibit improved generation quality via looping. Our approach alternates between left-to-right decoding and self-infilling, relying solely on the pre-trained code model without additional training. Nevertheless, our work can be further augmented with other signals like syntactic representation (Zheng et al., 2023) and execution results (Ni et al., 2023) to generate better samples.

C.5. Additional Ablation Studies

This section provides additional ablation studies on the effect of hyper-parameters τ and N under various code language models, including CODE LLAMA 13B (Figure 7), STARCODER (Figure 8), and STARCODERBASE (Figure 9). Note that setting $\tau = 0$ and $N = 0$ reduces the looping mechanism to conventional left-to-right decoding. However, this configuration still adds a sentinel `<PRE>` to the beginning of the input prompt, potentially leading to a subtle performance difference.

C.6. Examples of Self-infilling

In this section, we present additional examples (Figures 10, 11, 12 and 13 for HUMANEVAL and Figures 14, 15 and 16 for DS-1000 problems) to illustrate the distinct decoding behaviors of the self-infilling approach compared to traditional left-to-right decoding. Besides, we also provide demonstrations of self-infilling generation with the interruption and looping mechanism, as shown in Figures 17, 18, 19, 20, 21 and 22.

These examples are generated using CODE LLAMA 13B with greedy decoding. For the sake of brevity and clarity, the occurrence of stop tokens during generation is omitted. Also note that the sentinel token `<PRE>` used in self-infilling, which is usually positioned at the beginning of the problem description, is not displayed in these illustrations for readability.

Table 11. Comparison results of self-infilling against various multi-sample baselines on HUMANEVAL. N denotes the number of iterations in our looping mechanism, while S indicates the beam size for Beamsearch and the number of samples for other approaches. Samples are generated with nucleus sampling at temperature 0.3 and top-p 0.95. The *Rank* strategy selects the generation with the highest mean token log probabilities (Chen et al., 2021).

Method	Setup	Model			
		STARCODER		CODE LLAMA	
		15B-BASE	15B	7B	13B
Greedy	$S = 1$	31.7	35.4	34.1	35.4
Beamsearch	$S = 2$	36.0	37.2	34.1	40.8
	$S = 4$	34.8	**40.2**	16.5	19.5
Sample	$S = 1$	29.3	30.5	29.3	34.8
Rank	$S = 2$	36.0	37.2	37.2	37.2
	$S = 4$	32.9	36.6	32.3	34.1
Ours	$S = 1, N = 1$	33.5	37.8	34.1	38.4
	$S = 1, N = 2$	36.0	38.4	**39.0**	40.8
Ours + Rank	$S = 2, N = 1$	34.8	**40.2**	33.5	**42.1**
	$S = 2, N = 2$	**37.8**	37.8	36.5	40.2

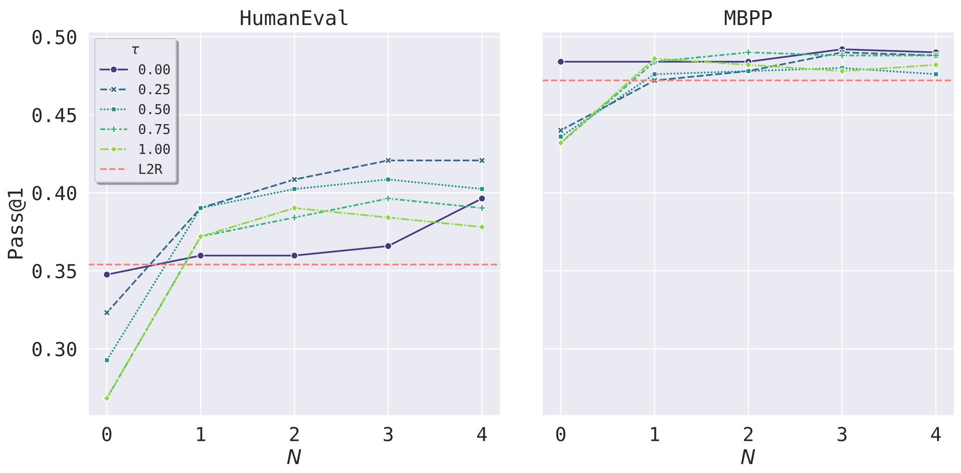


Figure 7. Results on HUMANEVAL and MBPP with different values of probability threshold τ and looping times N on CODE LLAMA 13B. $N = 0$ indicates the looping mechanism is disabled, and the horizontal dashed line represents the performance of the vanilla left-to-right baseline (L2R).

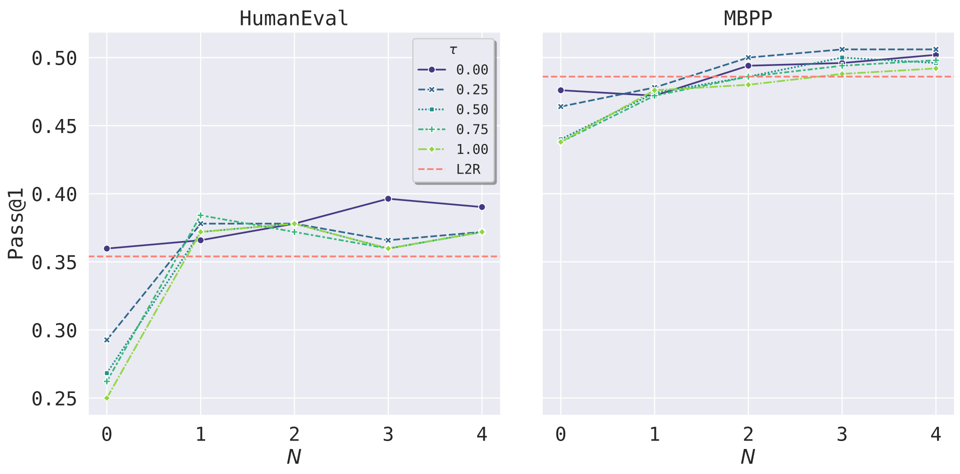


Figure 8. Results on HUMANEVAL and MBPP with different values of probability threshold τ and looping times N on STARCODER. $N = 0$ indicates that the looping mechanism is disabled, and the horizontal dashed line represents the performance of the vanilla left-to-right baseline (L2R).

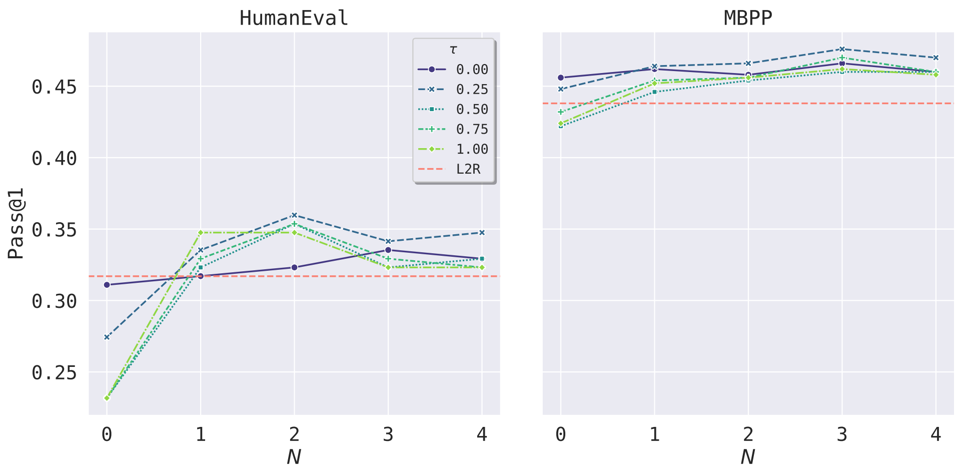


Figure 9. Results on HUMANEVAL and MBPP with different values of probability threshold τ and looping times N on STARCODERBASE. $N = 0$ indicates that the looping mechanism is disabled, and the horizontal dashed line represents the performance of the vanilla left-to-right baseline (L2R).

Problem Description

```
from typing import List


def remove_duplicates(numbers: List[int]) -> List[int]:
    """ From a list of integers, remove all elements that occur more than once.
    Keep order of elements left the same as in the input.
    >>> remove_duplicates([1, 2, 3, 2, 4])
    [1, 3, 4]
    """
```

Vanilla Generation

```
    # TODO: implement this function
    return []
```

Self-infilled Generation

```
    <SUF>return numbers


<MID> seen = set()
    for number in numbers:
        if number in seen:
            numbers.remove(number)
        else:
            seen.add(number)
    <EOT>
```

Figure 10. An illustration of self-infilling decoding with **interruption** on HUMANEVAL: vanilla left-to-right generation exhibits degenerate behaviors that output an empty program due to the initially generated comment; however, self-infilling early interrupts the decoding flow and drafts a plausible suffix first, which then drives subsequent decoding towards joining that suffix.

Problem Description

```
def is_happy(s):
    """You are given a string s.
    Your task is to check if the string is happy or not.
    A string is happy if its length is at least 3 and every 3 consecutive letters are distinct
    For example:
    is_happy(a) => False
    is_happy(aa) => False
    is_happy(abcd) => True
    is_happy(aabb) => False
    is_happy(adb) => True
    is_happy(xyy) => False
    """
```

Vanilla Generation

```
    if len(s) < 3:
        return False
    for i in range(len(s) - 2):
        if s[i] == s[i + 1] == s[i + 2]:
            return False
    return True
```

Self-infilled Generation

```
    if len(s) < 3:
        return False
    <SUF>return True


<MID> for i in range(len(s) - 2):
        if s[i] == s[i + 1] == s[i + 2]:
            return False
    <EOT>
```

Figure 11. An illustration of self-infilling decoding with **interruption** on HUMANEVAL: both vanilla left-to-right and self-infilling generation produce the same code snippet, despite through different decoding trajectories.

Problem Description

```
def total_match(lst1, lst2):
    '''
    Write a function that accepts two lists of strings and returns the list that has
    total number of chars in the all strings of the list less than the other list.

    if the two lists have the same number of chars, return the first list.

    Examples
    total_match([], []) -> []
    total_match(['hi', 'admin'], ['hI', 'Hi']) -> ['hI', 'Hi']
    total_match(['hi', 'admin'], ['hi', 'hi', 'admin', 'project']) -> ['hi', 'admin']
    total_match(['hi', 'admin'], ['hI', 'hi', 'hi']) -> ['hI', 'hi', 'hi']
    total_match(['4'], ['1', '2', '3', '4', '5']) -> ['4']
    '''
```

Vanilla Generation

```
    # lst1 = ['hi', 'admin']
    # lst2 = ['hI', 'Hi']
    # lst1 = ['hi', 'admin']
    # lst2 = ['hi', 'hi', 'admin', 'project']
    # lst1 = ['hi', 'admin']
    # lst2 = ['hI', 'hi', 'hi']
    # lst1 = ['4']
    # lst2 = ['1', '2', '3', '4', '5']
    # lst1 = ['hi', 'admin']
    # lst2 = ['hI', 'hi', 'hi']
    # lst1 = ['hi', 'admin']
    # lst2 = ['hI', 'hi', 'hi']
    # lst1 = ['hi', 'admin']
    # lst2 = ['hI', 'hi', 'hi']
    # lst1 = ['hi', 'admin']
    ...
```

Self-infilled Generation

```
    #<SUF>return lst1
    else:
        return lst2


<MID> Write your code here.
    total_chars_lst1 = 0
    total_chars_lst2 = 0
    for i in lst1:
        total_chars_lst1 += len(i)
    for i in lst2:
        total_chars_lst2 += len(i)
    if total_chars_lst1 < total_chars_lst2:
        <EOT>
```

Figure 12. An illustration of self-infilling decoding with **interruption** on HUMANEVAL: vanilla left-to-right generation tends to repeat the same snippet; self-infilling first starts the program with a symbol # that signifies a comment; however, the specific content within the comment is uncertain to the model, which thus issues an interruption of current decoding and proceeds to the suffix, circumventing a context prone of degeneracy.

Problem Description

```
    def get_closest_vowel(word):
        """You are given a word. Your task is to find the closest vowel that stands between two consonants from
    the right side of the word (case sensitive).

        Vowels in the beginning and ending doesn't count. Return empty string if you didn't find any vowel met
    the above condition.

        You may assume that the given string contains English letter only.

        Example:
        get_closest_vowel("yogurt") ==> "u"
        get_closest_vowel("FULL") ==> "U"
        get_closest_vowel("quick") ==> ""
        get_closest_vowel("ab") ==> ""
        """
```

Vanilla Generation

```
    vowels = "aeiouAEIOU"
    word = word[::-1]
    for i in range(len(word)):
        if word[i] in vowels:
            if i == 0:
                return ""
            else:
                return word[i]
    return ""
```

Self-infilled Generation

```
    vowels =<SUF>return vowels[0]
    else:
        return vowels[-1]


<MID> "aeiouAEIOU"
    word = word[::-1]
    for i in range(len(word)):
        if word[i] in vowels:
            if i != 0:
                return word[i]
            else:
                return ""
    return ""
```

Figure 13. An illustration of self-infilling decoding with **interruption** on HUMANEVAL: both vanilla left-to-right and self-infilling generation produce an incorrect code snippet. While self-infilling is invoked to yield a suffix first, it fails to complete the infilling operation and keeps generating the code *without* respecting the previously produced suffix.

Problem Description

```
Problem:

I have used the

sklearn.preprocessing.OneHotEncoder
to transform some data the output is scipy.sparse.csr.csr_matrix how can I merge it back into my original
dataframe along with the other columns?

I tried to use pd.concat but I get

TypeError: cannot concatenate a non-NDFrame object
Thanks

A:

<code>
import pandas as pd
import numpy as np
from scipy.sparse import csr_matrix
df_origin, transform_output = load_data()
def solve(df, transform_output):
    # return the solution in this function
    # result = solve(df, transform_output)
    ### BEGIN SOLUTION
```

Vanilla Generation

```
    # TODO: Implement your solution here
    #
    #
    #
    #
    #
    # ...
```

Self-infilled Generation

```
    <SUF>return result

result = solve(df_origin, transform_output)
<MID> result = pd.concat([df, transform_output],
axis=1)
    ### END SOLUTION
    <EOT>
```

Figure 14. An illustration of self-infilling decoding with **interruption** on DS-1000: vanilla left-to-right generation exhibits degenerate behaviors that output an empty program since it produces a misleading comment at the very beginning of the solution; however, self-infilling early drafts a plausible suffix first, which in turn drives the overall generation process to yield a plausible solution.

Problem Description

```
Problem:

I may be missing something obvious, but I can't find a way to compute this.

Given two tensors, I want to keep elements with the minimum absolute values, in each one of them as well as
the sign.

I thought about

sign_x = torch.sign(x)
sign_y = torch.sign(y)
min = torch.min(torch.abs(x), torch.abs(y))
in order to eventually multiply the signs with the obtained minimums, but then I have no method to multiply
the correct sign to each element that was kept and must choose one of the two tensors.


A:

<code>
import numpy as np
import pandas as pd
import torch
x, y = load_data()
</code>
signed_min = ... # put solution in this variable
BEGIN SOLUTION
<code>
```

Vanilla Generation

```
sign_x = torch.sign(x)
sign_y = torch.sign(y)
min = torch.min(torch.abs(x), torch.abs(y))
</code>
```

Self-infilled Generation

```
<SUF>signed_min = torch.sign(x) *
torch.min(torch.abs(x), torch.abs(y))
</code><MID><EOT>
```

Figure 15. An illustration of self-infilling decoding with **interruption** on DS-1000: vanilla left-to-right generation follows the description well but does not store the result to the specified variable; instead, self-infilling ensures the overall generation must include the expected variable through suffix prompting.

Problem Description

```
Problem:
I am trying to find col duplicates rows in a pandas dataframe.
df=pd.DataFrame(data=[[1,1,2,5],[1,3,4,1],[4,1,2,5],[5,1,4,9],[1,1,2,5]],columns=['val',
'col1','col2','3col'])
df
Out[15]:
   val  col1  col2  3col
0    1     1     2     5
1    1     3     4     1
2    4     1     2     5
3    5     1     4     9
4    1     1     2     5
duplicate_bool = df.duplicated(subset=['col1','col2', '3col'], keep='first')
duplicate = df.loc[duplicate_bool == True]
duplicate
Out[16]:
   val  col1  col2  3col
2    1     1     2      5
4    1     1     2      5


Is there a way to add a column referring to the index of the first duplicate (the one kept)
duplicate
Out[16]:
   val  col1  col2 3col   index_original
2     4    1     2      5          0
4     1    1     2      5          0


Note: df could be very very big in my case....


A:
<code>
import pandas as pd


df=pd.DataFrame(data=[[1,1,2,5],[1,3,4,1],[4,1,2,5],[5,1,4,9],[1,1,2,5]],columns=['val',
'col1','col2','3col'])
</code>
result = ... # put solution in this variable
BEGIN SOLUTION
<code>
```

Vanilla Generation

```
df['index_original'] = df.groupby(['col1','col2',
'3col']).cumcount()
</code>
```

Self-infilled Generation

```
<SUF>result = df.loc[duplicate_bool == True]
result['index_original'] = df.loc[duplicate_bool
== True].index.values
result = result.reset_index(drop=True)
</code><MID>duplicate_bool =
df.duplicated(subset=['col1','col2', '3col'],
keep='first')
<EOT>
```

Figure 16. An illustration of self-infilling decoding with **interruption** on DS-1000: vanilla left-to-right generation adheres to the description but fails to allocate the result to the designated variable. In contrast, self-infilling guarantees that the final output is correctly assigned to the expected variable via suffix prompting.

Problem Description

```
def get_row(lst, x):
    """
    You are given a 2 dimensional data, as a nested lists,
    which is similar to matrix, however, unlike matrices,
    each row may contain a different number of columns.
    Given lst, and integer x, find integers x in the list,
    and return list of tuples, [(x1, y1), (x2, y2) ...] such that
    each tuple is a coordinate - (row, columns), starting with 0.
    Sort coordinates initially by rows in ascending order.
    Also, sort coordinates of the row by columns in descending order.

    Examples:
    get_row([
      [1,2,3,4,5,6],
      [1,2,3,4,1,6],
      [1,2,3,4,5,1]
    ], 1) == [(0, 0), (1, 4), (1, 0), (2, 5), (2, 0)]
    get_row([], 1) == []
    get_row([[], [1], [1, 2, 3]], 3) == [(2, 2)]
    """
```

$N = 1$: Self-infilling

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

      <SUF>return []
       result = []
       for i in range(len(lst)):
           for j in range(len(lst[i])):
               if lst[i][j] == x:
                   result.append((i, j))
       result.sort(key=lambda x: (x[0], -x[1]))
       return result


   <MID>    if not lst:
           <EOT>
```

$N = 1$: Left-to-right

```
       if not lst:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

           return []
       result = []
       for i, row in enumerate(lst):
           if x in row:
               result.append((i, row.index(x)))
       return sorted(result, key=lambda x: (x[0],
   -x[1]))
```

$N = 2$: Self-infilling

```
   <SUF>    for i, row in enumerate(lst):
           if x in row:
               result.append((i, row.index(x)))
       return
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

               sorted(result, key=lambda x: (x[0],
   -x[1]))

   <MID>    result = []
   <EOT>
```

$N = 2$: Left-to-right

```
       result = []
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

       for i, row in enumerate(lst):
           for j, col in enumerate(row):
               if col == x:
                   result.append((i, j))
       return sorted(result, key=lambda x: (x[0],
   -x[1]))
```

Figure 17. An illustration of self-infilling decoding with **interruption** and **looping** on HUMANEVAL, which successfully corrects the initially wrong solution at $N = 1$ (classified as the 'Incorrect $\rightarrow$ Correct' category in Figure 3). The area above the dashed line indicates the current context, while the content below represents the corresponding completion.

Problem Description

```
def remove_vowels(text):
    """
    remove_vowels is a function that takes string and returns string without vowels.
    >>> remove_vowels('')
    ''
    >>> remove_vowels("abcdef\nghijklm")
    'bcdf\nghjklm'
    >>> remove_vowels('abcdef')
    'bcdf'
    >>> remove_vowels('aaaaa')
    ''
    >>> remove_vowels('aaBAA')
    'B'
    >>> remove_vowels('zbcd')
    'zbcd'
    """
```

$N = 1$: Self-infilling

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    vowels = 'aeiou'
   <SUF>return text

<MID> for vowel in vowels:
        text = text.replace(vowel, '')
    <EOT>
```

$N = 1$: Left-to-right

```
    vowels = 'aeiou'
    for vowel in vowels:
        text = text.replace(vowel, '')
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    return text
```

$N = 2$: Self-infilling

```
<SUF>        text = text.replace(vowel, '')
    return
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

          text
<MID>    vowels = ['a', 'e', 'i', 'o', 'u', 'A',
'E', 'I', 'O', 'U']
    for vowel in vowels:
<EOT>
```

$N = 2$: Left-to-right

```
    vowels = ['a', 'e', 'i', 'o', 'u', 'A', 'E',
'I', 'O', 'U']
    for vowel in vowels:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

        text = text.replace(vowel, '')
    return text
```

Figure 18. An illustration of self-infilling decoding with **interruption** and **looping** on HUMANEVAL, which successfully corrects the initially wrong solution at $N = 1$ (classified as the 'Incorrect $\rightarrow$ Correct' category in Figure 3). The area above the dashed line indicates the current context, while the content below represents the corresponding completion.

Problem Description

```
def sort_array(array):
    """
    Given an array of non-negative integers, return a copy of the given array after sorting,
    you will sort the given array in ascending order if the sum( first index value, last index value) is
odd,
    or sort it in descending order if the sum( first index value, last index value) is even.

    Note:
    * don't change the given array.

    Examples:
    * sort_array([]) => []
    * sort_array([5]) => [5]
    * sort_array([2, 4, 3, 0, 1, 5]) => [0, 1, 2, 3, 4, 5]
    * sort_array([2, 4, 3, 0, 1, 5, 6]) => [6, 5, 4, 3, 2, 1, 0]
    """
```

$N = 1$: Self-infilling

```
    if len(array) == 0:
        return array
    elif len(array) == 1:
        return array
    else:
       <SUF>return array


<MID> if sum(array[0], array[-1]) % 2 == 0:
            return sorted(array, reverse=True)
        else:
            <EOT>
```

$N = 1$: Left-to-right

```
    if len(array) == 0:
        return array
    elif len(array) == 1:
        return array
    else:
        if sum(array[0], array[-1]) % 2 == 0:
            return sorted(array, reverse=True)
        else:
```

```
            return sorted(array)
```

$N = 2$: Self-infilling

```
<SUF>    else:
        if sum(array[0], array[-1]) % 2 == 0:
            return sorted(array, reverse=True)
        else:
            return
```

```
                    sorted(array)

<MID>    if len(array) == 0:
        return []
    elif len(array) == 1:
        return array
<EOT>
```

$N = 2$: Left-to-right

```
    if len(array) == 0:
        return []
    elif len(array) == 1:
        return array
```

```
    else:
        if sum(array[0], array[-1]) % 2 == 0:
            return sorted(array, reverse=True)
        else:
            return sorted(array)
```

Figure 19. An illustration of self-infilling decoding with **interruption** and **looping** on HUMANEVAL, where the solution generated at $N = 1$ is incorrect and does *not* get fixed after the subsequent update (the 'Changed but Remained Incorrect' category in Figure 3). The area above the dashed line indicates the current context, while the content below represents the corresponding completion.

Problem Description

```
You are an expert Python programmer, and here is your task: Write a function to find the similar elements
from the given two tuple lists. Your code should pass these tests:

assert similar_elements((3, 4, 5, 6),(5, 7, 4, 10)) == (4, 5)
assert similar_elements((1, 2, 3, 4),(5, 4, 3, 7)) == (3, 4)
assert similar_elements((11, 12, 14, 13),(17, 15, 14, 13)) == (13, 14)
[BEGIN]
def similar_elements(test_tup1, test_tup2):
  res = tuple(set(test_tup1) & set(test_tup2))
  return (res)
[DONE]

... (2 other in-context examples)

You are an expert Python programmer, and here is your task: Write a function to extract the index minimum
value record from the given tuples. Your code should pass these tests:

assert index_minimum([('Rash', 143), ('Manjeet', 200), ('Varsha', 100)]) == 'Varsha'
assert index_minimum([('Yash', 185), ('Dawood', 125), ('Sanya', 175)]) == 'Dawood'
assert index_minimum([('Sai', 345), ('Salman', 145), ('Ayesha', 96)]) == 'Ayesha'
[BEGIN]
```

$N = 1$: Self-infilling

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

def index_minimum(<SUF>return min_value
[DONE]<MID>tuples):
    min_value = min(tuples, key=lambda x: x[1])[0]
    <EOT>
```

$N = 1$: Left-to-right

```
def index_minimum(tuples):
    min_value = min(tuples, key=lambda x: x[1])[0]

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    return min_value
[DONE]
```

$N = 2$: Self-infilling

```
<SUF>    min_value = min(tuples, key=lambda x:
x[1])[0]
    return

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

           min_value
[DONE]<MID>def index_minimum(tuples):
<EOT>
```

$N = 2$: Left-to-right

```
def index_minimum(tuples):

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    min_value = min(tuples, key=lambda x: x[1])
    return min_value[0]
[DONE]
```

Figure 20. An illustration of self-infilling decoding with **interruption** and **looping** on MBPP, where the solution generated at $N = 1$ remains correct after the subsequent update (the 'Changed but Remained Correct' category in Figure 3). The area above the dashed line indicates the current context, while the content below represents the corresponding completion.

Problem Description

```
You are an expert Python programmer, and here is your task: Write a function to find the similar elements
from the given two tuple lists. Your code should pass these tests:

assert similar_elements((3, 4, 5, 6),(5, 7, 4, 10)) == (4, 5)
assert similar_elements((1, 2, 3, 4),(5, 4, 3, 7)) == (3, 4)
assert similar_elements((11, 12, 14, 13),(17, 15, 14, 13)) == (13, 14)
[BEGIN]
def similar_elements(test_tup1, test_tup2):
  res = tuple(set(test_tup1) & set(test_tup2))
  return (res)
[DONE]

... (2 other in-context examples)

You are an expert Python programmer, and here is your task: Write a function to find the maximum difference
between the number of 0s and number of 1s in any sub-string of the given binary string. Your code should
pass these tests:

assert find_length("11000010001", 11) == 6
assert find_length("10111", 5) == 1
assert find_length("11011101100101", 14) == 2
[BEGIN]
```

$N = 1$: Self-infilling

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
def find_length(<SUF>return max_diff
[DONE]<MID>binary_string, n):
  max_diff = 0
  for i in range(n):
    zero_count = 0
    one_count = 0
    for j in range(i, n):
      if binary_string[j] == '0':
        zero_count += 1
      else:
        one_count += 1
      if abs(zero_count - one_count) > max_diff:
        max_diff = abs(zero_count - one_count)
  <EOT>
```

$N = 1$: Left-to-right

```
def find_length(binary_string, n):
  max_diff = 0
  for i in range(n):
    zero_count = 0
    one_count = 0
    for j in range(i, n):
      if binary_string[j] == '0':
        zero_count += 1
      else:
        one_count += 1
      if abs(zero_count - one_count) > max_diff:
        max_diff = abs(zero_count - one_count)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  return max_diff
[DONE]
```

$N = 2$: Self-infilling

```
<SUF>      if binary_string[j] == '0':
        zero_count += 1
      else:
        one_count += 1
      if abs(zero_count - one_count) > max_diff:
        max_diff = abs(zero_count - one_count)
  return
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
          max_diff
[DONE]<MID>def find_length(binary_string, n):
  max_diff = 0
  for i in range(n):
    zero_count = 0
    one_count = 0
    for j in range(i, n):
<EOT>
```

$N = 2$: Left-to-right

```
def find_length(binary_string, n):
  max_diff = 0
  for i in range(n):
    zero_count = 0
    one_count = 0
    for j in range(i, n):
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      if binary_string[j] == '0':
        zero_count += 1
      else:
        one_count += 1
      if abs(zero_count - one_count) > max_diff:
        max_diff = abs(zero_count - one_count)
  return max_diff
[DONE]
```

Figure 21. An illustration of self-infilling decoding with **interruption** and **looping** on MBPP, where the solution generated at $N = 1$ remains unchanged after the subsequent update (the 'Unchanged' category in Figure 3). The area above the dashed line indicates the current context, while the content below represents the corresponding completion.

Problem Description

```
You are an expert Python programmer, and here is your task: Write a function to find the similar elements
from the given two tuple lists. Your code should pass these tests:

assert similar_elements((3, 4, 5, 6),(5, 7, 4, 10)) == (4, 5)
assert similar_elements((1, 2, 3, 4),(5, 4, 3, 7)) == (3, 4)
assert similar_elements((11, 12, 14, 13),(17, 15, 14, 13)) == (13, 14)
[BEGIN]
def similar_elements(test_tup1, test_tup2):
  res = tuple(set(test_tup1) & set(test_tup2))
  return (res)
[DONE]

... (2 other in-context examples)

You are an expert Python programmer, and here is your task: Write a python function to check whether the
frequency of each digit is less than or equal to the digit itself. Your code should pass these tests:

assert validate(1234) == True
assert validate(51241) == False
assert validate(321) == True
[BEGIN]
```

$N = 1$: Self-infilling

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

def validate(num):
   <SUF>return True
    else:
        return False
[DONE]<MID> num = str(num)
    for i in num:
        if num.count(i) > int(i):
            <EOT>
```

$N = 1$: Left-to-right

```
def validate(num):
    num = str(num)
    for i in num:
        if num.count(i) > int(i):
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

            return False
    return True
[DONE]
```

$N = 2$: Self-infilling

```
<SUF>        if num.count(i) > int(i):
            return False
    return
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

           True
[DONE]<MID>def validate(num):
    for i in num:
<EOT>
```

$N = 2$: Left-to-right

```
def validate(num):
    for i in num:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

        if num.count(i) > int(i):
            return False
    return True
[DONE]
```

Figure 22. An illustration of self-infilling decoding with **interruption** and **looping** on MBPP, where the looped update introduces bugs to an initially correct solution at $N = 1$ (classified as the 'Correct $\rightarrow$ Incorrect' category in Figure 3). The area above the dashed line indicates the current context, while the content below represents the corresponding completion.