Predict, Don't Iterate: Efficient Adaptive-Length Infilling for Diffusion Language Models

Anonymous ACL submission

Abstract

Diffusion language models (DLMs) have emerged as a promising alternative to the autoregressive paradigm. With bidirectional attention and any-order generation, DLMs naturally fit infilling tasks, which require generating a middle span conditioned on both the prefix and the suffix. However, infilling is sensitive to the length of the span, while DLMs require the length to be fixed before generation. Although prior studies extend DLMs to dynamic lengths, they still suffer from two limitations. (i) *Sensitivity to initial length*. These methods require a preset length to initialize the search and are highly sensitive to this initial length, often yielding suboptimal results. (ii) *Inference inefficiency*. They either insert length-changing operations during generation or repeatedly search for an appropriate length using multi-step denoising confidence, both of which introduce substantial extra forward passes and computational cost. Therefore, we propose PILL (**P**robing-based **I**nfi**L**ling with preset-**L**ength-free decoding), an efficient infilling method for DLMs that requires no preset initial length and adds far fewer extra forward passes than baselines, substantially reducing inference time. Experiments show that, across five DLMs spanning different families, architectures, and training recipes on eight infilling benchmarks, PILL improves over the strongest baseline by $+4.8$ average pass rate on code and $+6.0$ BLEU-2 on text, while running $1.82\times$ faster than that baseline and adding less than 10% time over fixed-length decoding.

1 Introduction

Recently, diffusion language models (DLMs) have emerged as a promising alternative to autoregressive (AR) paradigm, showing great potential across various tasks (Nie et al., 2026; Zhu et al., 2025b,a; Ye et al., 2025). Unlike AR models that rely on causal attention and decode strictly left to right, DLMs employ bidirectional attention and generate in any order, allowing them to condition on both the left and right context simultaneously (Hoogeboom et al., 2021). This property makes DLMs particularly well suited to infilling, generating a middle span that is coherent with both a given prefix and a given suffix, a setting that AR models are hard to handle (Bavarian et al., 2022).

However, DLMs require the number of masked positions to be fixed before decoding, whereas the correct infilling length is unknown in advance and varies from case to case (Bie et al., 2025). Crucially, infilling quality is highly sensitive to the chosen length: as shown in Figure 1(a), the code-infilling example is solved correctly only when the mask span is given exactly the right number of tokens, while slightly shorter or longer spans produce incorrect completions. The model must therefore determine a length before it knows what to generate, even though the appropriate length depends on the content itself. Recent studies tackle this through adaptive-length decoding (Li et al., 2025a; Wu et al., 2026; Kim et al., 2025; Liu et al., 2026), but they still suffer from two limitations.

First, existing methods are ***sensitive to the initial length***. They require a preset length to start the search for a better infilling length. For example, CAL (Liu et al., 2026) will be trapped into local minima near the initial length, yielding varying performances, as shown in Fig. 1(b). Second, they incur ***substantial inference overhead***. Methods either inject explicit one token length-changing operations (Wu et al., 2026; Li et al., 2025a), or repeatedly probe denoising confidence over many decoding steps to estimate a length (Liu et al., 2026), introducing many extra forward passes and cost.

To this end, we propose PILL, a framework that enables fixed-length DLMs to perform variable-length infilling with only two extra forward passes. PILL consists of three stages. (1) Instead of searching for a length, we *directly predict* it from the bidirectional hidden state of a single mask token

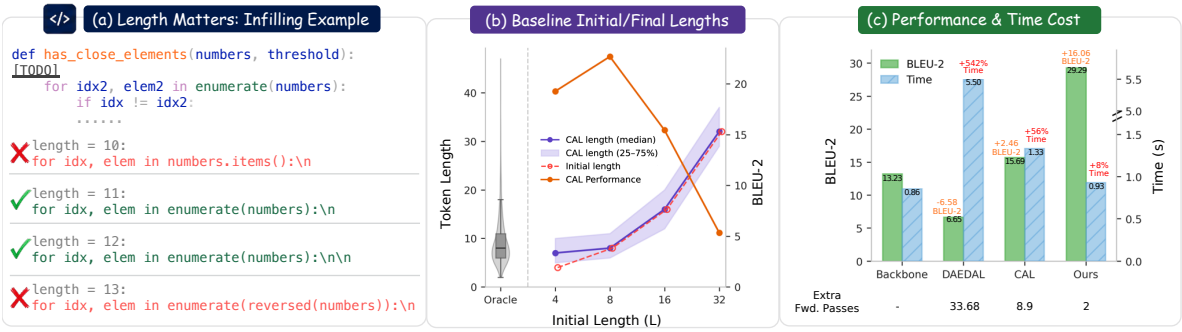


Figure 1: **Motivation and Performance of PILL.** (a) For DLM infilling tasks, correctness is sensitive to the span lengths. (b) Baselines must preset an initial length L and their final length stays tied to it, whereas PILL requires no preset length. (c) PILL attains the best performance with minimal overhead compared to fixed-length generation with backbones. Baselines are reported at initial length $L = \{4, 8, 16, 32\}$.

inserted between the prefix and suffix. (2) Since generation is sensitive to length, we expand this estimate into a small set of candidate lengths and decode them *in parallel* at a cost close to decoding one. (3) Since DLMs cannot score a generated sequence by next-token likelihood, we *select* the final span with a post-hoc score that favors candidates coherent both internally and with the suffix, using one additional forward pass. In summary, it is more efficient than baselines: it *replaces their iterative length search*, which spends a forward pass per step, with just two extra passes, while the multi-slot decoding generates all candidates in parallel at almost no extra cost. As shown in Figure 1(c), it achieves the highest quality while adding $< 10\%$ wall-clock time over vanilla fixed-length decoding, whereas baselines pay far more for a smaller gain. Our contributions are three-fold:

- **A preset-free infilling framework**. We propose PILL, a backbone-frozen framework that enables DLMs to perform adaptive-length infilling beyond the fixed lengths with only two extra forward passes, centered on a length probe that predicts target lengths directly from a mask token's bidirectional hidden state.
- **Parallel decoding and selection.** We decode a set of length candidates in parallel at near single-candidate cost via a slot-wise attention mask, and select the final span with a post-hoc coherence score in one pass.
- **Extensive validation.** Across five DLMs and eight code and text benchmarks, PILL surpasses the strongest baselines by $+4.8$ pass rate on code and $+6.0$ BLEU-2 on text while adding only $< 10\%$ time compared to vanilla fixed-length decoding.

2 Related Work

Diffusion Language Models. DLMs generate text by iteratively denoising masked or corrupted sequences. The idea originates from early works such as D3PM (Austin et al., 2021a) and Diffusion-LM (Li et al., 2022a), and is further developed by Masked Diffusion Language Models (MDLMs), which improve the modeling objective, sampling, and scalability (Shi et al., 2024; Sahoo et al., 2024). Recent models scale DLMs to larger parameters, including research previews such as Gemini-Diffusion and Seed Diffusion (Song et al., 2025) and open-sourced models such as LLaDA (Nie et al., 2026) and Dream (Ye et al., 2025), reaching competitive performance across tasks including code generation, instruction following, and reasoning (Xie et al., 2025; Bie et al., 2025). Owing to their bidirectional attention, DLMs can condition on both left and right context, which makes them naturally suited to infilling (Li et al., 2025b).

Infilling and Adaptive-Length Decoding. Infilling has been extensively studied in the AR paradigm, including FIM-style training (Bavarian et al., 2022) and code-specialized models (Fried et al., 2022; Li et al., 2023; Roziere et al., 2023), all of which decode the missing span left-to-right via causal attention. While effective, AR models can only access the suffix indirectly through prompt conditioning, motivating bidirectional alternatives such as DLMs that natively condition on both prefix and suffix (Ye et al., 2025). However, DLMs typically require the number of masked positions to be specified before decoding, restricting them to fixed-length generation (Lin et al., 2026). Recent studies address this by adaptive length decoding. DAEDAL (Li et al., 2025a) dynamically expands the generation canvas during inference, but only focuses on appending at the end of se-

quences. DreamOn (Wu et al., 2026) introduces explicit length-changing operations into the diffusion process, but requires fine-tuning the DLM, and potentially compromises general-purpose capabilities. FlexMDM (Kim et al., 2025) and DDOT (Zhang et al., 2025) also introduces additional fine-tuning cost. CAL (Liu et al., 2026) searches for suitable lengths using early-step denoising confidence, but incurs extra forward passes and is highly sensitive to the initial length. In contrast, our approach uses a lightweight probe to directly predict the target length, and a multi-slot decoding scheme that generates and selects among multiple length candidates in parallel. This yields adaptive-length infilling with minimal additional cost and no reliance on a preset initial length.

3 Preliminaries

Diffusion Language Models. Let $x = (x_1, \ldots, x_M)$ be a token sequence over vocabulary $\mathcal{V}$, augmented with a mask token $[\mathtt{M}]$. DLMs define a forward process that independently replaces each x_i with $[\mathtt{M}]$ with probability $t \in [0, 1]$, yielding a corrupted sequence $x^{(t)}$. A network $p_\theta(x^{(0)} \mid x^{(t)})$ is trained to reconstruct the original tokens at masked positions (Sahoo et al., 2024; Shi et al., 2024). Unlike auto-regressive models, DLMs use bidirectional attention, so every prediction conditions on all other positions on both sides. At inference, given a prefix P and a target length L, decoding starts from $x^{(K)} = P \oplus [\mathtt{M}]^L$, where $\oplus$ denotes concatenation, and proceeds for K steps. At each step $k = K, \ldots, 1$, the model produces a distribution over $\mathcal{V}$ at every masked position; a subset of positions with the highest confidence are unmasked, while the rest remain masked for step $k-1$. The final $x^{(0)}$ contains no masks. A central limitation is that L must be specified *before* decoding and remains fixed throughout.

Infilling Tasks for DLMs. The inference setting above fills a masked region given only a prefix P. Infilling extends this setting by additionally fixing a suffix S as a right-side condition, which DLMs can exploit directly thanks to their bidirectional attention and any-order generation. Given a prefix P and a suffix S, the infilling task asks for a middle span $M = (w_1, \ldots, w_{L^*})$ with $w_i \in \mathcal{V}$, such that the concatenation $P \oplus M \oplus S$ maximizes a task-specific quality metric $\mathcal{Q}$ (e.g., pass rate for code, BLEU-2 for text). The oracle length L^* is unknown at inference time, and $\mathcal{Q}$ is inaccessible during decoding. DLMs instantiate the masked region as $P \oplus [\mathtt{M}]^{L_0} \oplus S$ at a preset length L_0 and run iterative masked decoding to fill it.

4 Method

4.1 Overview

We cast the infilling task as two coupled subproblems: (1) determining the length of the mask span and (2) generating content coherent with both the prefix and the suffix. The two subproblems are entangled, since the appropriate length depends on what to generate, yet DLMs can only generate with fixed lengths. PILL decouples them through a three-stage design shown in Fig. 2. (1) Stage I predicts a target length $\hat{L}$ directly from hidden states of mask token through a single forward pass with a lightweight MLP probe, avoiding iterative length search. Since the generation is sensitive to lengths, we expand $\hat{L}$ into a small candidate set $\{\hat{L}-r, \cdots, \hat{L}+r\}$ to enhance predictive reliability. (2) Stage II generates all candidate lengths in parallel at nearly the cost of decoding one, where each candidate length occupies a separate decoding region (a slot) and a slot-wise attention mask isolates the slots so they share prefix/suffix context but do not leak into one another, accelerating generation. (3) Stage III selects the final span from candidates via a post-hoc score that jointly measures each candidate's internal coherence and how naturally the suffix continues from it in a single additional forward pass, equipped with a well-designed attention mask mechanism. Throughout the pipeline, we only introduce two passes, the probing pass and the scoring pass. Also, by predicting the length directly rather than searching from a preset value, PILL requires no initial lengths, yielding consistent performance across datasets.

4.2 Length Probing

The first stage estimates the length of the missing span directly from hidden states, as shown in Fig. 2(a). Given a prefix P and suffix S, we construct a *probing sequence* by inserting a single mask token between them, denoted as $x_{\text{probe}} = P \oplus [\mathtt{M}] \oplus S$, and run a single forward pass of the frozen DLM backbone. We read the hidden state $h \in \mathbb{R}^d$ of the mask position at a certain layer. Because the backbone uses bidirectional attention, h aggregates information from both prefix and suffix, making it a natural summary of how much content the missing span is expected to hold. Also, it

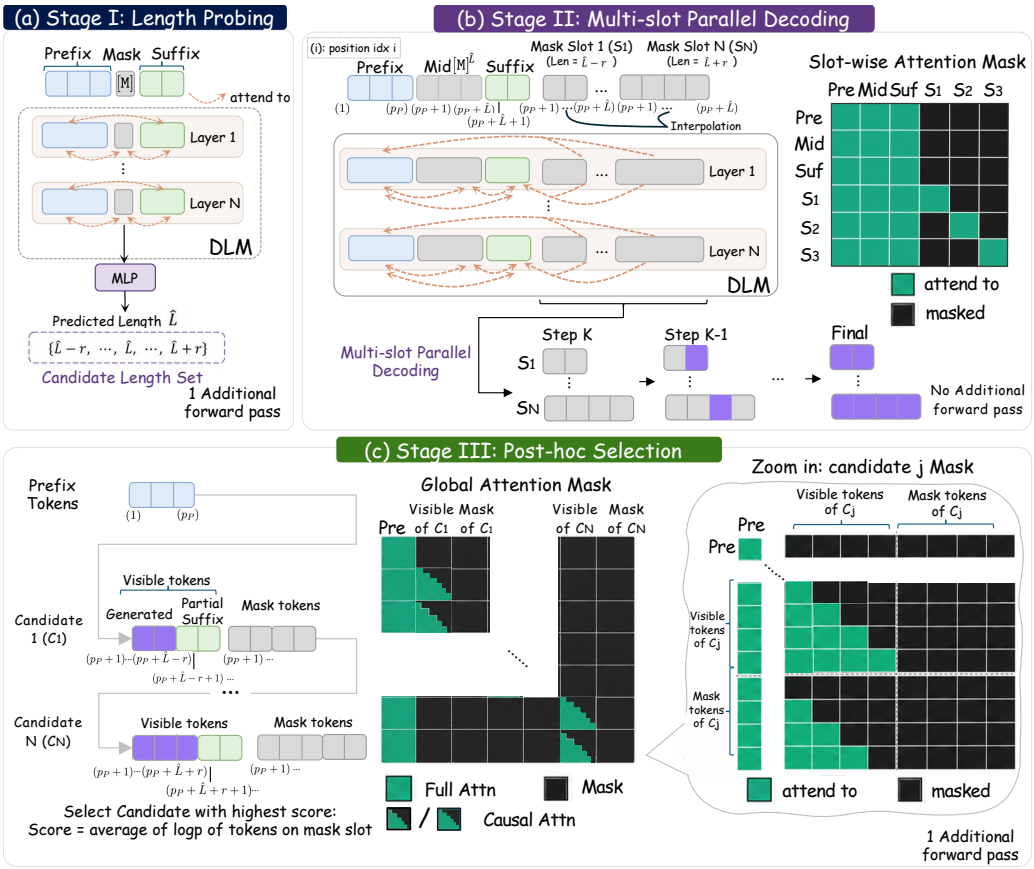


Figure 2: **Pipeline of the proposed method.** (a) Stage I probes the hidden state of a mask token to predict the target length $\hat{L}$; (b) Stage II decodes the candidate lengths $\{\hat{L}-r, \ldots, \hat{L}+r\}$ in parallel with a slot-wise attention mask; (c) Stage III selects the final span with post-hoc score in one forward pass via a well-designed attention mask.

is a rich representation rather than a single confidence score that DAEDAL (Li et al., 2025a) and CAL (Liu et al., 2026) rely on, providing more information to predict. Then, a lightweight 3-layer MLP probe $f_\phi(\cdot)$ maps h to a scalar length estimate

$$\hat{L} = \lfloor \exp\left(f_\phi(h)\right) \rceil, \tag{1}$$

where $\lfloor \cdot \rceil$ rounds to the nearest positive integer. We predict $\log L$ rather than L directly, which guarantees a positive estimate and reduces the influence of long-tailed length distribution. More information about the probe can be found in Appendix A.7. The lightweight probe does not touch the backbone, preserving the DLM ability without finetuning the backbone. Also, we only introduce one single forward pass in this stage to avoid presetting an initial length, with minimal overhead. We also validate the probing by comparing predicted lengths against the oracle, indicating that our probe predicts lengths more accurately than the baseline while requiring no preset initial length (Appendix A.1).

4.3 Multi-slot Parallel Decoding

Since the DLM generation is highly sensitive to the length of the mask slot, we expand the length $\hat{L}$ obtained by the probe within a local radius r into $\{\hat{L}-r, \cdots, \hat{L}+r\}$ to better reduce prediction errors, yielding $N = 2r+1$ candidates, and assign the j-th candidate length ℓ_j to a separate decoding slot, which is a contiguous block of mask tokens that is decoded independently. But it yields much more decoding cost, therefore we propose a *multi-slot parallel decoding* strategy at a cost close to decoding one slot alone (Fig. 2(b)).

Sequence Layout. Since all the candidates share the prefix and suffix, we pack the candidate slots together with a shared context region, $P \oplus [\mathtt{M}]^{\hat{L}} \oplus S \oplus [S_1] \cdots [S_N]$, where each slot S_j is initialized with ℓ_j mask tokens. The prefix, middle mask slot and suffix form a shared context, and they attend to each other and are attended by every slot S_j. In addition, to avoid interference between mask slots S_j, we isolate the slots through a slot-wise attention mask. Each slot S_j attends only to its own tokens and the shared context $\{P, [\mathtt{M}]^{\hat{L}}, S\}$, but not to any other mask slot $S_{j'} (j' \neq j)$. Because each slot sees nearly the same context as in single-slot generation, its denoising behavior, and thus its output, closely

matches that of decoding the slot alone. This holds only when each slot is given appropriate position ids, which we address below.

Position IDs. A subtlety is how to assign position ids to slots of different lengths. The key principle is that a DLM places each predicted token at the position dictated by its position id. If the mask span is given more position ids than the content needs, the model treats the extra positions as space still to be filled and under-generates. We illustrate this with a concrete example, in which `[TODO]` denotes the span to be filled.

```
I have 13 books, and you have 23 books.
(1)
We have [TODO] books.
    (17)
```

The numbers below the text are the position ids assigned by LLaDA. The intended answer is `"36"` (two tokens). When the gap is given exactly two position ids, the model correctly produces `"36"`. But when we keep a single mask token yet leave a one-position gap between it and one side of the context, the model fills only half the answer: leaving the gap on the suffix side makes the mask take position id `"(18)"` while the suffix resumes at `"(20)"`, and the model outputs only `"3"` (the first token of `"36"`); leaving the gap on the prefix side instead yields `"6"` (the second token). In each case the model anchors its output to the position ids and leaves the gap unfilled, i.e., it under-generates. It motivates a no-gap mask positional id design. The mask slot should start at the position id immediately after the prefix and end immediately before the suffix, leaving no gaps from them.

We therefore fix the suffix anchor and *interpolate* each slot's position ids over a shared gap interval, so that all slots present an identical layout to the suffix regardless of their length. Let the last prefix token have position id p_P. We fix the suffix to start at $p_S = p_P + \hat{L} + 1$, leaving a gap interval $[\,p_P + 1,\ p_S - 1\,]$ of width $\hat{L}$. For a slot S_j of length ℓ_j, its k-th token ($k = 1, \ldots, \ell_j$) is assigned

$$\mathrm{pos}(S_j, k) = (p_P + 1) + (k - 1) \cdot \frac{\hat{L} - 1}{\ell_j - 1}, \quad (2)$$

linearly interpolating the ℓ_j tokens between $p_P + 1$ and $p_S - 1$. Every slot thus spans the same interval and the suffix is anchored at p_S for all candidates, eliminating the length-dependent misalignment above. This design is critical: padding-based alternatives that left- or right-align the slot leave a position-id gap between the candidate and one side of the context, which leads to under-generation by the mechanism above. In contrast, interpolation removes this gap and substantially outperforms both padding variants (Appendix A.2).

Parallel Decoding. Within each slot, we run the original denoising/unmasking process. Since the mask slots do not interfere with each other and can be denoised in parallel, we obtain N infill candidates $\{C_1, \cdots, C_N\}$ in almost the same time as decoding a single slot. This also saves KV-cache computation, since the shared context $\{P, [\mathtt{M}]^{\hat{L}}, S\}$ remains unchanged throughout decoding. The middle $[\mathtt{M}]^{\hat{L}}$ produces no output, and it merely anchors the shared position-id interval, and its KV is identical across all slots and steps. This motivates placing the shared context first in the sequence layout $(P \oplus [\mathtt{M}]^{\hat{L}} \oplus S \oplus [S_1] \cdots [S_N])$, so that its *KV can be cached and reused* across decoding steps. Please see Fig. 2(c) for an illustration.

4.4 Post-hoc Selection

Stage II returns N infill candidates $\{C_1, \ldots, C_N\}$ of different lengths, and we must commit to one. AR models can score a sequence by its left-to-right next-token likelihood. In contrast, DLMs are trained to predict the token *at* a masked position rather than the *next* token, so the distribution at an already-committed token is not a predictive distribution and cannot serve as a score (Fig. 3). Our goal is to select the candidate that makes the

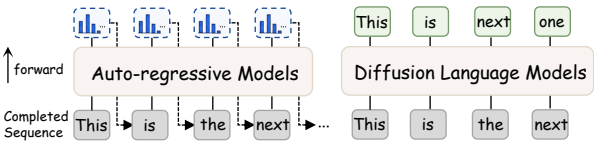


Figure 3: Forward-pass targets in ARMs and DLMs.

completed sequence $P \oplus C_j \oplus S$ most coherent as a whole, rather than one that looks good in isolation. A natural score is the pseudo-log-likelihood, which masks each token in turn and reads its log-likelihood from the rest; however, this requires one forward pass per token and is prohibitively expensive. Instead, we fix the prefix as a shared condition across all candidates and score the inner part for internal coherence and the suffix for how naturally it continues from the candidate, all in a single forward pass.

Masked Scoring Probes. For candidate C_j we lay out a visible block $\boldsymbol{v} = C_j \oplus \tilde{S} = (v_1, \ldots, v_{\ell_j + m})$, the ℓ_j committed candidate tokens followed by the m selected suffix tokens $\tilde{S}$-a small, representative

subset of the suffix used to judge how well it continues from the candidate (introduced later)- and a block of *scoring probes*, $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_{\ell_j+m})$, where each μ_k is a mask token and shares the position id of v_k. We then carefully design an attention mask (Fig. 2(c)), enforcing a strictly causal read. The prefix P is visible to the entire block. (1) Each *visible* token v_k attends to P and to $v_{\leq k}$ (inclusive), so the visible block stays internally causal and no visible representation contains information of a later token. (2) Each *probe* μ_k attends to P and to $v_{<k}$ only (strict), never to v_k itself, to any later position, or to another probe, to prevent it from accessing $v_{\geq k}$. Consequently the output at μ_k is $p_\theta(\cdot \mid P, v_{<k})$, and reading it at the observed token gives $\log p_\theta(v_k \mid P, v_{<k})$ with no label leakage; the first probe conditions on P alone. The inclusive-causal visible block is essential: were it bidirectional, $v_{<k}$ would already encode v_k and leak the token the probe is meant to predict. Since the suffix tokens follow the full candidate in v, their probes automatically condition on all of C_j, exactly the quantity needed to judge continuation.

Selection Score. Let c_k and $\tilde{s}_k$ be the k-th token in C_j and $\tilde{S}_j$. Partitioning the per-position log-probs of v into its candidate and suffix segments gives an inner score and a suffix score,

$$s_j^{\text{in}} = \frac{1}{\ell_j} \sum_{k=1}^{\ell_j} \log p_\theta(c_k \mid P, c_{<k}) \,,$$

$$s_j^{\text{suf}} = \frac{1}{m} \sum_{t=1}^{m} \log p_\theta(\tilde{s}_t \mid P, C_j, s_{<t}) \,,$$

and we select the candidate maximizing their convex combination,

$$j^\star = \arg\max_j \; \alpha \, s_j^{\text{in}} + (1-\alpha) \, s_j^{\text{suf}}. \qquad (3)$$

The inner score favors spans the model finds internally coherent, while the suffix score favors spans after which the observed suffix becomes likely, penalizing over- and under-generated spans that read fluently on their own but break the transition into S. The weight α balances local fluency against suffix alignment. We choose $\alpha = 0.5$ in practice.

Sparse Suffix Selection. Scoring the entire suffix is unnecessary, as the continuation signal concentrates in a few tokens. We build $\tilde{S}$ from (i) suffix head: the first few suffix tokens, which carry the immediate transition, and (ii) important tokens: the suffix tokens that received the highest attention from the infilling region in Stage II, which capture longer-range dependence (e.g., some variables appear later in the code). These two groups need not be contiguous: the intervening suffix tokens are not scored, but are kept as visible context so that each scored token still conditions on the full true suffix preceding it, $\log p_\theta(\tilde{s}_t \mid P, C_j, s_{<t})$, with an attention mask. We cap the considered suffix length at 32 and use $m \in [4, 8]$ (the two groups may overlap) scored tokens in practice. For clarity, Fig. 2(c) depicts only the suffix head; the salient tokens follow the same scoring structure and are omitted to save space.

Single-pass Scoring. Finally, the N candidate blocks are packed into one sequence and isolated by a block-wise attention mask (Fig. 2(c), Global Attention Mask): the visible tokens and probes of candidate j attend only to the shared prefix and to C_j's own block, never across candidates, so the blocks cannot interfere with each other. All candidates are therefore scored together in a single additional forward pass.

5 Experiments

5.1 Experimental Setup

Models. We apply PILL to five diffusion large language models (DLMs): LLaDA-8B-Base/Instruct (Nie et al., 2026), LLaDA-MoE-Base (Zhu et al., 2025b), Dream-7B-Base (Ye et al., 2025), and Dream-Coder-7B-Base (Xie et al., 2025). These models cover both general-purpose and code-oriented DLMs, spanning different model families (LLaDA/Dream), architectures (dense/MoE), and training recipes (base/instruction-tuned).

Datasets. We evaluate on code and natural-language infilling benchmarks, all disjoint from probe training set. For code, we use HumanEval-Infilling (Bavarian et al., 2022) with single-line (-S) and multi-line (-M) tasks, construct MBPP-S/-M from MBPP (Austin et al., 2021b) by a similar protocol, and build single-line Java and C/C++ tasks from MultiPL-E (Cassano et al., 2022). For text, we construct random contiguous word-span tasks from WikiText (Merity et al., 2016) and arXiv abstracts. We report pass rate for code and BLEU-2/ROUGE-L for text. See Appendix A.8 for details.

Baselines. We compare PILL with the backbone models (fixed-length decoding) and two dynamic decoding methods, DAEDAL (Li et al., 2025a) and CAL (Liu et al., 2026). Since all baselines require an initial generation length, we follow (Wu et al., 2026; Liu et al., 2026) and report the average performance over $L \in \{4, 8, 16, 32\}$. This provides

Datasets	Python				Other Language		Text	
	Human-S	Human-M	MBPP-S	MBPP-M	Java	C/C++	Wiki	Arxiv
LLaDA-8B-Base								
Backbone	44.68	15.92	37.45	20.45	34.09	34.84	19.90 / 34.48	13.23 / 30.30
DAEDAL	44.99	16.76	38.86	22.48	35.34	36.03	10.82 / 20.78	6.65 / 16.61
CAL	64.74	24.61	54.49	32.21	50.31	49.22	23.00 / 37.25	15.69 / 32.69
PILL	**71.35**	**28.20**	**57.49**	**38.19**	**57.10**	**54.57**	**29.29 / 44.96**	**20.28 / 39.82**
PILL (Oracle)	81.90	38.54	71.56	50.92	60.98	60.08	41.51 / 56.76	29.35 / 50.43
LLaDA-8B-Instruct								
Backbone	49.73	17.62	37.47	21.15	32.42	33.40	18.07 / 32.77	12.57 / 29.44
DAEDAL	50.20	18.68	39.17	23.02	34.31	34.81	10.37 / 19.98	6.56 / 16.64
CAL	**69.77**	26.93	48.64	25.31	49.20	48.66	20.52 / 34.93	14.77 / 31.33
PILL	67.86	**27.38**	**52.27**	**36.86**	**52.66**	**52.35**	**25.58 / 41.91**	**18.94 / 38.13**
PILL (Oracle)	78.99	41.03	72.38	56.67	61.53	56.95	38.07 / 54.54	28.46 / 49.60
LLaDA-MoE-Base								
Backbone	45.84	16.43	40.42	22.56	34.53	34.92	17.23 / 31.51	13.04 / 29.69
DAEDAL	46.17	17.56	41.50	24.77	37.25	37.14	8.90 / 18.48	6.06 / 15.70
CAL	67.79	25.96	58.32	34.45	53.06	50.41	19.39 / 33.75	14.78 / 31.25
PILL	**71.44**	**31.92**	**58.52**	**43.12**	**55.76**	**55.23**	**25.76 / 41.84**	**19.13 / 38.30**
PILL (Oracle)	81.80	41.34	67.56	54.62	58.98	58.77	37.51 / 53.41	28.03 / 49.12
Dream-7B-Base								
Backbone	42.79	16.13	40.12	22.15	33.90	34.30	19.12 / 33.75	13.56 / 30.97
DAEDAL	42.62	17.60	41.22	25.11	38.20	37.88	9.68 / 19.33	6.72 / 17.22
CAL	64.48	26.38	56.00	33.60	47.92	48.00	21.68 / 36.68	15.98 / 33.52
PILL	**68.34**	**28.55**	**57.97**	**37.72**	**58.98**	**56.30**	**32.36 / 48.13**	**22.89 / 42.79**
PILL (Oracle)	79.77	39.19	70.91	52.11	63.64	62.63	44.66 / 59.98	33.64 / 54.49
Dream-Coder-7B-Base								
Backbone	49.64	19.00	43.94	25.62	36.89	37.66	-	-
DAEDAL	50.85	21.14	47.79	30.80	40.96	40.72	-	-
CAL	68.56	28.24	61.35	38.91	51.14	49.28	-	-
PILL	**71.35**	**32.14**	**62.59**	**47.28**	**59.87**	**59.01**	-	-
PILL (Oracle)	82.38	44.01	73.59	63.41	62.20	62.88	-	-

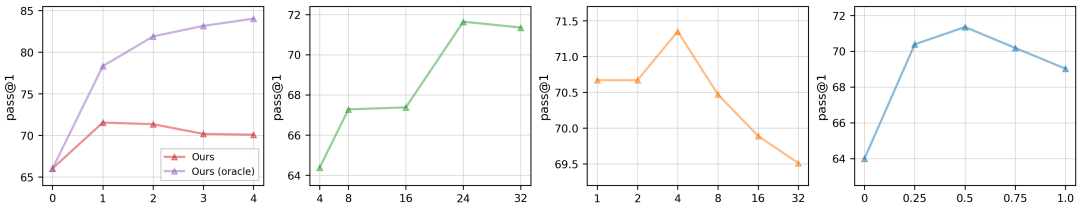
Table 1: Performance Comparison. We compare PILL with baselines on eight benchmarks across five models.

a prior-free evaluation protocol, and per-length results are provided in Appendix. CAL is evaluated using its official implementation and hyperparameters. We adapt DAEDAL following the implementation in (Liu et al., 2026) to fit the infilling setting. We also report comparison with DreamOn (Wu et al., 2026) on text tasks. All experiments run on a single NVIDIA A40 GPU.

5.2 Main Results

Table 1 reports infilling quality across five DLMs and eight code and natural-language benchmarks. PILL delivers the strongest results on nearly every model-dataset combination, improving the most competitive baseline CAL by $+4.8$ pass rate on average across the code benchmarks and by $+6.0$ BLEU-2 points on text. These gains hold across model families (LLaDA/Dream), architectures (dense/MoE), training recipes (base/instruction-tuned), and both code and text, indicating that PILL fits different models and tasks.

The baselines further illustrate why adaptive-length infilling is needed. DAEDAL, which only appends at the end of the canvas, cannot respect the suffix constraint and degrades even below the fixed-length backbone on text. CAL is the strongest baseline but still trails PILL while incurring repeated confidence-search passes and remaining sensitive to the preset initial length (Fig. 1(b)), whereas PILL needs no preset length at all. Finally, the oracle upper bound PILL (Oracle) shows that the candidate set almost always contains a strong completion, confirming that the multi-slot stage adequately covers the correct length. The remaining gap to the



(a) Radius (b) Probe Input Layer (c) Selection Suffix Length (d) Selection Alpha

Figure 4: Ablation studies on four design choices of PILL.

oracle indicates that the principal headroom lies in post-hoc selection rather than generation.

5.3 Efficiency Analysis

Beyond the WikiText results in Fig. 1(c), Table 2 compares quality and cost on HumanEval-S with LLaDA-8B-Base. PILL attains the highest Pass@1 while adding only 7% wall-clock time over the fixed-length backbone, in contrast to CAL (+94%) and DAEDAL (+117%). Equivalently, PILL is 1.82× faster than the strongest baseline CAL yet 6.6 points more accurate. One of the reasons for this efficiency is that PILL introduces only two extra forward passes, the probing pass and the scoring pass, whereas the iterative length search in CAL and DAEDAL requires much more.

Method	Pass@1	Extra Fwd. Passes	Time Cost
Backbone	44.67	-	1.10 s (+0%)
DAEDAL	44.80	11.98	2.29 s (+117%)
CAL	64.74	13.79	2.13 s (+94%)
PILL	**71.35**	2.00	1.18 s (+7%)

Table 2: Efficiency of LLaDA-8B-Base on Human-S.

5.4 Ablation Study

Length probing and candidate set. Fig. 4(b) varies the layer from which the probe reads the mask hidden state: Pass@1 increases monotonically with depth, consistent with deeper states aggregating richer bidirectional context from both prefix and suffix. We therefore read h from the last layer for convenience. Fig. 4(a) varies the candidate radius r. The oracle upper bound keeps rising with r, confirming that a larger candidate set is more likely to contain the correct length. PILL, however, saturates beyond $r = 2$, since candidate set is larger and the ratio of correct ones decreases. Also, we validate the effectiveness of length probing, as shown in Appendix A.1.

Post-hoc selection. Fig. 4(c) varies the number of scored suffix tokens m. Pass@1 peaks at $m = 4$ and declines thereafter, supporting our suffix selection: the continuation signal concentrates in a few tokens near the transition, and scoring more of them only injects noise. Fig. 4(d) varies the weight α between the inner and suffix scores. Performance peaks around $\alpha = 0.5$ and drops at both extremes, showing that internal coherence and suffix continuation are complementary and neither alone suffices.

5.5 Comparison with DreamOn

DreamOn (Wu et al., 2026) attains variable-length infilling by fine-tuning the DLM with explicit length-changing operations, whereas PILL requires no backbone-finetuning and leaves the backbone untouched. We compare on the text infilling tasks, which are out-of-distribution for both methods (Table 3): DreamOn is trained on code, and for this comparison, our probe is also fitted only on code. Despite this disadvantageous setting, PILL outperforms DreamOn across all of its length configurations (initial length $l \in \{1, 4, 8\}$) on both datasets, indicating that our length prediction generalizes more robustly across domains than a fine-tuned length-changing mechanism. It also implies that DreamOn is sensitive to initial lengths, while PILL requires no preset length.

Dataset	Wiki		Arxiv	
	BLEU-2	ROUGE-L	BLEU-2	ROUGE-L
DreamOn (l=1)	18.72	33.43	13.76	30.25
DreamOn (l=4)	21.33	36.09	15.62	33.17
DreamOn (l=8)	17.39	31.34	14.50	31.57
PILL	**23.67**	**38.89**	**16.11**	**34.43**

Table 3: Comparison with DreamOn (l: initial length).

6 Conclusion

We presented PILL, a backbone-frozen framework that turns fixed-length DLMs into adaptive-length infillers through three lightweight stages: probing the mask hidden state to predict the target length, decoding multiple length candidates in parallel via slot-wise attention, and selecting among them with a post-hoc coherence score. Adding only two forward passes and requiring no preset length, PILL outperforms prior adaptive-length baselines across five DLMs on both code and text infilling, running substantially faster than them.

Limitations

While PILL is effective and efficient across a range of DLMs and infilling benchmarks, our evaluation focuses on single-span infilling. Many practical scenarios involve multiple disjoint gaps or nested infilling, and extending the length probe and the multi-slot decoding scheme to jointly handle several spans is left to future work. In addition, our experiments are conducted on DLMs up to roughly 7B/8B parameters on a single GPU. Whether the behavior of the length probe and the post-hoc selection scales to substantially larger diffusion backbones remains to be verified.

References

Jacob Austin, Daniel D Johnson, Jonathan Ho, Daniel Tarlow, and Rianne Van Den Berg. 2021a. Structured denoising diffusion models in discrete state-spaces. *Advances in neural information processing systems*, 34:17981–17993.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021b. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

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

Tiwei Bie, Maosong Cao, Kun Chen, Lun Du, Mingliang Gong, Zhuochen Gong, Yanmei Gu, Jiaqi Hu, Zenan Huang, Zhenzhong Lan, and 1 others. 2025. Llada2. 0: Scaling up diffusion language models to 100b. *arXiv preprint arXiv:2512.15745*.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, and 1 others. 2022. Multiple: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227*.

Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. 2023. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

Emiel Hoogeboom, Didrik Nielsen, Priyank Jaini, Patrick Forré, and Max Welling. 2021. Argmax flows and multinomial diffusion: Learning categorical distributions. *Advances in neural information processing systems*, 34:12454–12465.

Jaeyeon Kim, Lee Cheuk-Kit, Carles Domingo-Enrich, Yilun Du, Sham Kakade, Timothy Ngotiaoco, Sitan Chen, and Michael Albergo. 2025. Any-order flexible length masked diffusion. *arXiv preprint arXiv:2509.01025*.

Jinsong Li, Xiaoyi Dong, Yuhang Zang, Yuhang Cao, Jiaqi Wang, and Dahua Lin. 2025a. Beyond fixed: Training-free variable-length denoising for diffusion large language models. *arXiv preprint arXiv:2508.00819*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Tianyi Li, Mingda Chen, Bowei Guo, and Zhiqiang Shen. 2025b. A survey on diffusion language models. *arXiv preprint arXiv:2508.10875*.

Xiang Li, John Thickstun, Ishaan Gulrajani, Percy S Liang, and Tatsunori B Hashimoto. 2022a. Diffusion-lm improves controllable text generation. *Advances in neural information processing systems*, 35:4328–4343.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022b. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Haokun Lin, Xinle Jia, Shaozhen Liu, Shujun Xia, Weitao Huang, Haobo Xu, Junyang Li, Yicheng Xiao, Xingrun Xing, Ziyu Guo, and 1 others. 2026. Efficient diffusion language models: A comprehensive survey. *Authorea Preprints*, 3.

Hengchang Liu, Zhao Yang, and Bing Su. 2026. Diffusion lms can approximate optimal infilling lengths implicitly. *arXiv preprint arXiv:2602.00476*.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*.

Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. 2026. Large language diffusion models. *Advances in Neural Information Processing Systems*, 38:50608–50646.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67.

Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices*, 51(10):731–747.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Subham S Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T Chiu, Alexander Rush, and Volodymyr Kuleshov. 2024. Simple and effective masked diffusion language models. *Advances in Neural Information Processing Systems*, 37:130136–130184.

Jiaxin Shi, Kehang Han, Zhe Wang, Arnaud Doucet, and Michalis Titsias. 2024. Simplified and generalized masked diffusion for discrete data. *Advances in neural information processing systems*, 37:103131–103167.

Yuxuan Song, Zheng Zhang, Cheng Luo, Pengyang Gao, Fan Xia, Hao Luo, Zheng Li, Yuehang Yang, Hongli Yu, Xingwei Qu, and 1 others. 2025. Seed diffusion: A large-scale diffusion language model with high-speed inference. *arXiv preprint arXiv:2508.02193*.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063.

Zirui Wu, Lin Zheng, Zhihui Xie, Jiacheng Ye, Jiahui Gao, Shansan Gong, Yansong Feng, Zhenguo Li, Wei Bi, Guorui Zhou, and 1 others. 2026. Dreamon: Diffusion language models for code infilling beyond fixed-size canvas. *arXiv preprint arXiv:2602.01326*.

Zhihui Xie, Jiacheng Ye, Lin Zheng, Jiahui Gao, Jingwei Dong, Zirui Wu, Xueliang Zhao, Shansan Gong, Xin Jiang, Zhenguo Li, and 1 others. 2025. Dreamcoder 7b: An open diffusion language model for code. *arXiv preprint arXiv:2509.01142*.

Jiacheng Ye, Zhihui Xie, Lin Zheng, Jiahui Gao, Zirui Wu, Xin Jiang, Zhenguo Li, and Lingpeng Kong. 2025. Dream 7b: Diffusion large language models. *arXiv preprint arXiv:2508.15487*.

Andrew Zhang, Anushka Sivakumar, Chia-Wei Tang, and Chris Thomas. 2025. Flexible-length text infilling for discrete diffusion models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 31332–31347.

Fengqi Zhu, Rongzhen Wang, Shen Nie, Xiaolu Zhang, Chunwei Wu, Jun Hu, Jun Zhou, Jianfei Chen, Yankai Lin, Ji-Rong Wen, and 1 others. 2025a. Llada 1.5: Variance-reduced preference optimization for large language diffusion models. *arXiv preprint arXiv:2505.19223*.

Fengqi Zhu, Zebin You, Yipeng Xing, Zenan Huang, Lin Liu, Yihong Zhuang, Guoshan Lu, Kangyu Wang, Xudong Wang, Lanning Wei, and 1 others. 2025b. Llada-moe: A sparse moe diffusion language model. *arXiv preprint arXiv:2509.24389*.

A Appendix

A.1 Length Probing Effectiveness

To isolate the quality of length estimation from the rest of the pipeline, we directly evaluate predicted lengths against the oracle length L^* on Human-S with LLaDA-8B-Base. We report mean absolute error (MAE) and Acc@k, the fraction of examples whose predicted length falls within a window of size k around L^*. For CAL, which searches from a preset initial length l, we sweep $l \in \{4, 8, 16, 32\}$; PILL predicts the length directly and uses no initial length.

Table 4 shows two findings. First, CAL is highly sensitive to its initial length: its MAE degrades from 3.56 at l=4 to 24.13 at l=32, and Acc@1 collapses from 0.58 to 0.02, confirming that a poorly chosen initial length is difficult to recover from. PILL removes this dependence entirely and attains the lowest MAE (2.50) with a single length-free prediction, matching or surpassing CAL's best per-l result without any tuning. Second, although CAL with a well-chosen small initial length (l=4) reaches a higher Acc@1, PILL is substantially better at Acc@3 (0.78) and Acc@5 (0.86). This is exactly the regime that matters for our method: PILL does not rely on a single exact hit but expands the prediction into a candidate set $\{\hat{L} - r, \ldots, \hat{L} + r\}$ and selects among them, so a high probability of placing L^* within a small window around the estimate—rather than landing on it precisely—is what drives final infilling quality. The strong Acc@5 indicates that the oracle length almost always lies within the candidate window, consistent with the PILL (Oracle) upper bound in Table 1.

Method	**MAE**	**Acc@1**	**Acc@3**	**Acc@5**
CAL ($l = 4$)	3.5644	0.5808	0.7086	0.7909
CAL ($l = 8$)	3.7106	0.5595	0.6912	0.7773
CAL ($l = 16$)	9.0600	0.2023	0.2730	0.3611
CAL ($l = 32$)	24.1336	0.0165	0.0203	0.0329
PILL	2.4974	0.4472	0.7822	0.8635

Table 4: Length prediction performance on LLaDA-8B-Base over Human-S dataset (l: initial length).

A.2 Effect of Slot Position-ID Design

Since most of the mainstream DLMs, such as LLaDA and Dream, use RoPE (Su et al., 2024), which is continuous and well-defined at non-integer positions (as exploited in (Chen et al., 2023; Zhang et al., 2025)), our interpolated position ids require no tuning; we find a frozen DLM generalizes to them at inference, outperforming padding, as introduced below.

We compare three ways of assigning position ids to a candidate slot of length ℓ_j: left-padding and right-padding, which align the slot to one side of the shared gap interval and leave a position-id gap on the other, and our interpolation scheme (Eq. 2), which spreads the slot evenly across the whole interval so every slot presents an identical layout to the suffix. Specifically, a length-one slot is placed at $p_P + 1$. The study uses LLaDA-8B-Base on Human-S with radius $r = 2$ (i.e., $N = 5$ candidates per example).

We report two metrics. *Avg Pass@1* is the pass rate averaged over all N candidates of each example, reflecting how good a typical candidate is. *Oracle Pass@1* selects, for each example, the best of the N candidates and reports its pass rate, reflecting how good the candidate set is at its best. As shown in Table 5, interpolation is clearly superior on both: it lifts the candidate average from 28.2/31.6 (left/right padding) to 51.6, and the oracle pass rate from 73.2/74.2 to 81.9.

Crucially, the gap remains large even under oracle selection, where the best candidate is hand-picked. This shows that the padding-induced misalignment is not a selection problem that a stronger scorer could fix, but a structural failure: the position-id gap causes under-generation within the slot itself, so no candidate in the set is correct to begin with. Interpolation removes this gap and is therefore essential to the multi-slot design.

We also report *Oracle (single slot)*, an upper bound where each candidate is decoded independently in its own full-length sequence and the best is selected, free of any multi-slot constraint. Our interpolation nearly matches it (81.9/51.6 vs. 83.6/52.9), showing that the shared layout with interpolated position ids sacrifices almost no quality, whereas padding falls short by 9-10 points-a structural gap, not a selection issue.

Position-ID Design	Oracle Pass@1	Avg Pass@1
Left-padding	73.2	28.2
Right-padding	74.2	31.6
Interpolation (ours)	**81.9**	**51.6**
Oracle (single slot)	83.6	52.9

Table 5: Effect of slot position-id design (LLaDA-8B-Base, Human-S, r=2). *Avg Pass@1*: pass rate averaged over all candidates; *Oracle Pass@1*: pass rate of the best candidate per example.

A.3 Ablation of Post-hoc Selection Score

To make the role of each selection signal explicit, we summarize the ablation of the post-hoc selection score in Table 6. Random candidate uniformly selects one candidate from the decoded candidate set. Predicted length only uses only the center candidate given by the probed length $\hat{L}$, without post-hoc selection. Inner-only corresponds to $\alpha = 1$ in Eq. (3), using only the internal coherence score. Suffix-only corresponds to $\alpha = 0$, using only the suffix continuation score. Our default setting uses $\alpha = 0.5$, combining both signals.

As shown in Table 6, random selection performs poorly despite the candidate set containing strong completions, showing that the selection rule is necessary. Using only the predicted length gives a much stronger result, but it cannot correct local length prediction errors. Both inner-only and suffix-only scoring further improve over these simple alternatives, while combining the two scores achieves the best performance. This confirms that internal coherence and suffix continuation provide complementary signals for selecting the final infill.

Selection Strategy	Pass@1
Random candidate	51.62
Predicted length only	66.02
Inner-only ($\alpha = 1$)	69.02
Suffix-only ($\alpha = 0$)	63.99
Inner + Suffix ($\alpha = 0.5$)	**71.35**

Table 6: Ablation of the post-hoc selection score on LLaDA-8B-Base and HumanEval-S. Inner-only and suffix-only correspond to $\alpha = 1$ and $\alpha = 0$ in Eq. (3), respectively. Combining both internal coherence and suffix continuation achieves the best performance.

A.4 Per-Stage Overhead at a Fixed Length

To attribute cost to each stage, we decompose PILL's runtime on LLaDA-8B-Base (Human-S) against a controlled baseline: a single fixed-length decoding at the *same* predicted length $\hat{L}$ (denoted

Basic Dec.). This isolates the cost added by each stage from any effect of the decoding length itself, and therefore differs from the end-to-end comparison in Table 2, where the backbone is averaged over preset lengths $L \in \{4, 8, 16, 32\}$ and thus decodes at different lengths. As shown in Table 7, relative to this matched-length decoding, the probe, multi-slot decoding, and selection add 8.9%, 3.4%, and 9.4% respectively. Notably, the multi-slot stage adds almost nothing despite generating all $2r + 1$ candidates, confirming that the slot-wise parallel design decodes the full candidate set at a cost close to a single slot; the two lightweight probes account for the remainder.

Breakdown	Basic Dec.	Probe	Multi-slot	Selection	Total
Time Cost	0.967s	0.086s	0.033s	0.091s	1.177s
Overhead	-	8.9%	3.4%	9.4%	21.7%

Table 7: Per-stage overhead at a fixed length.

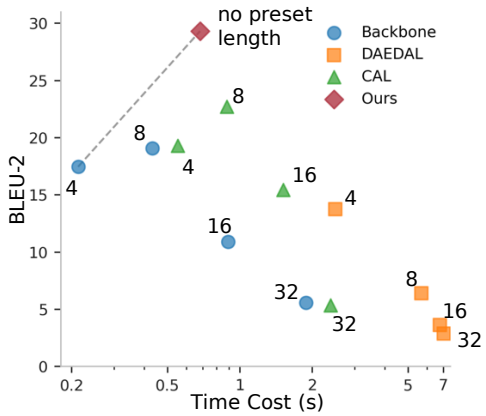


Figure 5: Quality-efficiency trade-off on Wikitext (BLEU-2 vs. wall-clock time, log-scaled x-axis). Each baseline is annotated with its preset length $L \in \{4, 8, 16, 32\}$; PILL requires no preset length. PILL sits at the upper-left frontier and dominates all baseline configurations.

A.5 Full Results across Initial Lengths

Table 8 reports the per-length results that are averaged into Table 1 of the main paper, on three categories of tasks: Python (Human-S), other language (Java), and text (Wikitext). Two patterns stand out. First, baseline quality is highly sensitive to the preset length and varies non-monotonically with it: the backbone peaks at L=16 on Human-S (56.24) but at L=8 on Wikitext (19.04), and even the strongest baseline CAL drops from 70.86 at L=4 to 50.82 at L=32 on Human-S, and from 22.69 to 5.35 over the same range on Wikitext. Crucially, the best-performing length differs across datasets, so no single preset length transfers, which is precisely the difficulty an adaptive method must resolve. Second, larger lengths inflate the time cost monotonically while quality does not follow, so the longer configurations pay more for worse results. In contrast, PILL uses no preset length and a single run already surpasses the best length-specific result of every baseline, at a time cost close to the cheapest backbone configuration.

A.6 Quality-Efficiency Trade-off

Figure 5 visualizes the quality–time trade-off on Wikitext, with each baseline shown at all four preset lengths $L \in \{4, 8, 16, 32\}$ and PILL shown as a single length-free point. PILL lies at the upper-left frontier, attaining the highest BLEU-2 at a time cost lower than nearly all baseline configurations; every baseline point is strictly dominated. Sweeping a baseline along L traces a curve that moves rightward (more time) without consistently moving upward (better quality), illustrating that tuning the preset length cannot recover the operating point PILL reaches without any tuning.

A.7 Details of the Length Probe

Architecture. The probe is implemented as a 3-layer MLP with hidden dimensions of 512 and 128. We use GeLU as the activation function.

Training Datasets. For each model, we train a probe. We collect 100k training samples for each probe, from two sources: (1) a code corpus (∼80k samples), including Py150 (Raychev et al., 2016), LeetCode, and CodeContests (Li et al., 2022b); and (2) a text corpus (∼20k samples) from C4 (Raffel et al., 2020). For each sample, we randomly mask a contiguous span from the original code or text sequence, using the remaining left and right contexts as the prefix and suffix, respectively. The length of the masked span is used as the target length. All training datasets are distinct from the evaluation datasets and are strictly non-overlapping. More details are provided in Sec. A.8.

Training Recipe. For the length probe, we train a lightweight MLP regressor with a regression objective. Unless otherwise specified, the probe is optimized with AdamW using a learning rate of 1×10^{-3}, weight decay of 1×10^{-4} , batch size of 16, and dropout rate of 0.1. The model is trained

(a) Results on Human-S dataset

Init Length	4		8		16		32		Avg	
	BLEU-2	**Time**	**BLEU-2**	**Time**	**BLEU-2**	**Time**	**BLEU-2**	**Time**	**BLEU-2**	**Time**
Backbone	22.36	0.279	51.21	0.567	56.24	1.155	48.89	2.411	44.68	1.103
DAEDAL	29.72	0.519	53.34	1.123	53.63	3.095	43.27	4.417	44.99	2.288
CAL	70.86	1.143	73.48	1.465	63.79	2.275	50.82	3.604	64.74	2.122
PILL	-	-	-	-	-	-	-	-	71.35	1.176

(b) Results on Java dataset

Init Length	4		8		16		32		Avg	
	BLEU-2	**Time**	**BLEU-2**	**Time**	**BLEU-2**	**Time**	**BLEU-2**	**Time**	**BLEU-2**	**Time**
Backbone	11.20	0.576	38.36	1.163	43.90	2.365	42.90	4.831	34.09	2.234
DAEDAL	14.75	0.828	41.24	1.520	44.79	4.265	40.58	5.800	35.34	3.103
CAL	49.78	2.379	54.99	3.039	50.89	4.432	45.57	7.038	50.31	4.222
PILL	-	-	-	-	-	-	-	-	57.10	2.555

(c) Results on Wikitext dataset

Init Length	4		8		16		32		Avg	
	BLEU-2	**Time**	**BLEU-2**	**Time**	**BLEU-2**	**Time**	**BLEU-2**	**Time**	**BLEU-2**	**Time**
Backbone	17.44	0.213	19.04	0.434	10.88	0.894	5.57	1.887	13.23	0.857
DAEDAL	13.73	2.494	6.39	5.681	3.62	6.794	2.87	7.040	6.65	5.502
CAL	19.27	0.553	22.69	0.883	15.44	1.513	5.35	2.387	15.69	1.334
PILL	-	-	-	-	-	-	-	-	29.29	0.928

Table 8: Baseline results under different initial lengths.

for up to 50 epochs with mean squared error (MSE) loss.

Discussion. All adaptive-length methods incur some preparation cost; the key difference lies in its *form* and *when it is paid*. The PILL probe is trained only *once* per backbone (about 100 minutes on a single NVIDIA A40 GPU) and is thereafter reused across all tasks and inputs with no further training, adding merely two forward passes at inference time. In contrast, CAL is *not* cost-free either: it re-initializes its confidence parameters for *each* setting (about 20 minutes, 2k samples), and this cost recurs *online*, since its iterative confidence search introduces many extra forward passes at inference (Table 2). DAEDAL likewise relies on fitted/tuned components rather than operating fully out of the box. PILL's cost is therefore a one-time, amortized expense, whereas the baselines pay repeatedly at inference. Moreover, the probe's training cost does *not* stem from overfitting to the evaluation distribution. As shown in our comparison with DreamOn (Section 5.5, Table 3), the text infilling tasks are out-of-distribution for our probe, which is fitted only on code; PILL nonetheless outperforms DreamOn across all of its length configurations on both datasets. This indicates that the length prediction generalizes beyond its training domain, so the gains arise from the method design rather than from in-distribution length statistics.

A.8 Details of Datasets

All datasets and models used in this work are publicly available under licenses permitting research use, and we use them accordingly. Also, all evaluation results for test dataset are from a single run with deterministic decoding. Following all prior adaptive-length infilling methods we compare against (Li et al., 2025a; Liu et al., 2026; Wu et al., 2026), we report results from a single run with deterministic decoding, which keeps the comparison protocol identical across methods.

Our use is consistent with these resources' intended research use, and the infilling benchmarks we derive from them are likewise intended for research only.

HumanEval. HumanEval-Infilling (Bavarian et al., 2022) is an infilling benchmark derived from the original HumanEval code generation benchmark. It evaluates whether a model can complete missing code spans given both the left and right contexts. Following prior work, the benchmark contains several infilling settings, including single-line, multi-line, random-span, and random-span-light, which cover different granularities and locations of missing code.

MBPP. Mostly Basic Python Problems (MBPP) (Austin et al., 2021b) is a Python program synthesis benchmark consisting of 974 crowd-sourced programming problems. Each example contains a natural language task description, a reference Python solution, and test cases for functional correctness. The problems are designed to be solvable by entry-level programmers and mainly cover basic programming concepts, standard library usage, and simple algorithmic reasoning.

Py150. Py150 (Raychev et al., 2016) is a large-scale Python corpus collected from GitHub repositories. The dataset contains 150k Python files, split into 100k training files and 50k evaluation files. The original release provides parsed Python ASTs as well as source files, and applies filtering steps such as removing duplicate files, excluding repository forks, keeping files that can be parsed, and focusing on permissive licenses. We use Py150 as a source of natural Python code for constructing length-probe training samples.

LeetCode. The LeetCode dataset[1] contains algorithmic programming problems from LeetCode, including problem titles, difficulty levels, descriptions, and reference solutions in multiple programming languages such as Python, Java, C++, and JavaScript. Compared with MBPP, these examples are generally more algorithmic and closer to interview-style coding tasks.

CodeContests. CodeContests (Li et al., 2022b) is a competitive programming dataset released by DeepMind and used in the development of AlphaCode. It contains programming problems collected from multiple online judge platforms, such as Aizu, AtCoder, CodeChef, Codeforces, and HackerEarth. Each problem includes natural language statements, input-output test cases, and both correct and incorrect human submissions in multiple programming languages. We use it as a source of challenging algorithmic code.

MultiPL-E. MultiPL-E (Cassano et al., 2022) is a multilingual code generation benchmark that translates unit-test-driven Python benchmarks, including HumanEval and MBPP, into 18 additional programming languages. It is designed to evaluate whether code generation models generalize beyond Python and across diverse programming languages. In our setting, MultiPL-E provides multilingual code examples and evaluation-style prompts for constructing infilling instances.

WikiText. WikiText (Merity et al., 2016) is a language modeling dataset extracted from verified Good and Featured articles on Wikipedia. Compared with heavily preprocessed corpora such as Penn Treebank, WikiText preserves original casing, punctuation, numbers, and long-form article structure, making it suitable for evaluating long-range dependencies in natural language modeling. We use WikiText as a source of high-quality natural text for non-code infilling samples.

C4. The Colossal Clean Crawled Corpus (C4) (Raffel et al., 2020) is a large-scale cleaned web corpus derived from Common Crawl and introduced with the T5 framework. It contains diverse English web text after a series of filtering and cleaning steps. We use C4 as a broad-domain natural language corpus to improve the coverage of text-side length-probe training examples.

arXiv. The arXiv dataset[2] provides metadata and abstracts for scholarly articles across domains such as physics, computer science, statistics, electrical engineering, quantitative biology, and economics. We use arXiv abstracts as scientific-domain natural text, which complements the general-domain text from C4 and WikiText.

A.9 Potential Risks

PILL is a backbone-frozen method for diffusion language models: it does not modify the underlying backbone, and its lightweight probe is trained only on public corpora. We do not expect any personally identifying information in these resources. As our method builds on pretrained DLMs, it may

[1] https://huggingface.co/datasets/greengerong/leetcode.

[2] https://huggingface.co/datasets/UniverseTBD/arxiv-abstracts-large.

inherit biases or factual errors present in these backbones, and the infilled spans it produces should be reviewed before use in sensitive applications.

A.10 Declaration of AI Assistance

We used AI tools for language polishing and part of the experimental implementation. These tools did not contribute to the idea, analysis, results, or scientific claims.