FROM INTERFACE TO INFERENCE: ELICITING ANY-ORDER INFERENCE FROM ANY-ORDER MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Many discrete reasoning tasks, such as code generation, are inherently non-causal: programmers move between high-level structure and local details, a process we call *any-order inference*. For autoregressive language models, which lack a native any-order interface, non-causal abilities such as infilling and next-edit prediction require hand-designed mechanisms. Can we instead design models that natively support any-order inference? Masked diffusion models have recently emerged as compelling candidates, as their any-order training objective naturally offers an any-order prediction interface. This *interface*, however, does not automatically yield any-order *inference*. We demonstrate that this interface-inference gap stems from *positional uncertainty*: fixed-canvas, token-level models may know *what* semantic component should appear without knowing *where* to place it. In light of this, we propose two complementary approaches: (1) *Insertion-based masked diffusion*, building on FlexMDM (Kim et al., 2025a), relaxes fixed-position commitments via insertions, enabling generation across non-contiguous regions. (2) *Latent-space masked diffusion* shifts prediction to coarser semantic segments, enabling search over latent generation orders. Empirically, we train a 7B FlexMDM for Python coding and a 125M LatentMDM for GSM8K and show that both approaches induce distinct any-order inference behaviors and improve downstream performance.

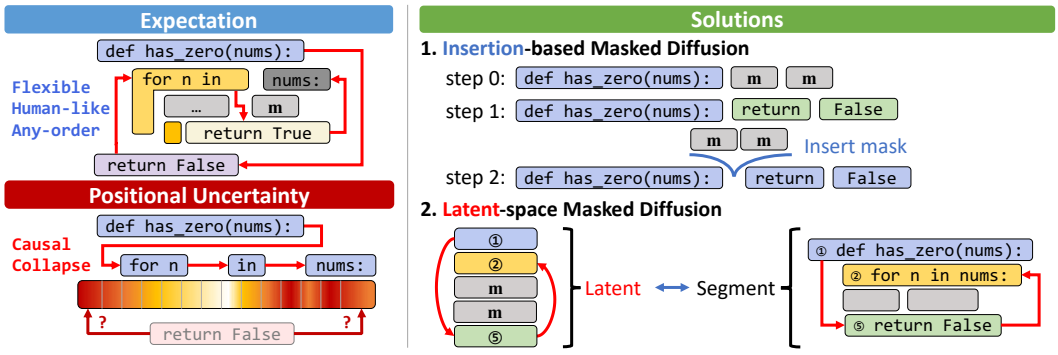


Figure 1: **Illustrative overview of our work.** Masked diffusion models cannot perform genuine *any-order* inference as they operate on a fixed canvas at the token level. We demonstrate that insertion-based and latent-space masked diffusion modeling overcome this limitation, enabling genuine *any-order* inference.

1 INTRODUCTION

Many real-world discrete reasoning tasks are inherently non-causal: when humans construct complex discrete objects such as code, they revise earlier decisions, insert missing pieces, and alternate between unfolding high-level structure and filling in local details. Autoregressive language models, the dominant paradigm in discrete generative modeling, decode left-to-right and thus do not natively support such trajectories. A substantial line of work has nevertheless extended them with non-causal mechanisms such as infilling (Bavarian et al., 2022), reordering (Shah et al., 2024), and next-edit prediction (Lu et al., 2025). While this shows that any-order inference is a desirable capability, the task-specific nature of these approaches motivates more native and principled generative paradigms.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

We focus on masked diffusion models (MDMs) (Shi et al., 2024; Sahoo et al., 2024), a popular instantiation of diffusion language models (Nie et al., 2025; Xie et al., 2025; Gong et al., 2025; Bie et al., 2025). MDMs are natural candidates for any-order inference as their training objective endows them with an any-order prediction *interface* (Ou et al., 2024; Zheng et al., 2024; Kim et al., 2025c): given a partially masked sequence, they predict clean tokens at arbitrary masked positions. This interface makes any-order inference possible in principle, but not automatic: the decoding policy must still choose which positions to reveal at each step, and poor choices can collapse generation order back toward left-to-right completion. We therefore ask whether MDMs actually deliver on this promise: *can an any-order prediction interface yield inference that is genuinely any-order in nature?*

Our contributions. In this work, we focus on code generation, a domain that would naturally benefit from genuine any-order inference. Our first finding is negative. We show that under confidence-based decoding, the dominant inference-time strategy for improving downstream performance, MDMs produce code that is *algorithmically close* to what they would have generated via strict left-to-right decoding (Section 3.1). This resonates with recent observations of *causal collapse* (Ni et al., 2026; Gong et al., 2025; Li et al., 2026), but we identify a more fundamental bottleneck underlying this phenomenon: *positional uncertainty*.

Positional uncertainty arises when the model knows *what* semantic component should be generated next, but remains uncertain about *where* to place it: its probability mass is spread across multiple plausible locations on the fixed masked canvas. Since confidence-based unmasking ranks individual positions, these globally meaningful but *positionally dispersed* components appear less confident than locally determined continuations, pushing decoding back toward left-to-right completion. These findings (Section 3.2) suggest that the any-order training objective of masked diffusion models does not guarantee genuine any-order inference *per se*. This motivates a broader question:

What generative models can achieve genuine any-order inference?

In this work, we propose two complementary remedies, each sidestepping positional uncertainty in a different way and inducing *distinct forms of any-order inference*. Our first solution, ***insertion-based masked diffusion*** (Section 4.1), builds on FlexMDM (Kim et al., 2025a), which equips MDMs with the ability to insert tokens. This circumvents the fixed-canvas bottleneck: unmasking no longer commits a token to a specific final position, since later insertions can shift its location in the final sequence. With this reduced token-commitment burden, we demonstrate that FlexMDM yields a structural form of any-order inference: under the parsed program tree, generation ***moves back and forth across different nodes*** rather than completing one contiguous region at a time. Empirically, we achieve this by significantly extending the practical scope of Kim et al. (2025a): we fine-tune Dream-Coder 7B (Xie et al., 2025) into a general-purpose Python-code-generating FlexMDM, which outperforms Dream-Coder 7B on HumanEval and HumanEval+ (Chen et al., 2021) at Pass@16, and moves ahead of it at Pass@1 on MBPP and MBPP+ (Austin et al., 2021) while remaining on par overall.

Our second solution, ***latent-space masked diffusion*** **(LatentMDM)** (Section 4.2), overcomes positional uncertainty by moving masked diffusion from token space to a continuous latent space. LatentMDM is distinct from prior embedding-space approaches, such as the Flow Map Language Model (Lee et al., 2026), and constitutes our main modeling contribution. Rather than selecting individual token positions, the model decides which masked latent segment to decode next, shifting the interface to a coarser semantic granularity. This enables a distinct form of any-order inference: ***searching over semantic generation orders in latent space***, allowing the model to discover orderings beyond the left-to-right token order. Unlike FlexMDM, LatentMDM has no off-the-shelf pretrained base model. We therefore pretrain a 125M-parameter latent masked diffusion model on TinyGSM (Liu et al., 2023) and show that it outperforms standard MDMs, embedding-space models such as DUO (Sahoo et al., 2025) and $\mathbb{S}$-FLM (Deschenaux & Gulcehre, 2026), and even an autoregressive model with KV caching under the same inference budget.

2 PRELIMINARIES

In this section, we review masked diffusion models (MDMs) (Shi et al., 2024; Sahoo et al., 2024).

Notation. Suppose our goal is to learn to generate samples from the data distribution $\mathbf{x} \sim p_{\text{data}}$ over length-L discrete sequences with a finite vocabulary $\mathcal{V}$. Let $\mathbf{x}^i$ denote the i-th element of a given sequence $\mathbf{x} = (\mathbf{x}^1, \dots, \mathbf{x}^L)$ and $\Delta(\mathcal{V})$ indicate the simplex of probability distributions over $\mathcal{V}$.

Training. Although MDMs admit several interpretations, we adopt an *any-order* language model interpretation (Ou et al., 2024; Zheng et al., 2024), which streamlines the prior account of MDM (Sahoo et al., 2024; Shi et al., 2024). Roughly speaking, MDMs introduce an auxiliary mask token $\mathbf{m}$ and learn, for each masked position, the posterior marginals of clean tokens conditioned on a masked sequence. To learn this posterior during training, one draws a clean sequence $\mathbf{x} \sim p_{\text{data}}$ and constructs a partially masked sequence $\mathbf{z}$ as follows: Sample $n \sim \text{Unif}\{0, \dots, L\}$ and replace the tokens at uniformly selected n indices in $\mathbf{x}$ with $\mathbf{m}$. Hence, a resulting $\mathbf{z}$ has n (randomly drawn) masked indices.

This masking procedure induces a joint distribution over $(\mathbf{x}, \mathbf{z})$ and we refer to the conditional marginal $\text{law}(\mathbf{x}^i \mid \mathbf{z})$ as the *unmasking posterior*. This unmasking posterior is the central object in MDMs and is modeled by a neural network f_θ that takes $\mathbf{z}$ as input and outputs a $|\mathcal{V}| \times L$ matrix. Concretely, its i-th column, $f_\theta^i(\cdot \mid \mathbf{z}) \in \Delta(\mathcal{V})$, models the unmasking posterior $f_\theta^i(v \mid \mathbf{z}) \approx p(\mathbf{x}^i = v \mid \mathbf{z})$. To train f_θ, we minimize cross-entropy loss summed over all masked indices.

$$\mathcal{L}(\theta) \colon = \mathbb{E}_{\mathbf{x},\mathbf{z}} \left[\frac{1}{n} \sum_{i \colon \mathbf{z}^i = \mathbf{m}} - \log f_\theta^i(\mathbf{x}^i \mid \mathbf{z}) \right].$$

MDM inference. A key feature of MDMs is their *any-order objective*: a pretrained MDM has learned to predict the posterior of *any masked position* in a sequence. This any-order property builds upon earlier work in masked language modeling (Devlin et al., 2019; Ghazvininejad et al., 2019; Wang & Cho, 2019). Next, we explain how this any-order property translates to the flexibility at inference time.

MDM inference starts from a length-L masked sequence $\mathbf{x}_1 = (\mathbf{m}, \dots, \mathbf{m})$ or generally a given prompt $\mathbf{x}_1 = (\texttt{[prompt]}, \mathbf{m}, \dots, \mathbf{m})$, and proceeds over a monotonically decreasing time grid $t_0 = 1 > \cdots > t_N = 0$. At each step t_ℓ, given a partially masked sequence $\mathbf{x}_{t_\ell} \in (\mathcal{V} \cup \{\mathbf{m}\})^L$, we proceed in two steps to obtain $\mathbf{x}_{t_{\ell+1}}$: **(a)** Choose a subset of masked positions $\mathcal{S} \subseteq \{i \mid \mathbf{x}_{t_\ell}^i = \mathbf{m}\}$ and **(b)** For each $i \in \mathcal{S}$, unmask $\mathbf{x}_{t_\ell}^i$ to a clean token v sampled from $v \sim f_\theta^i(\cdot \mid \mathbf{x}_{t_\ell}) \in \Delta(\mathcal{V})$. Notably, as the MDM training is any-order, i.e., f_θ predicts the clean token distribution over all masked positions, there is flexibility in the choice of $\mathcal{S}$, which is central to the downstream performance. Below, we describe several strategies for choosing $\mathcal{S}$ in practice.

Training-free methods, including Zheng et al. (2024); Kim et al. (2025c); Peng et al. (2025); Ben-Hamu et al. (2025); Nie et al. (2025); Wu et al. (2025b); Hayakawa et al. (2025), typically first compute a confidence score for each masked position and select a set $\mathcal{S}$ of the positions with the highest-scoring indices, i.e., $\mathcal{S} \leftarrow \text{TopK}_{i \colon \mathbf{x}_t^i = \mathbf{m}} [\text{score}(i)]$, where $\text{score}(i)$ quantifies how certain the model prediction is at position i.

Common recipes for $\text{score}(i)$ include the maximum predicted probability $\max_{v \in \mathcal{V}} f_\theta^i(v \mid \mathbf{x}_t)$, the margin between the top two probabilities $f_\theta^i(v_1 \mid \mathbf{x}_t) - f_\theta^i(v_2 \mid \mathbf{x}_t)$ (where v_1, v_2 are the top-2 tokens with the highest predicted probabilities), and the negative entropy of the categorical distribution. Some variants further incorporate an explicit position-dependent bias toward leftmost positions, often referred to as semi-autoregressive decoding. We refer to this broad family of training-free rules as *confidence-based decoding*.

3 The Interface-to-Inference Gap

As discussed in the introduction, many discrete objects, such as code, admit natural non-left-to-right generation orders. MDMs are appealing candidates for modeling such non-causal procedures due to their *any-order* interface. So far, however, this inference-time flexibility has primarily been used for efficient parallel decoding (DeepMind, 2025; Inception-Lab et al., 2025; Wu et al., 2025a;b; Song et al., 2025b). These works show that MDMs can exploit parallelism, but leave open a more fundamental question: *do MDMs actually exploit any-order flexibility to reason more effectively?*

Prior work gives positive evidence in restricted settings: on small, highly structured domains such as logical puzzles, MDMs can exploit their any-order interface to follow non-autoregressive generation orders and outperform autoregressive baselines (Ye et al., 2024a; Kim et al., 2025c; Trainin et al., 2026). At larger scales, however, recent empirical work paints a less optimistic picture: with confidence-based decoding, the dominant inference strategy in practice, MDMs exhibit a form of *causal collapse* on standard math and coding benchmarks. In particular, the token generation order

remains close to autoregressive. These findings suggest that any-order training alone **does not guarantee genuinely any-order inference** once MDMs are scaled.

Comparison to prior work. Prior work finds that confidence-based MDM decoding can reduce sample diversity (Ni et al., 2026; Fang et al., 2026) and yield AR-like generation orders (Gong et al., 2025; Li et al., 2026). We trace this collapse on Python programs to ***positional uncertainty*** in token-level, fixed-canvas inference and quantify how closely its outputs structurally approximate left-to-right decoding. Additional comparisons and alternative explanations appear in Appendix A.

Summary of the results. In Section 3.1, we first characterize the causal collapse from a *structural* perspective. Beyond analyzing the generation order itself, using tree-based parsing of generated code, we show that confidence-based MDM decoding produces samples that are **structurally similar** to those generated in a strictly left-to-right manner, and that MDM's apparent flexible interface does not translate into inference that exploits the hierarchical structure of programs. In Section 3.2, we then formalize *positional uncertainty* as the inevitable bottleneck that arises when token-level, fixed-canvas decoding must decide both ***what* should be generated** and ***where* it should be placed**. We show that this uncertainty persists across different MDMs and tasks.

3.1 Semantic Collapse in Python Code Generation

In this section, we demonstrate that confidence-based MDM decoding produces programs that are structurally similar to those generated by strict left-to-right (L2R) decoding. For each pretrained MDM, we hold the model fixed and vary only the unmasking policy. The L2R baseline reveals the leftmost remaining masked position at each step, while sampling its token from the same learned posterior $f_\theta^i(\cdot \mid \mathbf{x}_t)$. We then compare the final programs generated by each non-L2R policy against the corresponding L2R generations, using a structural similarity metric and best-match aggregation described below and detailed in Appendix B.

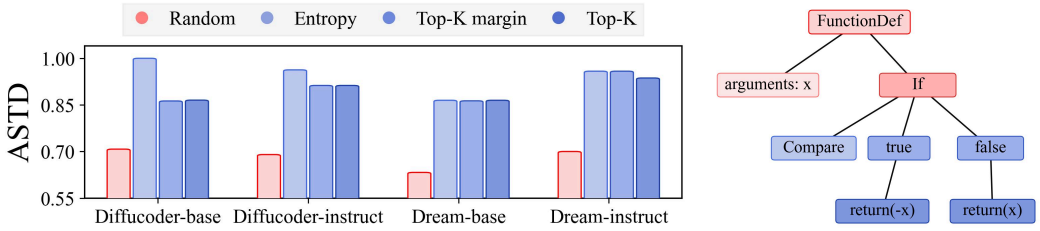


Figure 2: **(Left)** Confidence-based decoding yields code structurally close to left-to-right decoding. **(Right)** A simplified AST of the python function that returns `-x` if `x < 0` and `x` otherwise.

Code similarity metrics. To compare program structure, we use a standard AST-based tree-edit comparison. We parse each generated program with Python's built-in ast module, yielding an abstract syntax tree (see Figure 2 for example) whose nodes represent syntactic components such as functions, branches, loops, assignments, and returns. We then compute normalized tree-edit similarity between ASTs; higher values indicate more similar program structure. We refer to this score as *abstract syntax tree distance* (ASTD) and provide precise definitions, variants, and additional results in Appendix B.

Experiment setup. We take several large-scale pretrained MDMs, including their base and instruction- or RL-fine-tuned variants; Dream-7B-Base/Instruct (Xie et al., 2025), DiffuCoder-7B-Base/Instruct/cpGRPO (Gong et al., 2025), and Dream-Coder-7B-Instruct (Xie et al., 2025). On problems from HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and LiveCodeBench (Jain et al., 2024), we generate samples with sequence length 256 and 256 steps; further sampling details are given in Appendix B. We compare three confidence-based policies—Top-K, Top-K margin, and entropy-based decoding—against strict L2R decoding. For each prompt, we draw one sample from each confidence-based policy, since confidence-based decoding produces little sample diversity under our sampling settings. In contrast, we draw 32 samples from L2R and random decoding. For L2R, this gives a reference set for asking whether samples from other policies are already covered by L2R-biased generation; for random decoding, it provides a calibration baseline for how structurally diverse valid or correct programs can be when the reveal order has no causal bias. The larger sample budget also offsets the lower valid/correct rate of L2R and random decoding.

Best-match aggregation and filters. Since L2R is our reference for structurally left-to-right-biased generation, we examine whether each sample produced by a given policy has a close counterpart

among the L2R samples for the same prompt. We thus use *best-match aggregation*: for each generated program, we compute its similarity to the most similar L2R program, and then average these best-match scores across prompts. A high score means that the policy mostly discovers structures already covered by L2R decoding; a low score means that it finds programs structurally outside the L2R reference set. We compute this under two filters: valid–valid, where both programs parse, and success–success, where both programs pass the unit tests. Appendix B gives the precise definition.

Results. As shown in Figure 2, confidence-based decoding methods (blue bars) remain substantially closer to the L2R reference set than the Random-vs-L2R baseline (red bar). This trend holds across models and tasks, as shown in additional results in Appendix B. Thus, **the programs produced by confidence-based decoding share the same global skeleton and hierarchical structure** as L2R-generated code. By contrast, the lower Random-vs-L2R similarity shows that the benchmarks admit structurally different valid or correct programs, but confidence-based MDM decoding tends not to discover them.

3.2 Diagnosing Positional Uncertainty in MDMs

The observation in Section 3.1 raises the next question about *why* this causal collapse occurs. In this section, we argue that the key bottleneck lies in the fixed-canvas, token-level nature of MDMs.

Example: the *what* and *where* of a return node. Consider the code in Figure 1. The model may know that a fallback `return False` is needed, while its location—inside a loop, inside a branch, or after the loop—remains unresolved. In a fixed-canvas MDM, this positional uncertainty spreads probability mass across plausible locations and lowers confidence at each one. Confidence-based decoding therefore favors locally determined continuations and drifts toward left-to-right generation.

Positional uncertainty. We now formalize the intuition for *positional uncertainty* stated above. For a token v, we define its *aggregate mass* as $m(v; \mathbf{x}_t) = \sum_{i:\, \mathbf{x}_t^i = \mathbf{m}} f_\theta^i(v \mid \mathbf{x}_t)$. This measures the total probability assigned to token v across all currently masked positions. To measure whether this mass is concentrated at a single location or dispersed across many plausible locations, we define the *localization score*

$$\mathrm{LOC}(v; \mathbf{x}_t) := \max_{i:\, \mathbf{x}_t^i = \mathbf{m}} \left[\frac{f_\theta^i(v \mid \mathbf{x}_t)}{m(v; \mathbf{x}_t)} \right].$$

Thus, $\mathrm{LOC}(v; \cdot)$ is the largest fraction of token v's aggregate mass assigned to any one masked position. High values indicate a clear placement; low values imply dispersal. The case where the model knows *what to generate but not where to place it* corresponds to a token v with (1) high aggregated mass $m(v; \mathbf{x}_t)$ but (2) low $\mathrm{LOC}(v; \cdot)$.

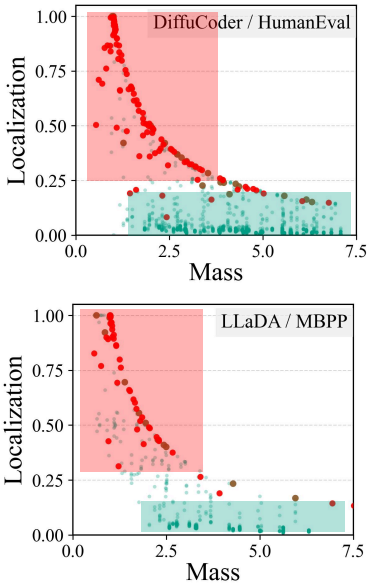


Figure 3: **Positional Uncertainty.**

Results. We examine whether positional uncertainty appears in practice. For each prompt and each partial sequence $\mathbf{x}_t$, we collect two types of points. First, to visualize tokens the model assigns high overall probability to, we take the top two tokens v with the largest aggregate mass $m(v; \mathbf{x}_t)$ and plot $(m(v; \mathbf{x}_t), \mathrm{LOC}(v; \mathbf{x}_t))$ in green. Second, to visualize what the model actually commits, for each revealed token v at position i, we plot its aggregate mass together with its committed-position localization $f_\theta^i(v \mid \mathbf{x}_t)/m(v; \mathbf{x}_t)$ in red. We aggregate these points across prompts and provide additional plots in Appendix C.

The resulting pattern in Figure 3 supports our claim. Many high-mass tokens lie in the lower-right region of the plot, indicating that the model assigns them substantial total probability, but does not decode them due to low localization. In contrast, committed tokens tend to have lower aggregate mass but higher localization, appearing toward the upper-left region.

Similarity analysis for learned decoding strategies. Post-trained unmasking policies may mitigate this bottleneck (Hong et al., 2025; Jazbec et al., 2025; Chen et al., 2025; Ye et al., 2024b). We analyze the reinforcement-learned unmasking policy of Chen et al. (2025). Although it changes the distribution over decoding orders, its outputs remain structurally close to strict left-to-right decoding (Appendix B.2). Thus, learning a policy alone may be insufficient to escape causal collapse in fixed-canvas token-level MDMs.

We do not claim this as an exhaustive evaluation of all planner-learning methods, since some of the public checkpoints are not available. Nevertheless, this result supports our broader view that positional uncertainty is a more fundamental limitation of fixed-canvas token-level MDMs for enabling genuine any-order inference, rather than merely a weakness of a particular confidence score.

4 Circumventing Positional Uncertainty

We identified positional uncertainty as a key bottleneck of MDMs. This suggests that genuine any-order inference may require relaxing at least one of two aspects of MDM generation: ***token-level*** and ***fixed-canvas***. In this section, we present two alternative generative models that each relax one of these aspects. We first summarize their modeling principles, then provide detailed descriptions and empirical evidence in Section 4.1 and Section 4.2.

FlexMDM: relaxing fixed-canvas. Our first approach is to employ FlexMDM (Kim et al., 2025a), an insertion-based masked diffusion model that not only decodes mask tokens but also inserts new masks during inference. This relaxes positional uncertainty since revealing a token no longer irrevocably commits it to a fixed position: later insertions can shift its location in the final sequence.

This relaxation induces a structural form of *any-order inference*. When we view generated code through its parsed algorithmic tree, we find that FlexMDM generation moves back and forth across different semantic nodes, rather than completing one contiguous region before moving to the next, thereby going beyond left-to-right generation. In this sense, insertion gives a mechanism for drafting multiple semantic regions and refining them later, closer to how human programmers develop code.

LatentMDM: relaxing token-level. Our second approach moves the masked-diffusion interface from individual tokens to segment-level latent variables. We split each sequence into coarse segments, such as lines of code, and jointly train a lightweight encoder-decoder pair that maps between segments and latent vectors, together with a masked diffusion model over the sequence of segment latents.

At inference time, the resulting LatentMDM chooses which *masked latent segment* to reveal next, and the decoder realizes the chosen latent as tokens. This switches the object of any-order inference from token positions to latent segments. By absorbing token-level positional uncertainty into segment-level latent variables, the revealed confidence becomes less tied to local token-position certainty and more tied to the compatibility of higher-level semantic components. **Thus, the form of any-order reasoning realized by LatentMDM is semantic-order search in a more compact exploration space:** deciding which high-level latent segment to instantiate next.

Experiments. We present a suite of experiments to answer the following questions:

Q1. ***Do our solutions mitigate causal collapse by relaxing positional uncertainty?***

Q2. ***What new forms of any-order inference do our solutions enable?***

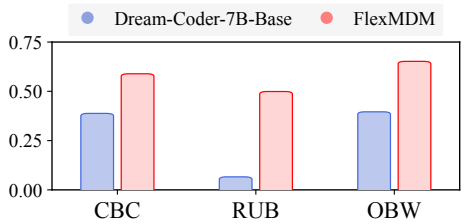
4.1 FlexMDM

Now we describe the training and inference procedures of FlexMDM and present empirical results obtained by fine-tuning Dream-Coder 7B (Xie et al., 2025).

Training. To enable insertion during training, FlexMDM augments the standard MDM masking procedure (recall Section 2) with *deletion*. As a result, during training, FlexMDM observes not only masked but also deleted versions of sequences $\mathbf{x} \sim p_{\text{data}}$. The masking and deletion rates are determined by two schedules: an unmasking and an insertion schedule. Given a corrupted sequence $\mathbf{z}$, a FlexMDM is trained to jointly predict the posterior distribution over masked tokens and the expected number of tokens to insert, conditioned on the current sequence $\mathbf{z}$.

Inference. At inference time, given a trained FlexMDM and the corresponding schedules, generation proceeds by alternating between two operations: (1) selecting which masked tokens to reveal according to the unmasking schedule and model's predicted unmasking posterior, and (2) inserting additional mask tokens according to the model's predicted insertion counts. Importantly, FlexMDM retains the any-order flexibility of standard MDM: for unmasking, one can still use confidence-based decoding to decide which masked tokens to reveal. For insertion, we sample the number of mask tokens to insert from a Poisson distribution whose rate is determined by the training-time insertion schedule together with the model's prediction. Like unmasking, insertion carries its own inference-time temperature: the per-gap insertion rates factorize into a total count and a placement distribution,

and tempering only the placement controls *where* new masks open while preserving the expected generation length (Appendix D.5). We defer technical details to Appendix D.1.



Benchmark	Model	@1	@4	@8	@16
HumanEval	Dream-Coder	58.65	81.34	86.94	90.85
	FlexMDM	50.65	78.69	86.86	92.07
HumanEval+	Dream-Coder	53.89	75.96	81.53	85.37
	FlexMDM	46.61	73.83	82.07	87.80

Figure 4: **(Left)** We quantify any-order inference by tracking how generation moves across the tree. **(Right)** FlexMDM outperforms Dream-Coder 7B on HumanEval and HumanEval+, at Pass@16.

Experimental setup. Rather than training FlexMDM from scratch, we initialize from Dream-Coder 7B (Xie et al., 2025), while attaching an auxiliary head and an AdaLN time embedding (Peebles & Xie, 2023) for insertion prediction. We construct the fine-tuning corpus by post-processing Python data from OpenCodeInstruct (Ahmad et al., 2025), rStar-Coder (Liu et al., 2025b), KodCode (Xu et al., 2025), and opc-stage-2-educational (Huang et al., 2025a), resulting in ≈2.6M training sequences. We then fine-tune the model using the FlexMDM training loss for ≈1150 GPU hours.

Results. FlexMDM by construction addresses Q1 by decoupling unmasking from fixed-position commitment. This relaxation induces an *any-order inference behavior* (Q2): FlexMDM generates drafts of non-contiguous regions of the generated code. We quantify this *any-order inference* by parsing generated code into an AST (as in Section 3.1) and tracking how generation moves across the tree. We use three complementary metrics: Coverage Before Commitment (CBC), the average number of sibling nodes previewed before completing the first child block; Return to Unfinished Blocks (RUB), which measures back-and-forth revisits to unfinished nodes; and Open-Block Width (OBW), the maximum number of sibling blocks kept open simultaneously. FlexMDM yields substantially higher values than Dream-Coder's standard inference across all metrics (Figure 4, left), supporting our claim of achieving any-order inference with insertion-based models: they generate code across non-contiguous semantic components, whereas an MDM with confidence-based decoding largely collapses to an autoregressive order. We defer precise definitions and additional results to Appendix D.3.

Next, we test whether this any-order behavior translates into downstream gains. On HumanEval and HumanEval+, FlexMDM outperforms Dream-Coder, our primary baseline, at Pass@16 (Figure 4, right). This suggests that any-order inference may help by enabling more diverse algorithmic solution paths. On MBPP and MBPP+, lowering the insertion temperature to 0.6 puts FlexMDM ahead of Dream-Coder at Pass@1, with overall performance matching the baseline (Table 5); Appendix D.5 ablates this knob.

4.2 LatentMDM

In this section, we explain the training and inference recipes of LatentMDM, and present empirical results on pre-training a 125M-scale LatentMDM on TinyGSM (Liu et al., 2023).

Training. We partition each sequence $\mathbf{x}$ into at most L_s (variable-length) segments, $\mathbf{x} = (\mathbf{y}^1, \ldots, \mathbf{y}^{L_s})$, where each segment $\mathbf{y}^i$ belongs to the segment space $\mathcal{Y}$. An encoder $E_\phi : \mathcal{Y} \to \mathbb{R}^H$ maps each segment to a fixed-dimensional latent vector; therefore, a full sequence is represented as a tensor in $\mathbb{R}^{L_s \times H}$. We train a masked diffusion model in this latent space by applying masking *at the segment level*. Concretely, we draw a partially masked segment sequence $\mathbf{z} = (\mathbf{z}^1, \ldots, \mathbf{z}^{L_s})$ with masked segment indices $\mathcal{M}(\mathbf{z}) \subseteq \{1, \ldots, L_s\}$. For each $i \in \mathcal{M}(\mathbf{z})$, the segment is replaced by $\mathbf{m}$, while unmasked segments remain unchanged, i.e., $\mathbf{z}^i = \mathbf{y}^i$ for $i \notin \mathcal{M}(\mathbf{z})$. The latent input is then formed by encoding unmasked segments with E_ϕ and replacing each masked segment with a learnable mask embedding $\mathbf{e}_\mathbf{m} \in \mathbb{R}^H$.

The LatentMDM, parameterized as $f_\theta : \mathbb{R}^{L_s \times H} \to \mathbb{R}^{L_s \times H'}$, takes this partially masked latent sequence, conditioned on the prompt, and predicts a latent vector in $\mathbb{R}^{H'}$ for every masked segment. The LatentMDM preserves and projects each segment into $\mathbb{R}^{H'}$, which serves as the conditioning space for the decoder. Finally, an *autoregressive* decoder D_ψ reconstructs each latent representation

in $\mathbb{R}^{H'}$ token by token. The training objective is the average autoregressive cross-entropy loss over the masked segments:

$$\mathcal{L}(\theta, \phi, \psi) = \mathbb{E}_{\mathbf{x},\mathbf{z}} \left[\frac{1}{|\mathcal{M}(\mathbf{z})|} \sum_{i \in \mathcal{M}(\mathbf{z})} \sum_{j=1}^{\text{len}(\mathbf{y}^i)} -\log D_\psi \left(\mathbf{y}^i_j \mid \mathbf{y}^i_{<j}, \mathbf{h}^i_{\theta,\phi}(\mathbf{z}) \right) \right],$$

where $\mathbf{h}_{\theta,\phi}(\mathbf{z}) \in \mathbb{R}^{L_s \times H'}$ denotes the latent conditioning produced by LatentMDM, $\text{len}(\mathbf{y}^i)$ denotes the length of $\mathbf{y}^i$, and the expectation is taken over the masking procedure defined above. Importantly, we use an autoregressive decoder for local segment sampling, since next-token prediction modeling remains effective for local sequence modeling (Kim et al., 2025c).

Inference. At inference time, the model is given only the prompt and maintains a partially filled segment sequence $\mathbf{z}$. Initially, all segment positions are masked. At each iteration, the revealed segments are encoded by E_ϕ, while masked positions are represented by the learned mask embedding. The LatentMDM then predicts a contextual latent $\mathbf{h}^i_{\theta,\phi}(\mathbf{z})$ for each masked segment position i.

The key design knob is which segment to reveal next: the autoregressive decoder D_ψ tentatively generates a candidate segment $\hat{\mathbf{y}}^i$ token-by-token from $\mathbf{h}^i_{\theta,\phi}(\mathbf{z})$. This tentative decoding is performed in parallel across all masked segments. We score each candidate by its length-normalized token log-likelihood under D_ψ, commit the highest-scoring segment, re-encode it with E_ϕ, and update the corresponding masked position in $\mathbf{z}$ for the next iteration. Thus, the inference-time decoding strategy is driven by *segment-level confidence*, which we view as *any-order inference* over semantic space: the model reveals the latent segment whose prediction can be most confidently realized.

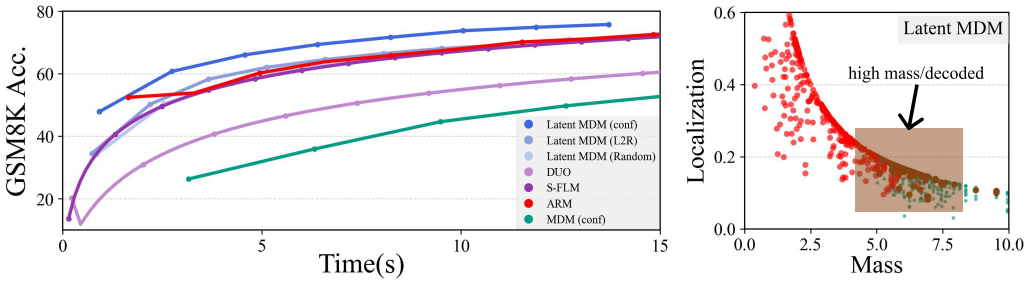


Figure 5: **(Left)** LatentMDM outperforms autoregressive and other baselines at matched wall-clock time. **(Right)** It mitigates positional uncertainty, decoding tokens with high aggregate mass.

Experimental setup. Since LatentMDM introduces fundamentally different modeling from MDMs, we had to train it from scratch. In particular, we target a setup where (1) models pretrained from scratch can achieve reasonable performance on downstream benchmarks, e.g., GSM8K (Cobbe et al., 2021), while (2) the compute budget remains feasible for running several controlled experiments. We adopt TinyGSM (Liu et al., 2023) as the pretraining corpus. TinyGSM converts GSM8K-style natural-language solutions into structured Python programs, making the task substantially more learnable at small model scales. We train LatentMDM and all baselines from scratch, using the same 125M-parameter scale with 500k optimizer steps and a global batch size of 256.

A1. To answer Q1, we revisit the *positional-uncertainty* diagnosis in Section 3.2. We examine the same mass-localization space defined by $m(v; \mathbf{x}_t)$ and $\text{LOC}(v; \mathbf{x}_t)$. Figure 5 (right) shows that LatentMDM commits a large fraction of high-mass but low-localization tokens (brown). This contrasts with MDM, where tokens in the same region remain undecoded (Figure 3). This demonstrates that moving the any-order interface from token positions to segment-level latents relaxes the positional bottleneck of fixed-canvas MDMs.

A2. To answer Q2, we scrutinize whether LatentMDM enables a useful form of any-order inference over semantic segments, in particular on GSM8K. We compare LatentMDM against representative discrete generative models: masked diffusion models (MDMs), autoregressive models, hyperspherical flow language model (Deschenaux & Gulcehre, 2026) ($\mathbb{S}$-FLM), and DUO (Sahoo et al., 2025).[1]

For hyperspherical flow language model and DUO, we largely follow the implementation details in the codebase of Deschenaux & Gulcehre (2026). Both methods support few-step sampling, which reduces

[1]We omit Flow Map Language Models (Lee et al., 2026), as Deschenaux & Gulcehre (2026) benchmark against it under an almost identical experimental setup.

latency but may come at the cost of lower accuracy. We therefore evaluate their latency–accuracy trade-offs across sampling steps in $\{8, 16, \ldots, 512\}$ and report the Pareto-optimal configurations: 16 steps for Spherical Flow Map Language Models and 8 steps for DUO. Moreover, we include two decoding-policy variants with the same LatentMDM: LatentMDM-random, which selects segments uniformly at random, and LatentMDM-L2R, which reveals segments in left-to-right order.

Fig. 5 (left) reports pass@K evaluation results, together with the corresponding wall-clock decoding time. The results show that LatentMDM consistently outperforms both LatentMDM-L2R and LatentMDM-random, by more than 6%. Since these variants differ only in their segment-selection policy, this result shows that the gain does not come merely from latent modeling, but from the ability to *choose which semantic segment to reveal next*, indicating that LatentMDM performs genuinely any-order decoding over segments.

We next compare against modeling baselines. Under a matched wall-clock budget, LatentMDM outperforms all other models by a gap of 5.5% at a sampling batch size of 1. This efficiency arises because one LatentMDM inference pass takes roughly half the latency of an autoregressive model, even with KV caching, as LatentMDM first samples latent segments, which are then decoded by a lightweight decoder. The gain further suggests that compact latent-space search preserves useful exploration, in contrast to token-level confidence decoding, where confidence over-selects locally easy positions and compresses sample diversity. We defer experimental details and additional results to Appendix E.

Together, these results support our central claim: moving the any-order interface from token positions to latent segments **relaxes the positional bottleneck, thus enabling semantic-order search through a distinct and practically advantageous form of any-order inference.**

Advances over prior work. For FlexMDM, we advance both the theory and the empirical scope of Kim et al. (2025a). Theoretically, we show that the schedules can be changed at inference time without retraining via a time reparameterization trick, yielding further flexibility at inference time (Appendix D.1). We further derive a count-preserving *insertion temperature* — an insertion-side analogue of the token temperature that gives flexible inference-time control over the structural stochasticity of insertion (Appendix D.5). We also reformulate insertion-length prediction in log space, which is geometrically more natural and leads to more stable training. Empirically, we move beyond the task-specific LoRA fine-tuning and fully fine-tune Dream-Coder-7B into a general-purpose Python FlexMDM. Thus, in our work, FlexMDM serves both as a theoretically extended model and a repurposed empirical vehicle for investigating how insertion can elicit genuinely any-order generation. For LatentMDM, we note that prior work has not studied masked diffusion over semantic latent spaces; we provide a detailed discussion of related work in Appendix A.

Discussion. FlexMDM and LatentMDM relax the positional bottleneck in complementary ways rather than serving as competing designs. FlexMDM stays in the token space, making it easier to *retrofit* from a pretrained MDM. LatentMDM enables segment-level order search but typically requires training from scratch. Combining insertion-based refinement with latent-space order search may realize the benefits of both forms of any-order inference; we leave this direction to future work.

5 Conclusion

Our two approaches, FlexMDM and LatentMDM, relax the fixed-canvas, token-level bottleneck of MDMs and thereby enable distinct forms of any-order inference. These approaches, however, also introduce trade-offs. More flexible inference procedures require additional design choices and may require extra care to ensure training and sampling stability. Moreover, our empirical study focuses primarily on Python code generation, leaving open whether similar forms of any-order inference transfer to broader domains, modalities, and scales.

Beyond masked-token diffusion, recent approaches based on continuous-state modeling or uniform-state diffusions (von Rütte et al., 2025; Sahoo et al., 2025; Lee et al., 2026; Roos et al., 2026; Potaptchik et al., 2026) may also support non-causal generation in principle. However, they lack the explicit any-order interface found in mask-based models, leaving the elicitation of genuine any-order inference in these alternative diffusion frameworks as an important open question. Another important direction is to investigate whether equipping masked diffusion models with self-correction mechanisms (Wang et al., 2026; Kim et al., 2025b; Huang et al., 2025b; Schiff et al., 2026) can elicit any-order inference behavior. More broadly, inference structure is a central modeling choice.

AI USE STATEMENT

We used generative AI tools to edit and polish the manuscript for clarity and readability. The authors reviewed the AI-assisted text and take responsibility for the final content of the paper.

REFERENCES

Manan Agarwal, Sheel Shah, Chanhyuk Lee, Jaehoon Yoo, Jerry Huang, Seunghoon Hong, Aditi Raghunathan, Jinwoo Kim, and Nicholas M Boffi. Posterior refinement: Fast language generation via any-order flow maps. *arXiv preprint arXiv:2606.24773*, 2026.

Wasi Uddin Ahmad, Aleksander Ficek, Mehrzad Samadi, Jocelyn Huang, Vahid Noroozi, Somshubra Majumdar, and Boris Ginsburg. OpenCodeInstruct: A large-scale instruction tuning dataset for code LLMs. *arXiv preprint arXiv:2504.04030*, 2025.

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

Loïc Barrault, Paul-Ambroise Duquenne, Maha Elbayad, Artyom Kozhevnikov, Belen Alastruey, Pierre Andrews, Mariano Coria, Guillaume Couairon, Marta R Costa-jussà, David Dale, et al. Large concept models: Language modeling in a sentence representation space. *arXiv preprint arXiv:2412.08821*, 2024.

Georgios Batzolis, Mark Girolami, and Luca Ambrogioni. Cobit: Language modeling with bitstream diffusion. *arXiv preprint arXiv:2605.07013*, 2026.

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

Heli Ben-Hamu, Itai Gat, Daniel Severo, Niklas Nolte, and Brian Karrer. Accelerated sampling from masked diffusion models via entropy bounded unmasking. *arXiv preprint arXiv:2505.24857*, 2025.

Tiwei Bie, Maosong Cao, Kun Chen, Lun Du, Mingliang Gong, Zhuochen Gong, Yanmei Gu, Jiaqi Hu, Zenan Huang, Zhenzhong Lan, Chengxi Li, Chongxuan Li, Jianguo Li, Zehuan Li, Huabin Liu, Ling Liu, Guoshan Lu, Xiaocheng Lu, Yuxin Ma, Jianfeng Tan, Lanning Wei, Ji-Rong Wen, Yipeng Xing, Xiaolu Zhang, Junbo Zhao, Da Zheng, Jun Zhou, Junlin Zhou, Zhanchao Zhou, Liwang Zhu, and Yihong Zhuang. LLaDA2.0: Scaling up diffusion language models to 100B, 2025. URL https://arxiv.org/abs/2512.15745.

Boyuan Chen, Diego Martí Monsó, Yilun Du, Max Simchowitz, Russ Tedrake, and Vincent Sitzmann. Diffusion forcing: Next-token prediction meets full-sequence diffusion. *Advances in Neural Information Processing Systems*, 37:24081–24125, 2024.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

Shirui Chen, Jiantao Jiao, Lillian J Ratliff, and Banghua Zhu. dUltra: Ultra-fast diffusion language models via reinforcement learning. *arXiv preprint arXiv:2512.21446*, 2025.

Yuxin Chen, Chumeng Liang, Hangke Sui, Ruihan Guo, Chaoran Cheng, Jiaxuan You, and Ge Liu. LangFlow: Continuous diffusion rivals discrete in language modeling. *arXiv preprint arXiv:2604.11748*, 2026.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Beiya Dai, Yuliang Liu, Daozheng Xue, Yunchong Song, Qipeng Guo, Kai Chen, Xinbing Wang, Bowen Zhou, and Zhouhan Lin. Context-level language modeling by learning predictive context embeddings. *arXiv preprint arXiv:2510.20280*, 2025.

Google DeepMind. Gemini diffusion, 2025. URL https://blog.google/technology/google-deepmind/gemini-diffusion/.

Justin Deschenaux and Caglar Gulcehre. Language modeling with hyperspherical flows. *arXiv preprint arXiv:2605.11125*, 2026.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pp. 4171–4186, 2019.

Liancheng Fang, Aiwei Liu, Henry Peng Zou, Yankai Chen, Enze Ma, Leyi Pan, Chunyu Miao, Wei-Chieh Huang, Xue Liu, and Philip S Yu. Locally confident, globally stuck: The quality-exploration dilemma in diffusion language models. *arXiv preprint arXiv:2604.00375*, 2026.

Marjan Ghazvininejad, Omer Levy, Yinhan Liu, and Luke Zettlemoyer. Mask-predict: Parallel decoding of conditional masked language models. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*, pp. 6112–6121, 2019.

Shansan Gong, Ruixiang Zhang, Huangjie Zheng, Jiatao Gu, Navdeep Jaitly, Lingpeng Kong, and Yizhe Zhang. Diffucoder: Understanding and improving masked diffusion models for code generation. *arXiv preprint arXiv:2506.20639*, 2025.

Daehoon Gwak, Minseo Jung, Junwoo Park, Minho Park, ChaeHun Park, Junha Hyung, and Jaegul Choo. Reward-weighted sampling: Enhancing non-autoregressive characteristics in masked diffusion LLMs. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 34562–34582, 2025.

Shibo Hao, Sainbayar Sukhbaatar, DiJia Su, Xian Li, Zhiting Hu, Jason Weston, and Yuandong Tian. Training large language models to reason in a continuous latent space. *arXiv preprint arXiv:2412.06769*, 2024.

Marton Havasi, Brian Karrer, Itai Gat, and Ricky TQ Chen. Edit flows: Flow matching with edit operations. *arXiv preprint arXiv:2506.09018*, 2025.

Satoshi Hayakawa, Yuhta Takida, Masaaki Imaizumi, Hiromi Wakaki, and Yuki Mitsufuji. Demystifying maskgit sampler and beyond: Adaptive order selection in masked diffusion. *arXiv preprint arXiv:2510.04525*, 2025.

Michael Hersche, Nicolas Menet, Ronan Tanios, and Abbas Rahimi. Locally coherent parallel decoding in diffusion language models. *arXiv preprint arXiv:2603.20216*, 2026.

Chunsan Hong, Seonho An, Min-Soo Kim, and Jong Chul Ye. Improving discrete diffusion unmasking policies beyond explicit reference policies. *arXiv preprint arXiv:2510.05725*, 2025.

Keya Hu, Linlu Qiu, Yiyang Lu, Hanhong Zhao, Tianhong Li, Yoon Kim, Jacob Andreas, and Kaiming He. Elf: Embedded language flows. *arXiv preprint arXiv:2605.10938*, 2026.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Weidi Xu, Jiaran Hao, Liuyihan Song, Yang Xu, Jian Yang, Jiaheng Liu, Chenchen Zhang, et al. OpenCoder: The open cookbook for top-tier code large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 33167–33193, 2025a.

Zemin Huang, Yuhang Wang, Zhiyang Chen, and Guo-Jun Qi. Don't settle too early: Self-reflective remasking for diffusion language models. *arXiv preprint arXiv:2509.23653*, 2025b.

Inception-Lab, Samar Khanna, Siddhant Kharbanda, Shufan Li, Harshit Varma, Eric Wang, Sawyer Birnbaum, Ziyang Luo, Yanis Miraoui, Akash Palrecha, et al. Mercury: Ultra-fast language models based on diffusion. *arXiv preprint arXiv:2506.17298*, 2025.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Metod Jazbec, Theo X Olausson, Louis Béthune, Pierre Ablin, Michael Kirchhof, Joao Monterio, Victor Turrisi, Jason Ramapuram, and Marco Cuturi. Learning unmasking policies for diffusion language models. *arXiv preprint arXiv:2512.09106*, 2025.

Haoqiang Kang, Yizhe Zhang, Nikki Lijing Kuang, Nicklas Majamaki, Navdeep Jaitly, Yi-An Ma, and Lianhui Qin. LaDiR: Latent diffusion enhances LLMs for text reasoning. *arXiv preprint arXiv:2510.04573*, 2025.

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

Jaeyeon Kim, Seunggeun Kim, Taekyun Lee, David Z Pan, Hyeji Kim, Sham Kakade, and Sitan Chen. Fine-tuning masked diffusion for provable self-correction. *arXiv preprint arXiv:2510.01384*, 2025b.

Jaeyeon Kim, Kulin Shah, Vasilis Kontonis, Sham Kakade, and Sitan Chen. Train for the worst, plan for the best: Understanding token ordering in masked diffusions. *arXiv preprint arXiv:2502.06768*, 2025c.

Jaeyeon Kim, Jonathan Geuter, David Alvarez-Melis, Sham Kakade, and Sitan Chen. Stop training for the worst: Progressive unmasking accelerates masked diffusion training. *arXiv preprint arXiv:2602.10314*, 2026.

Sean Lamont, Christian Walder, Paul Montague, Amir Dezfouli, and Michael Norrish. Free lunch for pass@ k? low cost diverse sampling for diffusion language models. *arXiv preprint arXiv:2603.04893*, 2026.

Chanhyuk Lee, Jaehoon Yoo, Manan Agarwal, Sheel Shah, Jerry Huang, Aditi Raghunathan, Seunghoon Hong, Nicholas M Boffi, and Jinwoo Kim. Flow map language models: One-step language modeling via continuous denoising. *arXiv preprint arXiv:2602.16813*, 2026.

Jia-Nan Li, Jian Guan, Wei Wu, and Chongxuan Li. ReFusion: A diffusion large language model with parallel autoregressive decoding. *arXiv preprint arXiv:2512.13586*, 2025.

Pengxiang Li, Dilxat Muhtar, Lu Yin, Tianlong Chen, and Shiwei Liu. Why diffusion language models struggle with truly parallel (non-autoregressive) decoding? *arXiv preprint arXiv:2602.23225*, 2026.

Tianhong Li, Yonglong Tian, He Li, Mingyang Deng, and Kaiming He. Autoregressive image generation without vector quantization. *Advances in Neural Information Processing Systems*, 37: 56424–56445, 2024.

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

Zhenghao Lin, Yeyun Gong, Yelong Shen, Tong Wu, Zhihao Fan, Chen Lin, Nan Duan, and Weizhu Chen. Text generation with diffusion language models: A pre-training approach with continuous paragraph denoise. In *International Conference on Machine Learning*, pp. 21051–21064. PMLR, 2023.

Bingbin Liu, Sebastien Bubeck, Ronen Eldan, Janardhan Kulkarni, Yuanzhi Li, Anh Nguyen, Rachel Ward, and Yi Zhang. TinyGSM: achieving >80% on GSM8k with small language models. *arXiv preprint arXiv:2312.09241*, 2023.

Jingyu Liu, Xin Dong, Zhifan Ye, Rishabh Mehta, Yonggan Fu, Vartika Singh, Jan Kautz, Ce Zhang, and Pavlo Molchanov. Tidar: Think in diffusion, talk in autoregression. *arXiv preprint arXiv:2511.08923*, 2025a.

Yifei Liu, Li Lyna Zhang, Yi Zhu, Bingcheng Dong, Xudong Zhou, Ning Shang, Fan Yang, and Mao Yang. rStar-Coder: Scaling competitive code reasoning with a large-scale verified dataset. *arXiv preprint arXiv:2505.21297*, 2025b.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=Bkg6RiCqY7.

Justin Lovelace, Varsha Kishore, Chao Wan, Eliot Shekhtman, and Kilian Q Weinberger. Latent diffusion for language generation. *Advances in Neural Information Processing Systems*, 36: 56998–57025, 2023.

Ruofan Lu, Yintong Huo, Meng Zhang, Yichen Li, and Michael R Lyu. Next edit prediction: Learning to predict code edits from context and interaction history. *arXiv preprint arXiv:2508.10074*, 2025.

Viacheslav Meshchaninov, Egor Chimbulatov, Alexander Shabalin, Aleksandr Abramov, and Dmitry Vetrov. Cosmos: Compressed and smooth latent space for text diffusion modeling. *arXiv preprint arXiv:2506.21170*, 2025.

Zanlin Ni, Shenzhi Wang, Yang Yue, Tianyu Yu, Weilin Zhao, Yeguo Hua, Tianyi Chen, Jun Song, Cheng Yu, Bo Zheng, et al. The flexibility trap: Why arbitrary order limits reasoning potential in diffusion language models. *arXiv preprint arXiv:2601.15165*, 2026.

Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Large language diffusion models. *arXiv preprint arXiv:2502.09992*, 2025.

Theo X Olausson, Metod Jazbec, Xi Wang, Armando Solar-Lezama, Christian A Naesseth, Stephan Mandt, and Eric Nalisnick. A tale of two temperatures: Simple, efficient, and diverse sampling from diffusion language models. *arXiv preprint arXiv:2604.09921*, 2026.

Jingyang Ou, Shen Nie, Kaiwen Xue, Fengqi Zhu, Jiacheng Sun, Zhenguo Li, and Chongxuan Li. Your absorbing discrete diffusion secretly models the conditional distributions of clean data. *arXiv preprint arXiv:2406.03736*, 2024.

Dhruvesh Patel, Tahira Naseem, Gaurav Pandey, Md Arafat Sultan, Andrew McCallum, and Ramón Fernandez Astudillo. Improved sampling from masked diffusion models with position contrastive guidance. In *NeurIPS 2025 Workshop on Structured Probabilistic Inference & Generative Modeling*, 2025a.

Dhruvesh Patel, Aishwarya Sahoo, Avinash Amballa, Tahira Naseem, Tim GJ Rudner, and Andrew McCallum. Insertion language models: Sequence generation with arbitrary-position insertions. *arXiv preprint arXiv:2505.05755*, 2025b.

Dhruvesh Patel, Benjamin Rozonoyer, Gaurav Pandey, Tahira Naseem, Ramón Fernandez Astudillo, and Andrew McCallum. Insertion based sequence generation with learnable order dynamics. *arXiv preprint arXiv:2602.18695*, 2026.

William Peebles and Saining Xie. Scalable diffusion models with transformers. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 4195–4205, 2023.

Fred Zhangzhi Peng, Zachary Bezemek, Sawan Patel, Jarrid Rector-Brooks, Sherwood Yao, Avishek Joey Bose, Alexander Tong, and Pranam Chatterjee. Path planning for masked diffusion model sampling, 2025. URL https://arxiv.org/abs/2502.03540.

Peter Potaptchik, Jason Yim, Adhi Saravanan, Peter Holderrieth, Eric Vanden-Eijnden, and Michael S Albergo. Discrete flow maps. *arXiv preprint arXiv:2604.09784*, 2026.

Patrick Pynadath, Jiaxin Shi, and Ruqi Zhang. CANDI: Hybrid discrete-continuous diffusion models. *arXiv preprint arXiv:2510.22510*, 2025.

Shuhui Qu. VDLM: Variable diffusion LMs via robust latent-to-text rendering. *arXiv preprint arXiv:2602.15870*, 2026.

Xingwei Qu, Shaowen Wang, Zihao Huang, Kai Hua, Fan Yin, Rui-Jie Zhu, Jundong Zhou, Qiyang Min, Zihao Wang, Yizhi Li, et al. Dynamic large concept models: Latent reasoning in an adaptive semantic space. *arXiv preprint arXiv:2512.24617*, 2025.

Daan Roos, Oscar Davis, Floor Eijkelboom, Michael Bronstein, Max Welling, İsmail İlkan Ceylan, Luca Ambrogioni, and Jan-Willem van de Meent. Categorical flow maps. *arXiv preprint arXiv:2602.12233*, 2026.

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

Subham Sekhar Sahoo, Justin Deschenaux, Aaron Gokaslan, Guanghan Wang, Justin Chiu, and Volodymyr Kuleshov. The diffusion duality. *arXiv preprint arXiv:2506.10892*, 2025.

Yair Schiff, Omer Belhasin, Roy Uziel, Guanghan Wang, Marianne Arriola, Gilad Turok, Ran Zilberstein, Michael Elad, and Volodymyr Kuleshov. Learn from your mistakes: Self-correcting masked diffusion models. *arXiv preprint arXiv:2602.11590*, 2026.

Johannes Schusterbauer, Ming Gui, Yusong Li, Pingchuan Ma, Felix Krause, and Björn Ommer. Denoising, fast and slow: Difficulty-aware adaptive sampling for image generation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 43260–43270, 2026.

Kulin Shah, Nishanth Dikkala, Xin Wang, and Rina Panigrahy. Causal language modeling can elicit search and reasoning capabilities on logic puzzles. *Advances in Neural Information Processing Systems*, 37:56674–56702, 2024.

Junzhe Shen, Jieru Zhao, Ziwei He, and Zhouhan Lin. CoDAR: Continuous diffusion language models are more powerful than you think. *arXiv preprint arXiv:2603.02547*, 2026a.

Yangyi Shen, Tianjian Feng, Jiaqi Han, Wen Wang, Tianlang Chen, Chunhua Shen, Jure Leskovec, and Stefano Ermon. Improving diffusion language model decoding through joint search in generation order and token space. *arXiv preprint arXiv:2601.20339*, 2026b.

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

Kiwhan Song, Boyuan Chen, Max Simchowitz, Yilun Du, Russ Tedrake, and Vincent Sitzmann. History-guided video diffusion. *arXiv preprint arXiv:2502.06764*, 2025a.

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

Sophia Tang, Yuchen Zhu, Molei Tao, and Pranam Chatterjee. A2d2: Fine-tuning any-length discrete diffusion for adaptive decoding. *arXiv preprint arXiv:2606.13565*, 2026.

Itamar Trainin, Shauli Ravfogel, Omri Abend, and Amir Feder. Discrete diffusion models exploit asymmetry to solve lookahead planning tasks. *arXiv preprint arXiv:2602.19980*, 2026.

Dimitri von Rütte, Janis Fluri, Omead Pooladzandi, Bernhard Schölkopf, Thomas Hofmann, and Antonio Orvieto. Scaling behavior of discrete diffusion language models. *arXiv preprint arXiv:2512.10858*, 2025.

Alex Wang and Kyunghyun Cho. Bert has a mouth, and it must speak: Bert as a markov random field language model. In *Proceedings of the workshop on methods for optimizing and evaluating neural language generation*, pp. 30–36, 2019.

Guanghan Wang, Yair Schiff, Subham Sahoo, and Volodymyr Kuleshov. Remasking discrete diffusion models with inference-time scaling. *Advances in Neural Information Processing Systems*, 38: 147282–147339, 2026.

Christopher Wewer, Bart Pogodzinski, Bernt Schiele, and Jan Eric Lenssen. Spatial reasoning with denoising models. *arXiv preprint arXiv:2502.21075*, 2025.

Chengyue Wu, Hao Zhang, Shuchen Xue, Shizhe Diao, Yonggan Fu, Zhijian Liu, Pavlo Molchanov, Ping Luo, Song Han, and Enze Xie. Fast-dLLM v2: Efficient block-diffusion LLM. *arXiv preprint arXiv:2509.26328*, 2025a.

Chengyue Wu, Hao Zhang, Shuchen Xue, Zhijian Liu, Shizhe Diao, Ligeng Zhu, Ping Luo, Song Han, and Enze Xie. Fast-dLLM: Training-free acceleration of diffusion LLM by enabling KV cache and parallel decoding. *arXiv preprint arXiv:2505.22618*, 2025b.

Zhihui Xie, Jiacheng Ye, Lin Zheng, Jiahui Gao, Jingwei Dong, Zirui Wu, Xueliang Zhao, Shansan Gong, Xin Jiang, Zhenguo Li, et al. Dream-Coder 7B: An open diffusion language model for code. *arXiv preprint arXiv:2509.01142*, 2025.

Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. KodCode: A diverse, challenging, and verifiable synthetic dataset for coding. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 6980–7008, 2025.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Jiacheng Ye, Jiahui Gao, Shansan Gong, Lin Zheng, Xin Jiang, Zhenguo Li, and Lingpeng Kong. Beyond autoregression: Discrete diffusion for complex reasoning and planning. *arXiv preprint arXiv:2410.14157*, 2024a.

Jiacheng Ye, Shansan Gong, Liheng Chen, Lin Zheng, Jiahui Gao, Han Shi, Chuan Wu, Xin Jiang, Zhenguo Li, Wei Bi, et al. Diffusion of thought: Chain-of-thought reasoning in diffusion language models. *Advances in Neural Information Processing Systems*, 37:105345–105374, 2024b.

Jaehoon Yoo, Wonjung Kim, Floor Eijkelboom, Chanhyuk Lee, Nicholas M Boffi, Seunghoon Hong, and Jinwoo Kim. Self-conditioned flow map language models via fixed-point flows. *arXiv preprint arXiv:2607.00714*, 2026.

Ruixiang Zhang, Shuangfei Zhai, Jiatao Gu, Yizhe Zhang, Huangjie Zheng, Tianrong Chen, Miguel Angel Bautista, Josh Susskind, and Navdeep Jaitly. Flexible language modeling in continuous space with transformer-based autoregressive flows. *arXiv preprint arXiv:2507.00425*, 2025.

Yangtian Zhang, Zhe Wang, Arthur Gretton, Rex Ying, David van Dijk, Michalis K Titsias, and Jiaxin Shi. Variational learning for insertion-based generation. *arXiv preprint arXiv:2606.02133*, 2026.

Yizhe Zhang, Jiatao Gu, Zhuofeng Wu, Shuangfei Zhai, Joshua Susskind, and Navdeep Jaitly. Planner: Generating diversified paragraph via latent language diffusion model. *Advances in Neural Information Processing Systems*, 36:80178–80190, 2023.

Huangjie Zheng, Shansan Gong, Ruixiang Zhang, Tianrong Chen, Jiatao Gu, Mingyuan Zhou, Navdeep Jaitly, and Yizhe Zhang. Continuously augmented discrete diffusion model for categorical generative modeling. *arXiv preprint arXiv:2510.01329*, 2025.

Kaiwen Zheng, Yongxin Chen, Hanzi Mao, Ming-Yu Liu, Jun Zhu, and Qinsheng Zhang. Masked diffusion models are secretly time-agnostic masked models and exploit inaccurate categorical sampling. *arXiv preprint arXiv:2409.02908*, 2024.

Cai Zhou, Chenxiao Yang, Yi Hu, Chenyu Wang, Chubin Zhang, Muhan Zhang, Lester Mackey, Tommi Jaakkola, Stephen Bates, and Dinghuai Zhang. Coevolutionary continuous discrete diffusion: Make your diffusion language model a latent reasoner. *arXiv preprint arXiv:2510.03206*, 2025.

Xiaochen Zhu, Georgi Karadzhov, Chenxi Whitehouse, and Andreas Vlachos. Segment-level diffusion: A framework for controllable long-form generation with diffusion language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4163–4183, 2025.

CONTENTS

1 Introduction **1**

2 Preliminaries **2**

3 The Interface-to-Inference Gap **3**

3.1 Semantic collapse in Python code generation 4

3.2 Diagnosing Positional Uncertainty in MDMs 5

4 Circumventing Positional Uncertainty **6**

4.1 FlexMDM 6

4.2 LatentMDM 7

5 Conclusion **9**

A Related Works **18**

B Details of Section 3.1 **20**

B.1 Additional similarity results 22

B.2 Semantic collapse in dUltra planner 22

C Details of Section 3.2 **23**

D Details of Section 4.1 **26**

D.1 FlexMDM Training 26

D.2 Training FlexMDM in Insertion-Progress Coordinates 28

D.3 Any-Order Metrics 30

D.4 FlexMDM Inference and Details on Evaluation Setup 32

D.5 Insertion Temperature 32

E Details of Section 4.2 **34**

E.1 Data Pre-processing 34

E.2 Architecture 35

E.3 Training Details 35

E.4 Inference Details 36

E.5 Additional Experiments 37

E.6 Generation Trace 37

A Related Works

Causal collapse in MDMs. Recent work has identified two related but distinct limitations of confidence-based decoding in masked diffusion models: reduced sample diversity and the collapse of flexible generation orders toward AR-like trajectories (Ni et al., 2026; Olausson et al., 2026; Shen et al., 2026b; Lamont et al., 2026; Fang et al., 2026; Gong et al., 2025; Li et al., 2026). As we make it clear in Section 3.1, the former concerns the entropy and coverage of the resulting samples, whereas the latter concerns the structure of the generation process itself.

A first line of work studies why confidence-based any-order decoding produces less diverse outputs than fixed left-to-right sampling with a nonzero token-sampling temperature. Ni et al. (2026) characterize this phenomenon as the *flexibility trap*. Since confidence-based decoding can postpone uncertain tokens and instead resolve low-entropy positions, it may bypass the branching points at which autoregressive trajectories would emerge. Motivated by a similar observation, Fang et al. (2026) describe a *quality–exploration dilemma*: greedily committing high-confidence tokens can improve the quality of an individual trajectory while suppressing sequence-level entropy and harming multi-sample exploration at Pass@K.

Several works develop inference-time methods to mitigate this diversity degradation. Olausson et al. (2026) distinguish the temperature used to sample token values from the temperature used to select or remask token positions. Lamont et al. (2026) instead coordinate multiple samples within a batch, discouraging each new trajectory from reproducing feature-space directions already explored by previous samples. Most closely related, Patel et al. (2025a) reweights a token's probability at a given position relative to a local or global cross-position reference. Their notion of *cross-position reference* is closely related to our aggregate probability mass. Whereas they leverage this signal for an inference-time algorithm, we use it as a diagnostic of positional uncertainty, making it complementary.

A second line of work more directly examines the generation trajectory itself. Gong et al. (2025) introduce local and global measures of *AR-ness*, quantifying how closely the order in which an MDM finalizes tokens follows a left-to-right trajectory. They show that this behavior depends on training and inference choices and that increasing sampling temperature can diversify not only token values but also generation orders. Gwak et al. (2025) similarly observe that independently selecting positions according to token-level confidence often yields sequential, AR-like trajectories, and propose reward-weighted sampling to favor groups of tokens that better preserve non-autoregressive generation. Li et al. (2026) attribute the tendency toward causal generation to biases inherited from the training data and propose a parallel-inference design intended to reduce this bias. This account does not fully explain code generation, where Python programs often admit multiple valid construction orders rather than a unique inherent causal order. Other methods treat generation order as an inference-time search problem. Rather than accepting the single trajectory induced by greedy confidence ranking, Shen et al. (2026b) jointly search over both which positions to decode and which token values to assign.

Our work focuses on this remaining mechanistic question: we ask why standard MDM inference, despite its *any-order interface*, fails to go meaningfully beyond AR-like reasoning in the first place. We quantify how closely confidence-based inference structurally approximates causal decoding on Python programs (Section 3.1) and attribute this collapse to a more fundamental bottleneck of token-level, fixed-canvas inference (Section 3.2). This mechanism also reveals broader limitations for tasks requiring globally structured generation.

Continuous latent reasoning in autoregressive LLMs. Reasoning in continuous representation space has recently gained attention as a way to reduce the cost of explicit chain-of-thought generation while preserving intermediate computation. Since such methods avoid verbalizing every reasoning step, they can improve efficiency, though at the cost of reduced interpretability. Coconut (Hao et al., 2024) enables LLMs to perform intermediate reasoning in a continuous hidden-state space by feeding the last hidden state back as the next input embedding, rather than decoding each reasoning step into language tokens. ContextLM (Dai et al., 2025) augments autoregressive LMs with latent context prediction, where fixed-size context embeddings guide subsequent token generation. DLCM (Qu et al., 2025) learns to parse sequences into variable-length concept segments, runs a causal Transformer over the compressed concept sequence, and decodes the resulting representations back to tokens, thereby reallocating computation around semantic boundaries. These works demonstrate the usefulness of continuous representations for efficient reasoning, but remain fundamentally autoregressive at the high-level inference interface.

Continuous generative models for language. Another line of work develops continuous generative models for text, often avoiding discrete-token realization during intermediate generation. Early approaches (Li et al., 2022; Lin et al., 2023) apply diffusion models to token-level continuous embeddings, mainly for controllable generation or sequence-to-sequence generation. TarflowLM (Zhang et al., 2025) models token-level Gaussian embeddings using Transformer-based autoregressive normalizing flows. A common issue in direct embedding-space generation is the rounding problem: predicted continuous embeddings must eventually be mapped back to discrete tokens, and this projection can introduce rounding errors.

More recent approaches (Chen et al., 2026; Lee et al., 2026) indirectly mitigate this issue by matching token posteriors rather than directly regressing word embeddings. This is especially explicit in FMLM (Lee et al., 2026), which operates in one-hot token space and decodes by taking the argmax over predicted token probabilities.

This flow-map approach to few-step generation of discrete data has advanced rapidly. Categorical Flow Maps (Roos et al., 2026) learn maps toward the probability simplex using distillation and endpoint-consistency objectives, while Discrete Flow Maps (Potaptchik et al., 2026) modify flow-map training to better reflect the discrete structure of language and support full-sequence generation in a single model evaluation. Building on FMLM, Yoo et al. (2026) interpret self-conditioning in flow-based language models as a fixed-point iteration and use this perspective to improve few-step generation. Other works replace one-hot token representations with more compact continuous parameterizations: the hyperspherical flow language model ($\mathbb{S}$-FLM) (Deschenaux & Gulcehre, 2026) runs latent flows on the unit hypersphere and narrows the gap to MDMs on mathematics and code (we adopt it as a representative continuous-flow baseline in Section 4.2); ELF (Hu et al., 2026) performs flow matching in word-embedding space and converts the resulting continuous states to token predictions through a learned output map; and CoBit (Batzolis et al., 2026), applies diffusion to analog bitstream representations of tokens, reducing the dimensional dependence of each token prediction from linear to logarithmic in the vocabulary size. Closest to our focus, Agarwal et al. (2026) equip flow-map language models with masking-style noise schedules, yielding an any-order continuous-space sampler with an inference-time self-correction mechanism (posterior refinement); their goal, however, is to improve the speed–quality trade-off of parallel generation rather than to analyze the structure of the generation orders that the any-order interface induces.

A complementary way to avoid token-level rounding is to separate high-level continuous planning from discrete token realization. LD4PG (Lovelace et al., 2023), PLANNER (Zhang et al., 2023), Segment-Level Diffusion (Zhu et al., 2025), VDLM (Qu, 2026), and CoDAR (Shen et al., 2026a) use continuous diffusion models over higher-level semantic latents (i.e., sentence-, paragraph-, or whole-sequence-level representations), and rely on an autoregressive token-level decoder or renderer to iteratively decode each denoised latent back into discrete text in the final stage of generation. COSMOS (Meshchaninov et al., 2025) similarly constructs a compressed continuous latent space for text diffusion, but decodes tokens in parallel. LCM (Barrault et al., 2024) and LaDiR (Kang et al., 2025) both use diffusion over continuous high-level semantic representations beyond tokens while performing autoregressive generation at the semantic level: LCM denoises sentence-level concept embeddings for semantic continuation, whereas LaDiR denoises structured latent chain-of-thought blocks for reasoning.

Related hybrid continuous–discrete diffusion methods, including CADD (Zheng et al., 2025), CANDI (Pynadath et al., 2025), and CCDD (Zhou et al., 2025), combine continuous and discrete diffusion to reduce rounding mismatch, though the continuous variables differ across methods, including word embeddings, one-hot vectors, and LLM-contextualized latents.

Our LatentMDM shares the broad motivation of using continuous latent variables to represent higher-level semantic units. However, our focus is different: we study whether moving the masked-diffusion interface from token positions to segment-level latents can elicit genuinely any-order inference. Most prior continuous or latent-space language models primarily target likelihood modeling, controllability, or fast generation, and typically do not analyze whether their generation traces differ structurally from ARM or token-level MDM decoding.

Hybrid autoregressive and masked-diffusion. Our LatentMDM also relates to hybrid architectures that combine non-autoregressive (NAR) prediction with autoregressive token generation. TiDAR (Liu et al., 2025a) uses a token-level MDM as a parallel drafter and an ARM as a verifier or sampler

within a unified module. ReFusion (Li et al., 2025) fine-tunes an ARM for block-level any-order generation while preserving token-level autoregressivity, enabling efficient inference with KV caching. CoDiLA (Hersche et al., 2026) uses a token-level MDM as a parallel drafter and a smaller local ARM as a conditional sampler given soft draft conditions. These approaches are promising for improving sampling efficiency, but their any-order structure is generally defined over fixed token blocks while preserving substantial token-level autoregressive-ness. In contrast, our LatentMDM moves the reveal decision to variable-length semantic segments and asks whether segment-level confidence can induce generation orders that escape the left-to-right bias observed in token-level MDMs.

Any-order models in continuous latent spaces. MAR (Li et al., 2024) and Diffusion Forcing (Song et al., 2025a; Chen et al., 2024) propose any-order interfaces for denoising-based diffusion models in continuous latent spaces. Their use of this flexibility, however, is mainly aimed at improving generation efficiency (Li et al., 2024) or enabling conditional and long-horizon video generation (Song et al., 2025a; Chen et al., 2024), rather than searching over semantic generation orders.

The works with the greatest conceptual overlap with ours are those of Wewer et al. (2025); Schusterbauer et al. (2026). They train diffusion-forcing-style, any-order continuous diffusion models in which each patch or frame can be assigned a separate time step, enabling flexible generation orders. Although our work shares this high-level conceptual connection, it differs substantially in formulation: these works focus on continuous-space diffusion modeling, whereas we adopt a masked-modeling formulation.

Insertion-based discrete generative models. Along with FlexMDM (Kim et al., 2025a), a growing body of work has sought to develop discrete generative models that support insertion (Havasi et al., 2025; Tang et al., 2026; Patel et al., 2025b; 2026; Zhang et al., 2026). We leave comparisons with these models, as well as investigating how they could support any-order inference, for future work.

B Details of Section 3.1

This appendix describes the code-similarity metrics used in Section 3.1. Each metric converts a generated Python code into a rooted ordered tree and then compares two such trees by tree edit distance. A tree node represents a syntactic component of the program, and an edge represents immediate containment: a parent node is a larger construct, and its children are the constructs that appear inside it. Children are ordered by their order in the source code.

For two rooted ordered labeled trees $\mathcal{T}$ and $\mathcal{T}'$, let $\mathrm{TED}_\rho(\mathcal{T}, \mathcal{T}')$ denote the tree edit distance between them, computed with unit insertion cost, unit deletion cost, and rename cost ρ. We normalize the distance into a similarity score:

$$\mathrm{Sim}_\rho(\mathcal{T}, \mathcal{T}') = 1 - \frac{\mathrm{TED}_\rho(\mathcal{T}, \mathcal{T}')}{\max\{|\mathcal{T}|, |\mathcal{T}'|\}},$$

where $|\mathcal{T}|$ is the number of nodes in $\mathcal{T}$. Scores are clipped to $[0, 1]$ when necessary. Larger values indicate a more similar tree structure. The three metrics below differ only in how they construct the tree before applying this normalized tree-edit similarity.

Abstract Syntax Tree Distance (ASTD). Our main metric, ASTD, uses Python's built-in parser to convert each program into an abstract syntax tree. In this tree, nodes correspond to semantic syntactic units used by the Python interpreter, such as `Module`, `FunctionDef`, `For`, `If`, `Assign`, `Return`, `Name`, `Constant`, and `Add`. The parent-child relation is given by Python's AST fields: for example, a `FunctionDef` contains its arguments and body statements, an `If` contains its condition and branches, and an `Assign` contains its target and value.

For example, for the code

```
def f(x):
  y = x + 1
  return y
```

the AST-based representation contains nodes such as `Module`, `FunctionDef`, `arguments`, `arg`, `Assign`, `Name`, `BinOp`, `Add`, `Constant`, and `Return`. The `FunctionDef` node contains the argument list and the two body statements; the `Assign` node contains the target `Name` and the value `BinOp`; and the `BinOp` node contains the left operand, operator, and right operand.

Before comparing ASTs, we anonymize identifiers so that arbitrary names do not dominate the score. Variable and argument names are replaced by canonical placeholders, function names are canonicalized, and top-level docstrings are ignored. The resulting tree still preserves the program's control flow, statement structure, operators, literals, and expression nesting.

Let $\mathcal{A}(c)$ be this anonymized Python AST for code c. We define

$$\mathrm{ASTD}(c, c') = \mathrm{Sim}_1\big(\mathcal{A}(c), \mathcal{A}(c')\big).$$

The rename cost is $\rho = 1$, so changing a node label, such as replacing `Add` with `Sub` or `Constant(1)` with `Constant(2)`, contributes edit cost. Therefore, ASTD measures both the global program skeleton and local syntactic choices inside that skeleton.

Tree Similarity of Edit Distance (TSED). TSED uses the same tree-edit-distance normalization, but builds the tree with `tree-sitter` rather than Python's AST parser. Python's AST is an interpreter-level abstraction, whereas tree-sitter follows the surface grammar of the written program. Its nodes correspond to grammar constructs such as `function_definition`, `parameters`, `block`, `assignment`, `return_statement`, and `call`. Edges again mean immediate containment in the parsed program, and children are ordered by source order.

For the same example above, the tree-sitter representation contains grammar nodes such as `module`, `function_definition`, `identifier`, `parameters`, `block`, `expression_statement`, `assignment`, `binary_operator`, and `return_statement`. Compared with the Python AST, this tree follows the written grammar more directly: for instance, the function body appears under a `block` node, and the assignment may appear under an `expression_statement` node.

Let $\mathcal{G}(c)$ be the tree-sitter parse tree of code c. We define

$$\mathrm{TSED}(c, c') = \mathrm{Sim}_0\big(\mathcal{G}(c), \mathcal{G}(c')\big).$$

Here the rename cost is $\rho = 0$. Thus, TSED does not charge for changing a node label; it mainly charges for inserting or deleting nodes to make the two parse trees align. In other words, ASTD asks whether the two ASTs have matching node types and values, while TSED asks a weaker question: whether the two programs have similarly shaped parse trees. This makes TSED a more permissive structural metric than ASTD.

Coarse ASTD (Coarse). Coarse ASTD is a lower-resolution variant of ASTD. It starts from the same anonymized Python AST, but keeps only statement-level structure. Nodes such as `FunctionDef`, `For`, `While`, `If`, `Assign`, and `Return` are retained, while fine-grained expression subtrees are collapsed into the label of the nearest statement node.

For the same example, Coarse ASTD keeps a much smaller tree: a root `Module` node, a `FunctionDef` node, and statement-level children such as `Assign` and `Return`. The internal expression `x + 1` is not expanded into separate `Name`, `Add`, and `Constant` nodes. Instead, it is treated as part of the assignment's statement-level label. Thus, Coarse ASTD ignores many expression-level differences and focuses on the high-level statement layout.

Let $\mathcal{C}(c)$ be this coarsened statement-level tree. We define

$$\mathrm{Coarse}(c, c') = \mathrm{Sim}_1\big(\mathcal{C}(c), \mathcal{C}(c')\big).$$

The rename cost is again $\rho = 1$, so mismatched statement labels contribute to the edit cost. However, because expression internals are not expanded into separate nodes, Coarse ASTD focuses more directly on high-level control flow and statement layout. It tests whether two decoding strategies produce programs with similar global skeletons even after ignoring much of the expression-level detail.

Sampling protocol. For each model and prompt, all decoding policies use temperature 0.2 and nucleus sampling top-$p = 0.95$, as in Gong et al. (2025). We generate one sample for each confidence-based policy, since confidence-based decoding at this temperature produces little diversity across repeated samples. We generate 32 samples for strict L2R and random decoding. The 32 L2R samples form the reference set used in best-match aggregation, while the 32 random samples provide a calibration baseline and help obtain enough valid or correct programs despite the lower success rates of random and L2R decoding.

Best-match aggregation over samples. The metrics above define the similarity between a pair of code snippets. In our experiments, each sample from an evaluated decoding strategy is compared against a set of L2R samples for the same prompt. We therefore aggregate pairwise similarities using a best-match rule. For a prompt q, let $\mathcal{D}_q^{\pi}$ be the samples generated by the decoding strategy being evaluated, and let $\mathcal{D}_q^{\mathrm{L2R}}$ be the reference L2R samples. For a chosen similarity metric $s(\cdot,\cdot)$, we compute

$$\mathrm{BM}_q(s) = \frac{1}{|\mathcal{D}_q^{\pi}|} \sum_{y \in \mathcal{D}_q^{\pi}} \max_{x \in \mathcal{D}_q^{\mathrm{L2R}}} s(y, x).$$

That is, each sample from the evaluated decoding strategy is matched to its most similar L2R sample, and we average these best-match scores. We then report the mean of $\mathrm{BM}_q(s)$ across prompts.

We apply the same aggregation separately under the valid–valid and success–success filters. For valid–valid, both y and its candidate matches x must be syntactically valid. For success–success, both must pass the unit tests. This aggregation asks whether each generated program has a close L2R counterpart.

B.1 Additional similarity results

Tables 1 and 2 report the L2R comparison results for DiffuCoder-base, DiffuCoder-instruct, DiffuCoder-cpGRPO, Dream-Base, Dream-instruct, LLaDA-instruct, LLaDA-cpGRPO, and `dUltra-coding-b128` on HumanEval and MBPP. Table 3 reports the comparison results for Dream-Coder-instruct on LiveCodeBench. We report TSED, ASTD, and Coarse ASTD under two filtering regimes: valid–valid pairs, where both codes are syntactically valid, and correct–correct pairs, where both codes pass the unit tests.

Across all models and both datasets, Random vs L2R consistently yields much lower similarity than Entropy, Top-K Margin, and Top-K vs L2R. This trend holds not only for syntactically valid codes, but also for correct codes. In other words, random decoding does not merely produce structurally different incorrect code. Even among successful generations, structures are meaningfully less similar to L2R outputs than those produced by confidence-based decoding. This supports our claim that confidence-based decoding remains structurally close to L2R, whereas random reveal orders provide a substantially more non-left-to-right reference.

B.2 Semantic collapse in dUltra planner

Experimental setup. `dUltra-coding-b128` (Chen et al., 2025) uses LLaDA as the base masked diffusion language model and augments it with an unmasking planner head. The base LLaDA model predicts the token values, whereas the planner head predicts which masked positions to reveal. The planner is first pretrained with the frozen LLaDA backbone, and then the base model and the planner are jointly optimized with GRPO. The goal is to learn an efficient reveal policy that selects weakly dependent masked positions for parallel unmasking while preserving the token-prediction quality.

Our goal is to test whether the learned dUltra unmasking planner produces structurally different outputs from standard decoding policies. We therefore apply the same similarity analysis as in Section 3.1 and compare each policy against strict L2R decoding under a Pass@64 evaluation setup. For a controlled comparison, all policies follow the inference configuration of `dUltra-coding-b128`, using the same LLaDA backbone, prompt formatting, maximum generation length, diffusion step schedule, block size, temperature, candidate sampling budget, and evaluation pipeline. For the planner-based policy, we use the trained dUltra unmasking head to select masked positions for reveal and keep the planner decoding temperature at its default value, 0. The planner-based sampling scheme still introduces stochasticity through its learned reveal-position selection procedure. For planner-free baselines, we disable the planner head and decode the same LLaDA token predictor with hand-designed reveal policies: Top-K, Top-K margin, entropy-based selection, random selection, and strict L2R. In these backbone-only baselines, we use a temperature 0.2 following the standard sampling setup. Thus, apart from this policy-specific temperature convention, the comparison controls for the backbone, prompts, decoding budget, and evaluation pipeline, and isolates how the reveal-position selection rule affects the resulting code structure. The dUltra experiments follow a Pass@64 protocol and compare all policies against the L2R baseline.

Table 1: HumanEval code-similarity to strict L2R baselines using best-match aggregation. Dream, DiffuCoder, LLaDA-instruct, and LLaDA-cpGRPO results are compared against L2R under the Pass@32 setup, while `dUltra-coding-b128` is evaluated under its Pass@64 setup. Valid pairs require both samples to parse; correct pairs require both samples to pass unit tests.

		Valid pairs			Correct pairs		
Model	Comparison	TSED	ASTD	Coarse	TSED	ASTD	Coarse
DiffuCoder-base	Entropy vs L2R	0.9994	0.9995	0.9994	1.0000	1.0000	1.0000
	Top-K Margin vs L2R	0.8890	0.8427	0.8527	0.9070	0.8627	0.8670
	Top-K vs L2R	0.8816	0.8320	0.8466	0.9070	0.8652	0.8708
	Random vs L2R	0.6857	0.5583	0.5963	0.8108	0.7077	0.7087
DiffuCoder-instruct	Entropy vs L2R	0.9637	0.9458	0.9424	0.9741	0.9628	0.9647
	Top-K Margin vs L2R	0.9128	0.8739	0.8801	0.9381	0.9126	0.9156
	Top-K vs L2R	0.9128	0.8739	0.8801	0.9381	0.9126	0.9156
	Random vs L2R	0.7070	0.5981	0.6494	0.7897	0.6902	0.7090
DiffuCoder-cpGRPO	Entropy vs L2R	0.9795	0.9706	0.9791	0.9883	0.9794	0.9801
	Top-K Margin vs L2R	0.9795	0.9706	0.9791	0.9883	0.9794	0.9801
	Top-K vs L2R	0.9795	0.9706	0.9791	0.9883	0.9794	0.9801
	Random vs L2R	0.8049	0.7245	0.7813	0.8592	0.7909	0.8173
Dream-base	Entropy vs L2R	0.8329	0.7717	0.7831	0.9047	0.8649	0.8691
	Top-K Margin vs L2R	0.8329	0.7715	0.7823	0.9035	0.8632	0.8673
	Top-K vs L2R	0.8329	0.7717	0.7831	0.9047	0.8649	0.8691
	Random vs L2R	0.5973	0.4578	0.5277	0.7325	0.6328	0.6482
Dream-instruct	Entropy vs L2R	0.9074	0.8706	0.8769	0.9714	0.9585	0.9554
	Top-K Margin vs L2R	0.9074	0.8706	0.8769	0.9714	0.9585	0.9554
	Top-K vs L2R	0.8913	0.8481	0.8576	0.9556	0.9366	0.9235
	Random vs L2R	0.6823	0.5626	0.6227	0.7881	0.7001	0.7288
LLaDA-instruct	Entropy vs L2R	0.9758	0.9470	0.9508	0.9718	0.9594	0.9555
	Top-K Margin vs L2R	0.9017	0.8494	0.8709	0.9151	0.8737	0.8959
	Top-K vs L2R	0.9246	0.8843	0.8984	0.9507	0.9323	0.9374
	Random vs L2R	0.7672	0.6611	0.6947	0.8595	0.7958	0.8097
LLaDA-cpGRPO	Entropy vs L2R	0.8927	0.8434	0.8437	0.9145	0.8739	0.8757
	Top-K Margin vs L2R	0.8780	0.8226	0.8277	0.9192	0.8802	0.8798
	Top-K vs L2R	0.8940	0.8416	0.8481	0.9287	0.8917	0.8878
	Random vs L2R	0.7306	0.6377	0.6451	0.8107	0.7355	0.7086
dUltra-coding-b128	Planner vs L2R	0.9632	0.8962	0.8993	0.9761	0.9689	0.9648
	Entropy vs L2R	1.0000	0.9295	0.9295	1.0000	1.0000	1.0000
	Top-K Margin vs L2R	0.9220	0.8386	0.8431	0.9358	0.9095	0.8918
	Top-K vs L2R	0.9261	0.8416	0.8460	0.9499	0.9272	0.9195
	Random vs L2R	0.6525	0.3085	0.3505	0.8966	0.8605	0.8822

Results. Tables 1 and 2 show that the learned dUltra planner produces outputs that remain highly similar to strict L2R decoding. On both HumanEval and MBPP, planner decoding has substantially higher valid-pair similarity to L2R than the original random-style decoding policy, indicating that the learned planner does not induce a structurally diverse any-order generation pattern. Moreover, the planner is more L2R-like than Top-K and Top-K margin on most similarity metrics. This suggests that explicitly learning an unmasking schedule is not sufficient to avoid semantic collapse, and the planner can still converge to reveal orders whose final code structures closely match those produced by L2R-style decoding.

C Details of Section 3.2

Experimental setup. We evaluate positional uncertainty on large-scale masked diffusion language models for code generation: Dream-7B-Base (Xie et al., 2025), and DiffuCoder-7B-Base, DiffuCoder-7B-Instruct, DiffuCoder-7B-cpGRPO (Gong et al., 2025), and LLaDA-8B-instruct (Nie et al., 2025). We use problem prompts from HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021).

Table 2: MBPP code-similarity to strict L2R baselines using best-match aggregation. Dream, DiffuCoder, LLaDA-instruct, and LLaDA-cpGRPO results are compared against L2R under the Pass@32 setup, while `dUltra-coding-b128` is evaluated under its Pass@64 setup. Valid pairs require both samples to parse; correct pairs require both samples to pass unit tests.

		Valid pairs			Correct pairs		
Model	Comparison	TSED	ASTD	Coarse	TSED	ASTD	Coarse
DiffuCoder-base	Entropy vs L2R	0.9960	0.9931	0.9915	0.9992	0.9987	0.9982
	Top-K Margin vs L2R	0.8717	0.8216	0.8781	0.9099	0.8668	0.9036
	Top-K vs L2R	0.8697	0.8190	0.8764	0.9116	0.8670	0.9019
	Random vs L2R	0.7107	0.5842	0.6916	0.7822	0.6856	0.7597
DiffuCoder-instruct	Entropy vs L2R	0.8552	0.8055	0.8306	0.8932	0.8539	0.8638
	Top-K Margin vs L2R	0.8552	0.8055	0.8306	0.8932	0.8539	0.8638
	Top-K vs L2R	0.8552	0.8055	0.8306	0.8932	0.8539	0.8638
	Random vs L2R	0.7445	0.6494	0.7363	0.8128	0.7421	0.7903
DiffuCoder-cpGRPO	Entropy vs L2R	0.9851	0.9808	0.9877	0.9916	0.9872	0.9952
	Top-K Margin vs L2R	0.9851	0.9808	0.9877	0.9916	0.9872	0.9952
	Top-K vs L2R	0.9851	0.9808	0.9877	0.9916	0.9872	0.9952
	Random vs L2R	0.8337	0.7676	0.8470	0.8592	0.8093	0.8717
Dream-base	Entropy vs L2R	0.8941	0.8406	0.8802	0.9276	0.8933	0.9236
	Top-K Margin vs L2R	0.8950	0.8421	0.8802	0.9289	0.8952	0.9250
	Top-K vs L2R	0.8950	0.8421	0.8802	0.9289	0.8952	0.9250
	Random vs L2R	0.6761	0.5537	0.6819	0.7870	0.7004	0.7739
Dream-instruct	Entropy vs L2R	0.9441	0.9211	0.9235	0.9656	0.9519	0.9417
	Top-K Margin vs L2R	0.9441	0.9211	0.9235	0.9656	0.9519	0.9417
	Top-K vs L2R	0.9441	0.9211	0.9235	0.9656	0.9519	0.9417
	Random vs L2R	0.7593	0.6683	0.7545	0.8383	0.7774	0.8359
LLaDA-instruct	Entropy vs L2R	0.9757	0.9573	0.9602	0.9760	0.9653	0.9661
	Top-K Margin vs L2R	0.8971	0.8407	0.8587	0.9328	0.8934	0.9035
	Top-K vs L2R	0.8991	0.8488	0.8595	0.9284	0.8981	0.9092
	Random vs L2R	0.7421	0.6301	0.6771	0.8286	0.7479	0.7838
LLaDA-cpGRPO	Entropy vs L2R	0.9033	0.8541	0.8677	0.9320	0.9023	0.9099
	Top-K Margin vs L2R	0.8902	0.8341	0.8517	0.9257	0.8915	0.9010
	Top-K vs L2R	0.9033	0.8541	0.8721	0.9417	0.9123	0.9205
	Random vs L2R	0.7435	0.6399	0.6921	0.8227	0.7417	0.7636
dUltra-coding-b128	Planner vs L2R	0.9731	0.9343	0.9401	0.9772	0.9677	0.9710
	Entropy vs L2R	1.0000	0.9552	0.9552	1.0000	1.0000	1.0000
	Top-K Margin vs L2R	0.9215	0.8513	0.8680	0.9461	0.9198	0.9272
	Top-K vs L2R	0.9208	0.8483	0.8661	0.9457	0.9199	0.9294
	Random vs L2R	0.5414	0.2646	0.3054	0.8603	0.8057	0.8356

For Dream and DiffuCoder, the logged runs use a 256-token canvas and 256 diffusion steps with temperature 0.2 and top-$p = 0.95$. LLaDA runs use a 512-token canvas and 512 denoising steps. For the released bundle visualizations, we export point data on the full HumanEval-164 and MBPP-500 task sets and render standalone plots at a step stride of 16.

All generated sequences include non-code text around them, including an instruction/prompt prefix before the code region and a natural language explanation after the code. We therefore extract the code span before computing positional-uncertainty statistics. Since generation is performed at the token level, the generated sequence can be truncated in the middle of a code region or continue beyond the end of the program into natural language text. In such cases, we discard tokens outside the extracted code span and compute all quantities only on the retained in-code token positions. Thus, the masked positions appearing in the definitions of $m(v; \mathbf{x}_t)$ and $\mathrm{LOC}(v; \mathbf{x}_t)$ are restricted to this trimmed code domain.

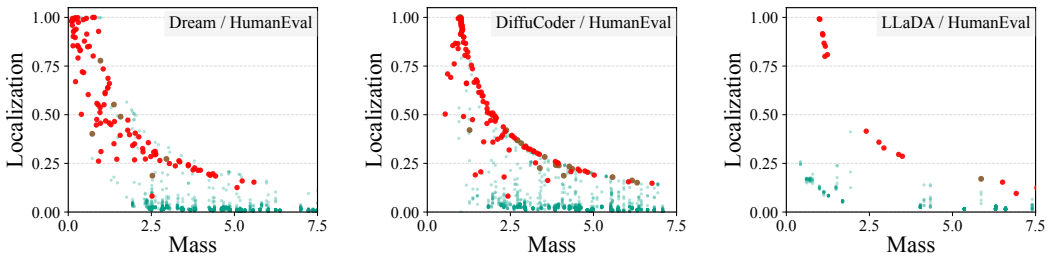
Because storing the full vocabulary distribution at every step is too expensive, we do not log the full per-position vocabulary posterior. Instead, at each analyzed step, we retain up to 200 selected

Table 3: Dream-Coder-instruct's code-similarity to the L2R baseline on LiveCodeBench v5 and v6, using best-match aggregation. For Dream-Coder, the entropy, top-K margin, and top-K generations on LCB are identical to each other under temperature 0.2; we thus report all three as "confidence vs L2R" instead of repeating three times.

		Valid pairs			Correct pairs		
Benchmark	Comparison	TSED	ASTD	Coarse	TSED	ASTD	Coarse
LCB-v5	Confidence vs L2R	0.7339	0.5960	0.5358	0.8488	0.7598	0.7030
	Random vs L2R	0.6032	0.4075	0.3681	0.7324	0.6098	0.6025
LCB-v6	Confidence vs L2R	0.7272	0.5866	0.5261	0.8488	0.7624	0.7111
	Random vs L2R	0.6000	0.4065	0.3632	0.7431	0.6254	0.6126

token IDs, chosen primarily by aggregate token mass, and we store their positional mass profiles over the trimmed code domain. Committed token IDs are additionally forced into the logged set when necessary, and committed-local statistics are recorded separately so that committed-position uncertainty can still be computed even when a committed token would otherwise fall outside the generic 200-token selection. Unless otherwise specified, the positional-uncertainty results in this subsection are reported under a Pass@32 setup against the strict L2R baseline.

Additional Results. Figure 6 shows representative positional-uncertainty plots across model families and benchmarks. Since Dream and DiffuCoder use a 256-step denoising schedule, whereas LLaDA uses a 512-step schedule, we visualize step 16 for Dream and DiffuCoder and step 32 for LLaDA. These choices correspond to the same relative denoising progress, i.e., $16/256 = 32/512$. Across both HumanEval and MBPP, we observe the same qualitative pattern as in Section 3.2: many high-mass tokens remain poorly localized, while committed tokens are biased toward higher localization. This supports that positional uncertainty is not specific to a single model family or benchmark.



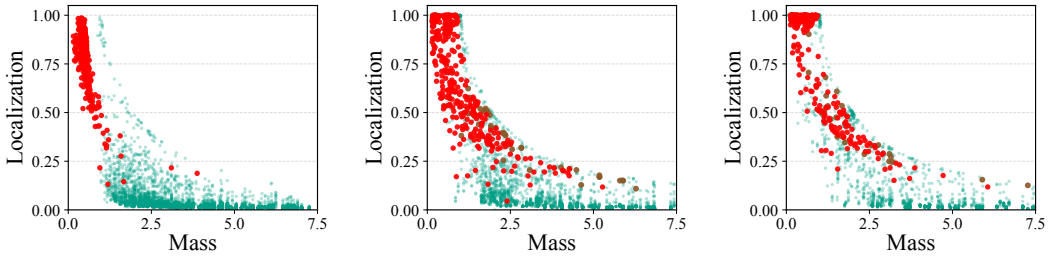
(a) Dream, HumanEval, step 16 (b) DiffuCoder, HumanEval, step 16 (c) LLaDA, HumanEval, step 32

(d) Dream, MBPP, step 16 (e) DiffuCoder, MBPP, step 16 (f) LLaDA, MBPP, step 32

Figure 6: Additional positional-uncertainty visualizations across model families and benchmarks. Dream and DiffuCoder are visualized at step 16 out of 256 denoising steps, while LLaDA is visualized at step 32 out of 512 denoising steps. Across settings, high aggregate token mass does not necessarily imply high localization, and committed tokens tend to be more localized.

We further visualize how positional uncertainty evolves across denoising steps for DiffuCoder on MBPP. Figure 7 shows the mass-localization plots at steps $0, 16, 32, 48, 64$, and 80. Across the trajectory, high-mass tokens can remain broadly dispersed over candidate positions, while committed

tokens tend to concentrate in regions with higher localization. This step-wise view illustrates that positional uncertainty persists over a substantial portion of the inference trajectory.



(a) Step 0 (b) Step 16 (c) Step 32

(d) Step 48 (e) Step 64 (f) Step 80

Figure 7: Step-wise positional-uncertainty visualization for DiffuCoder on MBPP. Each panel plots aggregate token mass against localization at a different denoising step. Across the trajectory, high-mass tokens are not necessarily localized, while committed tokens tend to have higher localization.

D Details of Section 4.1

In this appendix, we provide the theoretical foundation of FlexMDM, details on tree metrics for quantifying any-order inference, and an evaluation pipeline.

D.1 FlexMDM Training

Preliminary theory. Let α_t, β_t be smooth and monotone schedules with $\alpha_0 = \beta_0 = 0$ and $\alpha_1 = \beta_1 = 1$. They will be the insertion and unmasking schedules, respectively. Given a clean sequence $\mathbf{x}_1 = (\mathbf{x}_1^1, ..., \mathbf{x}_1^L) \sim p_{\text{data}}$ of length L, for each position $i \in [L]$, define its insertion and unmasking times T_1^i, T_2^i as

$$T_1^i \sim \dot{\alpha}_t dt, \quad T_2^i \sim 1_{t \geq T_1^i} \cdot \frac{\dot{\beta}_t}{1 - \beta_{T_1^i}} dt.$$

For $t \in [0, 1]$, define the (sorted) set s_t of inserted indices as

$$s_t = \{i \in [L] \mid t \geq T_1^i\}$$

with $s_t[1] < s_t[2] < \cdots < s_t[\text{len}(s_t)]$, and with the boundary convention that $s_t[0] = 0$ and $s_t[\text{len}(s_t) + 1] = L + 1$. Intuitively, this is the set of indices that have been inserted by time t. The stochastic interpolant $\mathbf{x}_t$, a sequence of length $|s_t|$, is then constructed by the following masking and deletion procedure:

$$\mathbf{x}_t^i = \begin{cases} \mathbf{m} & \text{if } t < T_2^{s_t[i]}, \\ \mathbf{x}_1^{s_t[i]} & \text{if } t \geq T_2^{s_t[i]}. \end{cases}$$

Similar to a standard MDM, a FlexMDM parametrizes an unmasking posterior $f_\theta(\mathbf{x}, t)[i, v] \approx \mathbb{P}[\mathbf{x}_1^{s_t[i]} = v \mid \mathbf{x}_t = \mathbf{x}]$; however, it also parametrizes an insertion prediction

$$g_\theta(\mathbf{x}, t)[i] \approx \log \mathbb{E}[s_t[i] - s_t[i-1] - 1 \mid \mathbf{x}_t = \mathbf{x}].$$

Intuitively, this is the log of the expected number of tokens to be inserted between the i-th and $(i-1)$-th positions in the current $\mathbf{x}_t$. At training time, we minimize the following loss function:

$$\mathcal{L}(\theta) = \underbrace{\int_0^1 \mathbb{E}_{\mathbf{x}_1, s_t, \mathbf{x}_t} \left[-\frac{\dot{\beta}_t}{1-\beta_t} \sum_{i=1}^{\text{len}(\mathbf{x}_t)+1} \mathbf{1}_{\mathbf{x}_t^i = \mathbf{m}} \log f_\theta(\mathbf{x}_t, t)[i, \mathbf{x}_1^{s_t[i]}] \right] dt}_{\text{unmasking loss}} + \underbrace{\int_0^1 \mathbb{E}_{\mathbf{x}_1, s_t, \mathbf{x}_t} \left[-\frac{\dot{\alpha}_t}{1-\alpha_t} \sum_{i=1}^{\text{len}(\mathbf{x}_1)+1} \phi(s_t[i] - s_t[i-1] - 1,\, g_\theta(\mathbf{x}_t, t)[i]) \right] dt}_{\text{insertion loss}}, \tag{1}$$

where $\phi(x, y) = e^y - xy$.

Notably, this differs from the original setup in Kim et al. (2025a), where the insertion head directly predicts the expected insertion length, whereas we found that empirically predicting in log space yields stabler training. Geometrically, insertion counts live on a positive, multiplicative scale and enter inference as Poisson intensities, so the log map converts variation in expected lengths into an unconstrained additive coordinate. This makes the insertion head predict the natural parameter of the Poisson rate, which better matches a neural network's real-valued outputs and avoids the boundary/scale issues of directly regressing a nonnegative expectation.

Implementation details. We choose the following insertion and unmasking schedules:

$$\alpha_t = 1 - (1-t)^a, \qquad \beta_t = 1 - (1-t)^{ab}, \tag{2}$$

where $a = b = 1.7$. We explicitly note that this parameterization is part of a more general class of schedules considered in prior work (Patel et al., 2026).

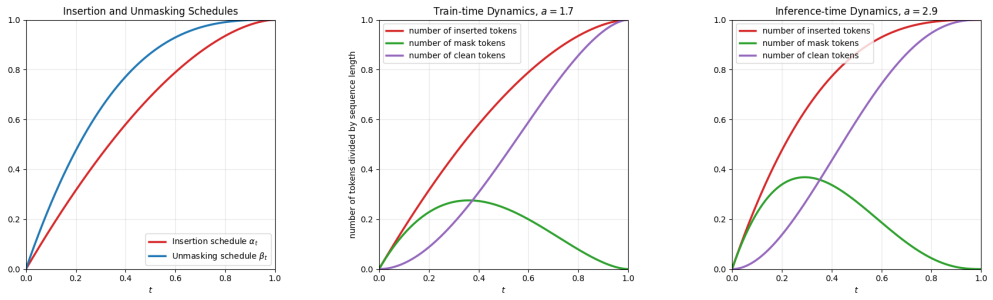


Figure 8: **(Left)** graphs of the insertion and unmasking schedules in (2). **(Center)** the number of inserted, mask, and clean tokens as a function of time during training, normalized by sequence length. **(Right)** the number of inserted, mask, and clean tokens as a function of time during inference, normalized by sequence length.

An important practical observation beyond the original FlexMDM formulation in Kim et al. (2025a) is that, at inference time, we are not locked to the insertion schedule used during training. Let α_t denote the insertion schedule used during training, and let $\tilde{\alpha}_t$ denote a desired insertion schedule at inference time. We define the monotone time reparameterization

$$\tau(t) := \alpha^{-1}(\tilde{\alpha}_t).$$

At inference step t, we query the model at the reparameterized time $\tau(t)$, i.e., we use the predictions $f_\theta(x, \tau(t))$ and $g_\theta(x, \tau(t))$, while evolving the insertion process with the inference-time schedule $\tilde{\alpha}_t$. In this way, the model is always evaluated at the training-time corruption level whose insertion fraction matches the desired inference-time insertion fraction, since

$$\alpha_{\tau(t)} = \tilde{\alpha}_t.$$

Equivalently, the insertion Poisson intensity at gap i is taken to be

$$\frac{\dot{\tilde{\alpha}}_t}{1-\tilde{\alpha}_t} \exp\big(g_\theta(x, \tau(t))[i]\big),$$

while the unmasking process continues to use the original unmasking schedule β_t.

For the power-family schedules in (2), this reparameterization is even more explicit: if the model is trained with $\alpha_t = 1 - (1-t)^a$ but we want to sample with $\tilde{\alpha}_t = 1 - (1-t)^{\tilde{a}}$, then

$$\tau(t) = \alpha^{-1}(\tilde{\alpha}_t) = 1 - (1-t)^{\tilde{a}/a}.$$

This flexibility is practically important. We found that any-order inference requires a sufficiently strong scaffold to be inserted early in the trajectory; otherwise, each currently visible position must summarize too broad a region of the eventual sequence, and token predictions become clogged amalgamations of several plausible future locations. In our experiments, we therefore use a more aggressive insertion schedule at inference time with $\tilde{a} = 2.9$, while keeping the same unmasking schedule. This front-loads scaffold construction *without* retraining the model parameters.

The unmasking and insertion dynamics are visualized in Figure 8. While we insert tokens more aggressively at the early stage of inference, the number of clean tokens remains sub-linear in this early phase, avoiding premature token commitments while the layout is still uncertain; we also retain a long tail of later insertions so the model can still add missing local details after partial content has been revealed, preserving the main flexibility that mitigates positional uncertainty.

We obtain the training dataset by processing open-source Python data from OpenCodeInstruct (Ahmad et al., 2025), rStar-Coder (Liu et al., 2025b), KodCode-V1-SFT-4o (Xu et al., 2025), and opc-sft-stage2-educational (Huang et al., 2025a), resulting in ≈2.6M training sequences. For OpenCodeInstruct, we keep only those samples whose `average_test_score` is at least 0.9 and from which valid Python code is extracted, resulting in 1.85M samples out of the total 5M. From rStar-Coder, we use the `seed_sft` and `synthetic_sft` splits, keeping only the samples verified as passing their tests, which yields 81K and 369K samples respectively. We use the entirety of KodCode-V1-SFT-4o (209K samples) and opc-sft-stage2-educational (118K samples) without additional filtering. During training, we up-sample rStar-Coder, KodCode-V1-SFT-4o, and opc-sft-stage2-educational by a factor of two relative to OpenCodeInstruct.

We initialize from Dream-Coder 7B (Xie et al., 2025), while attaching an auxiliary head and an AdaLN time embedding (Peebles & Xie, 2023) for insertion prediction. We use the AdamW (Loshchilov & Hutter, 2019) optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.95$, and weight decay 0.01. We use a learning rate schedule with linear warmup and cosine decay, with warmup ratio 0.1 and peak learning rate of 10^{-5} for the backbone and 2×10^{-5} for the insertion head. We train with the loss in (1) for 50000 optimizer steps, with global batch size 576, which took 3 days on 16 H100's.

D.2 Training FlexMDM in Insertion-Progress Coordinates

The standard FlexMDM construction conditions the model on a raw time variable $t \in [0,1]$, together with an insertion schedule α_t and an unmasking schedule β_t. We instead propose to condition the model directly on the *insertion progress*

$$\tau := \alpha_t \in [0,1].$$

This removes the dependence of the model interface on the particular parameterization of time. Raw time t is then used only as an external sampler clock, while the network always receives the schedule-invariant coordinate τ.

Let $\alpha : [0,1] \to [0,1]$ be a strictly increasing insertion schedule and let $\beta : [0,1] \to [0,1]$ be the unmasking schedule used in the original raw-time formulation. Define the corresponding unmasking schedule in insertion-progress coordinates by

$$B(\tau) := \beta\left(\alpha^{-1}(\tau)\right).$$

Thus, after changing variables from t to τ, insertion progress is the identity schedule, while unmasking is governed by B.

For a clean sequence x_1, let x_τ denote the partially corrupted sequence at insertion progress τ, and let s_τ be the alignment map from positions in x_τ to positions in x_1. The model is trained as

$$f_\theta(x_\tau, \tau)[i, v] \approx \mathbb{P}\left(x^1_{s_\tau[i]} = v \mid x_\tau\right),$$

and

$$g_\theta(x_\tau, \tau)[i] \approx \log \mathbb{E}\left[s_\tau[i] - s_\tau[i-1] - 1 \mid x_\tau\right],$$

where f_θ predicts clean token identities and g_θ predicts the log of the expected number of missing tokens in each gap.

In the τ-coordinate, the FlexMDM training objective becomes

$$\mathcal{L}(\theta) = \int_0^1 \mathbb{E}\Bigg[-\frac{B'(\tau)}{1-B(\tau)} \sum_{i: x_\tau^i = \mathbf{m}} \log f_\theta(x_\tau, \tau)\left[i, x_1^{s_\tau[i]}\right] + \frac{1}{1-\tau}\sum_i \phi\left(s_\tau[i] - s_\tau[i-1] - 1,\, g_\theta(x_\tau, \tau)[i]\right)\Bigg] d\tau.$$

where $\phi(x, y) = e^y - xy$. The insertion weight is $1/(1-\tau)$ because

$$\frac{\dot{\alpha}_t}{1-\alpha_t}\, dt = \frac{1}{1-\tau}\, d\tau,$$

and the unmasking weight is $B'(\tau)/(1-B(\tau))$ because

$$\frac{\dot{\beta}_t}{1-\beta_t}\, dt = \frac{B'(\tau)}{1-B(\tau)}\, d\tau.$$

Thus this objective is simply the original FlexMDM objective written in the coordinate τ.

Equivalently, training data can be generated directly in the τ-coordinate. For each clean token j, sample an insertion coordinate

$$A_j \sim \mathrm{Unif}(0, 1).$$

Conditional on A_j, sample an unmasking coordinate $U_j \geq A_j$ by drawing

$$Z_j \sim \mathrm{Unif}(B(A_j), 1), \qquad U_j = B^{-1}(Z_j).$$

At progress level τ, token j is deleted if $\tau < A_j$, masked if $A_j \leq \tau < U_j$, and revealed if $U_j \leq \tau$. The model is then given (x_τ, τ) rather than (x_t, t).

At inference time, choose any desired event-time insertion schedule

$$\Gamma : [0, 1] \to [0, 1].$$

At sampler time t, set $\tau_t := \Gamma(t)$ and query the model as

$$f_\theta(x_t, \tau_t), \qquad g_\theta(x_t, \tau_t).$$

The insertion CTMC in event time is then

$$R_t^{\mathrm{ins}}(x, x \triangleleft_i m) = \frac{\dot{\Gamma}(t)}{1-\Gamma(t)} \exp\big(g_\theta(x, \Gamma(t))[i]\big).$$

Thus the sampler uses the event-time hazard of the chosen inference schedule, but the model itself is always conditioned on the insertion-progress value $\tau_t = \Gamma(t)$.

With an event-time unmasking schedule $\Delta(t)$, the unmasking transition rate is

$$R_t^{\mathrm{unmask}}(x, x[i \leftarrow v]) = \frac{\dot{\Delta}(t)}{1-\Delta(t)} f_\theta(x, \Gamma(t))[i, v].$$

If one wants to preserve the original training unmasking dynamics in τ-coordinates, then the event-time unmasking schedule should just be $\Delta(t) = B(\Gamma(t))$.

The key distinction from a raw-time-conditioned model is that no inverse-time correction is required at inference. A model trained on raw time under α_{train} must be queried at

$$\alpha_{\mathrm{train}}^{-1}(\Gamma(t))$$

when using a new inference schedule Γ. In contrast, an insertion-progress-conditioned model is queried directly at

$$\tau_t = \Gamma(t).$$

Therefore changing the insertion schedule at inference only changes how quickly the sampler moves through insertion-progress space; it does not change the semantic meaning of the model's conditioning variable.

D.3 ANY-ORDER METRICS

To study the any-order inference ability of FlexMDM, we evaluate both FlexMDM and Dream-Coder on the three tree-based metrics, each measuring the any-order performance from a different perspective. Below, we first describe how we parse the extracted Python code into a tree, and then define the metrics precisely.

Parsing into trees. We base our metrics on the natural syntactic structure already exposed by the Python AST (Abstract Syntax Tree) parser. For each generated sample, we first extract the Python code span and parse it into an `ast.Module`, using standard fallbacks for fenced code blocks, closing fences, and longest valid prefixes when direct parsing fails. We then coarsen the full Python AST into a simplified statement-level tree: expression-level nodes are collapsed into their nearest containing statement, docstrings are collapsed into their owner node, and trailing generated tests or examples are optionally grouped into a synthetic reference node. Each Python token is assigned to the deepest compatible AST node, so every tree node is associated with the set of generated tokens in its subtree.

Coverage Before Commitment (CBC). CBC measures whether the model previews multiple sibling subtrees before committing to finishing one of them. This distinguishes genuinely any-order behavior from a merely serial traversal: at a branching point in the program tree, a model with higher CBC begins work on several child blocks before completing any one.

More precisely, let v be a non-leaf node in the syntax tree with k children ($k \geq 1$). We say v is *split* if $k \geq 2$, and *non-split* if $k = 1$. Write $\mathrm{child}(v) = \{c_1, \ldots, c_k\}$, where each c_i represents the subtree rooted at the i-th child node of v. Let $t_{\mathrm{commit}}(v)$ be the first generation step at which any child block $c_i \in \mathrm{child}(v)$ becomes complete, and let $N_{\mathrm{started}}(v)$ be the number of child blocks of v that has at least one generated token by that time. We define

$$\mathrm{CBC}(v) = \frac{N_{\mathrm{started}}(v)}{k}.$$

The normalization by k makes CBC comparable across different branching factors and makes the score measure how much of the *available* sibling structure was explored before commitment. At a split node, $\mathrm{CBC}(v) = 1/k$ is the strictly sequential baseline, while values closer to 1 indicate that the model opened most sibling blocks before completing the first one. For a non-split node, there is no branching opportunity in the inherent code structure, so sequential generation is the natural behavior; setting $\mathrm{CBC}(v) = 1$ ensures that the metric does not penalize the model for being sequential when the program structure itself is sequential.

Lastly, after computing $\mathrm{CBC}(v)$ for all non-leaf nodes v, we aggregate by reporting both the overall average over all non-leaf nodes and the split-only average over nodes with $k \geq 2$; the former measures alignment with the full tree structure, while the latter isolates behavior at genuine branching points.

Return to Unfinished Blocks (RUB). RUB measures whether the model returns to a partially generated child block after moving away from it. This captures a stronger form of any-order inference than CBC: rather than only checking whether several sibling blocks were previewed before commitment, RUB asks whether generation actually moves back and forth across unfinished parts of the same local tree region.

More precisely, let v be a non-leaf split node in the syntax tree. For a child block $c_i \in \mathrm{child}(v)$, let $t_{\mathrm{open}}(c_i)$ be the first generation step at which any token in c_i is generated, and let $t_{\mathrm{finish}}(c_i)$ be the first generation step at which this subtree becomes complete. We say that c_i is *returned to* if, after c_i is opened and before it is finished, the model generates at least one token belonging to another sibling block $c_j \in \mathrm{child}(v)$, $j \neq i$. Let $N_{\mathrm{returned}}(v)$ be the number of children of v that are returned to. We define

$$\mathrm{RUB}(v) = \frac{N_{\mathrm{returned}}(v)}{k}.$$

The normalization by k again makes RUB comparable across branching factors and measures what fraction of the available sibling blocks are revisited after being left unfinished. For a split node, $\mathrm{RUB}(v) = 0$ indicates no back-and-forth behavior across unfinished sibling blocks, while values closer to 1 indicate that many child blocks were revisited during generation. For a non-split node,

there is no alternative sibling block to move to, so we set $\mathrm{RUB}(v) = 1$ to avoid penalizing sequential generation when the underlying code structure offers no branching opportunity.

Graded Return to Unfinished Blocks (RUB$^+$). RUB is binary at each child: it saturates after a single return, so it cannot distinguish a trace that opens every sibling once and then closes them in turn from one that alternates between siblings many times. RUB$^+$ refines RUB into a three-level graded score, while capping the credit at two returns so that it does not reward pathological alternation.

More precisely, let v be a non-leaf split node and let $c_i \in \mathrm{child}(v)$. Restricting attention to generation steps that reveal a token in some child of v, define $\mathrm{visits}(c_i)$ to be the number of maximal contiguous runs of steps during which the model unmasks tokens belonging to c_i's subtree; equivalently, $\mathrm{visits}(c_i) - 1$ counts the number of times the model leaves c_i, works on a sibling, and returns. We define the per-child score

$$s(c_i) = \frac{\min\big(\mathrm{visits}(c_i) - 1, 2\big)}{2} \in \{0, 0.5, 1\},$$

so that $s(c_i)$ takes value 0, 0.5, or 1 according to whether the model never returns to c_i, returns exactly once, or returns at least twice. We then set

$$\mathrm{RUB}^+(v) = \frac{1}{k} \sum_{i=1}^{k} s(c_i).$$

The cap at two returns reflects the view that genuine any-order generation should exhibit a small number of revisits per branch — the kind a human writing code might exhibit — rather than frantic alternation; replacing the binary "did the model ever return" signal with this three-level score lets us separate single-return traces from richer back-and-forth without rewarding pathological interleaving. As with RUB, $\mathrm{RUB}^+(v) = 0$ indicates strictly serial generation at v, and we set $\mathrm{RUB}^+(v) = 1$ for non-split nodes by convention.

Lastly, same as above, we aggregate $\mathrm{RUB}(v)$ and $\mathrm{RUB}^+(v)$ by reporting both the overall average over all non-leaf nodes and the split-only average over nodes with $k \geq 2$.

Open-Block Width (OBW). OBW measures how many sibling blocks the model keeps simultaneously unfinished during generation. This captures the local breadth of any-order inference: at a branching point in the program tree, a model with higher OBW maintains progress on several child blocks at once rather than completing one child block before opening the next.

More precisely, let v be a non-leaf node in the syntax tree. For each generation step t, let $N_{\mathrm{open}}(v, t)$ be the number of children blocks $c_i \in \mathrm{child}(v)$ that has at least one generated token but is not yet complete at time t. We define

$$\mathrm{OBW}(v) = \frac{\max_t N_{\mathrm{open}}(v, t)}{k}.$$

Again, the normalization by k makes OBW comparable across branching factors and measures the largest fraction of available sibling blocks that are simultaneously active. For a non-split node, there is only one child block and we automatically get $\mathrm{OBW}(v) = 1$, which also avoids penalizing sequential generation when the code structure itself is sequential.

Same as the above two metrics, we report both overall and split-only average.

Complete any-order metrics results. We use the default inference setup in the original Dream-Coder paper (Xie et al., 2025), with temperature 0.2 on HumanEval and 0.1 on MBPP, and with confidence based on negative entropy on both benchmarks.

We observed that the models sometimes generate some test cases after the function body itself, such as standalone `print`, `assert`, or `__main__` statements, which may confound the evaluation on the actual function body. Thus, we sanitize the code and discard all nodes corresponding to the test cases. We report the any-order metrics both before and after the sanitizing, labeled as `full_output` and `code_only`, respectively.

As shown in Table 4, with both variants, our FlexMDM significantly outperforms Dream-Coder. Notably, the gap on split-only RUB shows that at branching points in the program tree, Dream-Coder

Table 4: Tree-based any-order metrics for Dream-Coder-7B-Base and FlexMDM (ours) under both `full_output` evaluation, without sanitizing, and `code_only` evaluation, after discarding generated test-case nodes. Overall scores average over all non-leaf nodes, while split-only scores average only over branching nodes.

Benchmark	Model	Temp.	Scope	Overall				Split-only			
				CBC	RUB	RUB$^+$	OBW	CBC	RUB	RUB$^+$	OBW
HumanEval	Dream-Coder	0.2	`full_output`	0.634	0.402	0.379	0.641	0.431	0.081	0.041	0.442
			`code_only`	0.742	0.596	0.583	0.745	0.388	0.066	0.034	0.396
	FlexMDM	0.1	`full_output`	0.833	0.780	0.716	0.855	0.706	0.620	0.510	0.746
			`code_only`	0.835	0.780	0.720	0.857	0.582	0.468	0.329	0.640
MBPP	Dream-Coder	0.1	`full_output`	0.798	0.666	0.664	0.798	0.382	0.011	0.006	0.383
			`code_only`	0.798	0.666	0.664	0.798	0.382	0.011	0.006	0.383
	FlexMDM	0.1	`full_output`	0.876	0.829	0.774	0.892	0.766	0.691	0.588	0.799
			`code_only`	0.876	0.823	0.768	0.891	0.601	0.464	0.306	0.657

almost never revisits a sibling block once it has moved away (split RUB ≈ 0.07 on HumanEval and ≈ 0.01 on MBPP), generating in a near-autoregressive manner, whereas FlexMDM revisits roughly half of all sibling branches on HumanEval (split RUB 0.62 and 0.47 under `full_output` and `code_only` respectively). The ratio between split-only RUB$^+$ and split-only RUB further sharpens this picture: a child returned to exactly once contributes 1 to RUB but only $\frac{1}{2}$ to RUB$^+$, while a child returned to at least twice contributes 1 to both, so this ratio reads off the fraction of returned children that were revisited at least twice. For Dream-Coder the ratio is ≈ 0.50 across both benchmarks, meaning that on the rare occasions it does return to an unfinished sibling, it does so exactly once; for FlexMDM the ratio rises to ≈ 0.70–0.82, meaning that a substantial share of revisits involve two or more genuine returns rather than a single one-pass ABA excursion.

D.4 FlexMDM Inference and Details on Evaluation Setup

At inference time, we follow the standard FlexMDM inference algorithm with top-K for unmasking, as detailed in Alg. 1. For all models, we use 512 sampling steps; Dream-Coder generates up to 512 new tokens, while FlexMDM caps the total sequence length (prompt plus generation) at 768 for HumanEval and 1100 for MBPP. For Dream-Coder, since using low temperature with the confidence-based sampling strategy collapses to one single generation, we use temperature 1.0 for all baseline models for diversity at pass@k. For FlexMDM, due to the inherent stochasticity in the insertion, we use a token temperature of 0.1; the insertion temperature (Appendix D.5) is $T_{\text{ins}} = 0.6$ on MBPP/MBPP+ and the neutral $T_{\text{ins}} = 1$ on HumanEval/HumanEval+. We use the same code extraction and grading criteria from Dream-Coder's official codebase across all models and benchmarks, executing each sample against the benchmark's full test suite under a 30-second wall-clock limit. Results are summarized in Table 5.

Discussion on FlexMDM's downstream performance. FlexMDM does not dominate Dream-Coder-7B uniformly: it trails at low k on HumanEval and by ≤ 1.6 points at $k \geq 2$ on MBPP. Two factors are at play. First, Dream-Coder-7B-Base is already a strong model — the best or second-best on HumanEval and MBPP among comparable baselines (Xie et al., 2025) — so the headroom left to fine-tuning is limited. Second, the token temperature never touches the insertion process, whose stochasticity perturbs *where* code is laid out and therefore matters most at small k. This axis has its own dial: as Appendix D.5 shows, sharpening insertion placement improves every Pass@k on MBPP/MBPP+, whereas on HumanEval we keep the neutral setting, trading Pass@1 for the placement diversity that drives the Pass@16 gains.

D.5 Insertion Temperature

Just as the token temperature shapes the unmasking posterior, the insertion process carries its own natural temperature. At each step, the insertion step of Alg. 1 (Subroutine 1) draws an independent Poisson count in every gap i with rate $\lambda_i = r\, e^{g_i}$, where r is the scalar schedule hazard and

Subroutine 1: FlexMDM inference

Require: Learned functions (f_θ, g_θ)
Require: Discretization $0 = t_1 < \cdots < t_N = 1$
Require: Insertion, Unmasking schedule α_t, β_t

Initialize $X_{t_1} \leftarrow$ `empty_sequence`
for $j = 1$ to $N - 1$ **do**
  $\tau \leftarrow t_{j+1} - t_j$
  Invoke Subroutine 2 for unmasking
  for i **in** $[\mathrm{len}(X_{t_j})] + 1$ **do**
    Set insertion rate $r_\alpha \leftarrow \frac{\dot\alpha_{t_j}}{1-\alpha_{t_j}} \cdot \tau$
    Sample $\ell \sim \mathrm{Poi}\big(r_\alpha \cdot \exp(g_\theta(X_{t_j}, t_j)[i])\big)$
    Insert ℓ masks between $X^{i-1}_{t_j}$ and $X^i_{t_j}$
  end for
end for
return X_{t_N}

Subroutine 2: Unmasking step (top-k)

Set unmasking rate $r_\beta \leftarrow \frac{\dot\beta_{t_j}}{1-\beta_{t_j}} \cdot \tau$
Sample $k \sim \mathrm{Poi}(r_\beta \cdot |\{i \mid X^i_{t_j} = \mathbf{m}\}|)$
for $i \in \{i \mid X^i_{t_j} = \mathbf{m}\}$ **do**
  Sample $v^i \sim \mathrm{Cat}(f_\theta(X_{t_j}, t_j)[i])$
  Compute $\mathcal{C}^i = f_\theta(X_{t_j}, t_j)[i, v^i]$
end for
for i in $\mathrm{argmaxk}(\mathcal{C})$ **do**
  Commit $X^i_{t_j} \leftarrow v^i$
end for

Algorithm 1: **FlexMDM inference.** At each step we perform **unmasking** and **insertion**. For **unmasking**, we use confidence-based top-k selection. The number of mask tokens to **insert** and the number of tokens to **unmask** are drawn from a Poisson distribution. **Notation**: Cat, Poi denote the categorical and Poisson distribution, respectively. $\mathrm{argmaxk}(\mathcal{C})$ is the indices set of the k largest components of $\mathcal{C}$. Adapted from Kim et al. (2025a).

Table 5: Full pass@k results for FlexMDM and Dream-Coder-7B-Base on HumanEval, HumanEval+, MBPP, and MBPP+. FlexMDM decodes with insertion temperature $T_{\text{ins}} = 0.6$ on MBPP/MBPP+ and $T_{\text{ins}} = 1$ on HumanEval/HumanEval+ (Appendix D.5).

Benchmark	Model	pass@1	pass@2	pass@4	pass@8	pass@16
HumanEval	FlexMDM (ours)	50.65	66.60	78.69	86.86	92.07
	Dream-Coder-7B-Base	58.65	72.22	81.34	86.94	90.85
HumanEval+	FlexMDM (ours)	46.61	61.89	73.83	82.07	87.80
	Dream-Coder-7B-Base	53.89	67.05	75.96	81.53	85.37
MBPP	FlexMDM (ours)	64.70	76.73	83.80	88.20	91.27
	Dream-Coder-7B-Base	64.53	77.33	84.66	89.29	92.86
MBPP+	FlexMDM (ours)	54.98	66.11	73.06	77.38	80.69
	Dream-Coder-7B-Base	53.98	66.13	73.55	78.25	82.01

$g_i = g_\theta(\mathbf{x}_t, t)[i]$. By Poisson superposition and thinning, this is exactly equivalent to first drawing the *total* number of insertions $N \sim \mathrm{Poisson}(\Lambda)$, $\Lambda = r\sum_i e^{g_i}$, and then *placing* the N masks multinomially with probabilities $p = \mathrm{softmax}(g)$. The single head g_θ thus encodes two separately tunable quantities — *how many* tokens to insert and *where* — and we define the *insertion temperature* T_{ins} by tempering the placement while holding the total fixed:

$$\lambda_i(T_{\text{ins}}) \;=\; \Lambda \cdot \mathrm{softmax}(g/T_{\text{ins}})_i,$$

which reduces to λ_i at $T_{\text{ins}} = 1$. Because the placement distribution is normalized for every T_{ins}, the expected number of insertions — and hence the length statistics the model was trained on — is invariant to the knob. Lowering T_{ins} makes structural commitments more decisive by concentrating insertions in the gaps the model is most confident about; raising it diversifies placement. Count preservation is what makes placement the right axis to temper: naively rescaling the rates as $e^{g_i/T}$ changes the expected length (the softmax denominator is exactly the missing normalization), while suppressing the stochasticity of the count channel instead drives the sampler off the stochastic-insertion process it was trained on — in our experiments collapsing generation length. The knob is inference-only and adds no compute.

Table 6: Insertion-temperature sweep for FlexMDM (16 samples/task, identical protocol to Table 5); Dream-Coder-7B-Base in italics for reference. **Bold** marks the per-benchmark setting adopted in Table 5. Sharpening to 0.6 helps at every k on MBPP/MBPP+; on HumanEval/HumanEval+, where prompts already fix the scaffold, sharpening recovers little Pass@1 and costs performance at larger k.

Benchmark	T_{ins}	pass@1	pass@2	pass@4	pass@8	pass@16
MBPP	*Dream-Coder-7B-Base*	64.53	77.33	84.66	89.29	92.86
	1.0	62.22	74.61	81.81	86.19	89.68
	0.75	62.85	75.72	83.22	87.69	91.27
	0.60	**64.70**	**76.73**	**83.80**	**88.20**	**91.27**
	0.50	64.30	75.84	82.41	86.42	89.42
MBPP+	*Dream-Coder-7B-Base*	53.98	66.13	73.55	78.25	82.01
	1.0	52.86	64.38	72.04	77.00	80.16
	0.75	53.54	65.26	72.85	77.44	80.69
	0.60	**54.98**	**66.11**	**73.06**	**77.38**	**80.69**
	0.50	55.47	66.36	73.29	77.38	80.16
HumanEval	*Dream-Coder-7B-Base*	58.65	72.22	81.34	86.94	90.85
	1.0	**50.65**	**66.60**	**78.69**	**86.86**	**92.07**
	0.70	51.64	67.02	77.90	85.49	91.46
HumanEval+	*Dream-Coder-7B-Base*	53.89	67.05	75.96	81.53	85.37
	1.0	**46.61**	**61.89**	**73.83**	**82.07**	**87.80**
	0.70	47.45	62.20	73.31	81.51	87.80

Table 6 sweeps T_{ins} under the identical protocol as Table 5, and the two benchmark families respond in opposite ways, tracking their prompt structure. MBPP prompts give a one-line description with no code scaffold, so the model must lay out the program structure itself, and placement noise perturbs exactly these least-reversible early commitments. Sharpening to $T_{\text{ins}} = 0.6$ accordingly improves on the neutral setting at *every* k: it is best or tied at every k on MBPP, within half a point of the sweep-best on MBPP+, and lifts Pass@1 by $+2.5$ (MBPP) and $+2.1$ (MBPP+) — past Dream-Coder-7B-Base. HumanEval prompts instead pin the scaffold in advance (signature, docstring, worked examples), so placement stochasticity acts as useful exploration over valid realizations — the source of the Pass@16 advantage; sharpening recovers less than a point of Pass@1 while giving up performance at larger k, so the neutral $T_{\text{ins}} = 1$ is preferable. We therefore select T_{ins} per benchmark in Table 5 (0.6 on MBPP/MBPP+, 1 on HumanEval/HumanEval+), just as token temperatures are routinely chosen per benchmark (Xie et al., 2025). In the language of Section 3.2, T_{ins} is a dial on the exploration–commitment trade-off of any-order inference: it sharpens placement precisely in the high-positional-uncertainty regime where the prompt does not anchor it.

E Details of Section 4.2

E.1 Data Pre-processing

We use TinyGSM (Liu et al., 2023) as the training corpus for all models in Section 4.2. Each example consists of a prompt P and a target Python solution $\mathbf{x}$. For LatentMDM, we segment each target solution using `\newline` as a delimiter, treating each line of code as a semantic segment. We fix the maximum number of segments to $L_s = 16$ during training; examples with more than L_s segments are truncated, while examples with fewer segments are padded with empty all-<EOS> segments.

Each segment is tokenized and padded to a maximum length of $L_{\text{seg}} = 32$. If a segment exceeds this length, we truncate it and replace the final token with <EOS>. We define $\text{len}(\mathbf{y}^i)$ as the position of the first <EOS> token in segment $\mathbf{y}^i$, which determines the effective length used in the loss and segment-level scoring during inference.

For the autoregressive and token-level MDM baselines, we do not apply segmentation. Instead, we concatenate the prompt and target solution, truncate the resulting sequence to 512 tokens, and pad shorter sequences with <EOS>. All models use the Qwen tokenizer (Yang et al., 2025) with vocabulary size 151,645.

E.2 Architecture

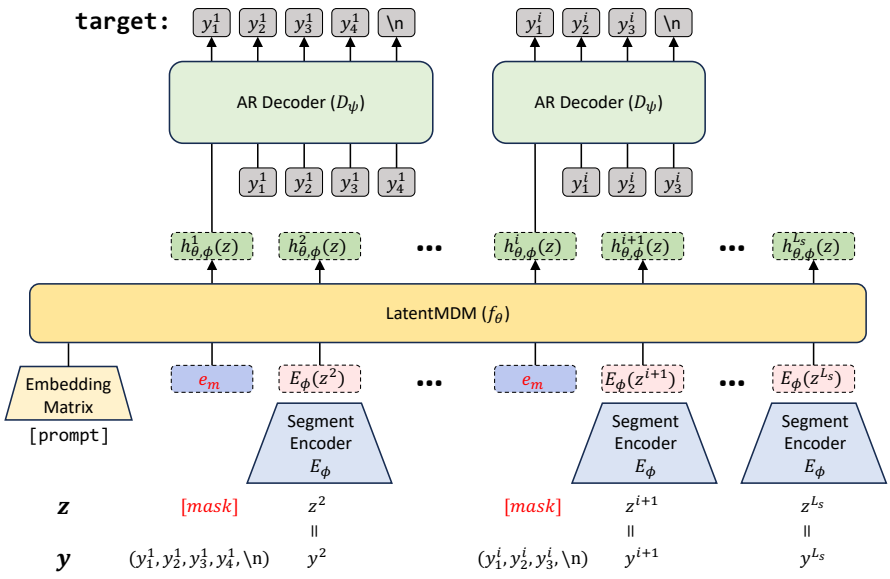


Figure 9: **LatentMDM training pipeline.**

We use a 125M-parameter architecture for LatentMDM and all token-space baselines, following the setup of Kim et al. (2026). All Transformer blocks follow the Qwen2-style design (Yang et al., 2025). We modify the attention mask according to each module: the segment encoder, LatentMDM, and token-level MDM baseline use bidirectional attention, while the autoregressive decoder and AR baseline use causal attention. Unless otherwise specified, all modules use a hidden dimension $H = 512$ and rotary positional embeddings (RoPE).

LatentMDM. The LatentMDM consists of a segment encoder E_ϕ, a bidirectional latent Transformer f_θ, and an autoregressive segment decoder D_ψ. The segment encoder E_ϕ is a bidirectional Transformer followed by mean pooling over non-<EOS> tokens, without an additional projection layer. Unlike LaDiR (Kang et al., 2025) and LD4PG (Lovelace et al., 2023), which use learnable query embeddings to compress variable-length sequences, we use mean pooling for simplicity. The LatentMDM f_θ takes the encoded segment sequence as input, replaces masked segments with a learned mask embedding $e_m \in \mathbb{R}^H$, and uses in-context conditioning by concatenating prompt token embeddings as a prefix to the segment-latent sequence. The autoregressive decoder D_ψ is a causal Transformer that reconstructs each segment autoregressively, using in-context conditioning on the corresponding LatentMDM output $\mathbf{h}^i_{\theta,\phi}(\mathbf{z}) \in \mathbb{R}^{H'}$, which is prepended as a one-token prefix. We set $H' = H = 512$ for all three modules, so no projection layer is required between the encoder, LatentMDM, and decoder.

For LatentMDM and all baselines, we tie the token embedding matrix to the output prediction head. The same embedding matrix is shared across the segment encoder input embedding, the LatentMDM prompt embedding, the decoder input embedding, and the decoder output head. Since the Qwen tokenizer has a vocabulary size of 151,645, embedding tying substantially reduces the parameter count and makes the comparison across models more controlled.

Baselines. The autoregressive baseline is a 14-layer Qwen2-style causal Transformer, and the token-level MDM baseline is a 14-layer Qwen2-style bidirectional Transformer. Both baselines use the same tokenizer, hidden dimension, RoPE positional encoding, and tied input/output embeddings as the LatentMDM.

E.3 Training Details

Figure 9 illustrates the resulting training pipeline. For each training example, the target solution is split into variable-length code segments, and a random subset of segment positions is masked. Unmasked segments are passed through the segment encoder E_ϕ, whereas masked positions are

Subroutine 1: LatentMDM parallel decoding

Require: LatentMDM f_θ, segment encoder E_ϕ, autoregressive decoder D_ψ, prompt $\mathbf{P}$, number of segments L_s, reveal size k
Initialize latent sequence $\mathbf{z} \leftarrow (\mathbf{e_m}, \ldots, \mathbf{e_m})$
Initialize decoded segments $\hat{\mathbf{y}} \leftarrow (\varnothing, \ldots, \varnothing)$
Set $N_s \leftarrow L_s/k$
for $n = 1$ to N_s **do**
  $\mathcal{M} \leftarrow \{i \mid \mathbf{z}^i = \mathbf{e_m}\}$
  $\mathbf{h} \leftarrow f_\theta(\mathbf{z}; \mathbf{P})$
  for $i \in \mathcal{M}$ **do** ▷ Parallel over masked segments
    `# Sample tokens autoregressively`
    `# and compute its score`
    $\hat{\mathbf{y}}^i, s^i \leftarrow$ SAMPLESEGMENT$(D_\psi, \mathbf{h}^i)$
  end for
  Sample top-k indices $\mathcal{S} \subseteq \mathcal{M}$ according to s^i
  for $i \in \mathcal{S}$ **do** ▷ Parallel over selected segments
    $\mathbf{z}^i \leftarrow E_\phi(\hat{\mathbf{y}}^i)$
  end for
end for
return $\hat{\mathbf{y}} = (\hat{\mathbf{y}}^1, \ldots, \hat{\mathbf{y}}^{L_s})$

Subroutine 2: SAMPLESEGMENT

function SAMPLESEGMENT$(D_\psi, \mathbf{h}^i)$
  Initialize: $\hat{\mathbf{y}}^i_0 \leftarrow \varnothing$, $s^i \leftarrow 0$, $l = 0$
  for $t = 1$ to L_{seg} **do**
    $l \leftarrow l + 1$
    $\mathbf{p}_t \leftarrow D_\psi(\cdot \mid \hat{\mathbf{y}}^i_{<t}, \mathbf{h}^i)$
    Sample $\hat{\mathbf{y}}^i_t \sim \text{Cat}(\mathbf{p}_t)$
    $s^i \leftarrow s^i + \log \mathbf{p}_t[\hat{\mathbf{y}}^i_t]$
    if $\hat{\mathbf{y}}^i_t =$ <EOS> **then**
      break
    end if
  end for
  return $\hat{\mathbf{y}}^i, s^i/l$
end function

Algorithm 2: **LatentMDM inference.** Subroutine 1 performs parallel segment-level decoding.

represented by the learned mask embedding $\mathbf{e_m}$. The LatentMDM f_θ then performs bidirectional contextual prediction over the prompt-conditioned segment-latent sequence, producing one latent conditioning vector for each masked segment. Finally, the autoregressive decoder D_ψ reconstructs each masked segment from its predicted latent representation using teacher forcing. Thus, the model learns latent-segment-level masked prediction while retaining token-level autoregressive modeling within each segment.

In the training setup, we never mask the prompt $\mathbf{P}$ for either LatentMDM or the token-level MDM baseline. For LatentMDM, masking is applied only at the segment level over the target solution. Namely, we sample $n \sim \text{Uniform}\{1, \ldots, L_s\}$ and uniformly choose n segment positions to mask among the L_s target segments. If a segment is selected for masking, it is not passed through the segment encoder E_ϕ; instead, its latent representation is directly replaced by the learned mask embedding $\mathbf{e_m}$. Unmasked segments are encoded by E_ϕ and provided to the LatentMDM together with the prompt prefix. Training LatentMDM took approximately three days on 4 80GB NVIDIA A100 GPUs. We defer the remaining optimization hyperparameters and training configurations to Table 8.

For the token-level MDM baseline, masking is applied only to ground-truth response tokens, including <EOS> tokens, while prompt tokens remain unmasked. Concretely, we sample $r \sim \text{Uniform}(0, 1)$ and mask $\lceil r \cdot (512 - \text{len}(\mathbf{P})) \rceil$ tokens uniformly among the response-token positions. This matches the LatentMDM setup in that the conditional prompt is always visible, and the model is trained to reconstruct only the target solution.

E.4 Inference Details

We further describe the LatentMDM inference procedure in a more general setting. In Section 4.2, we present the case where segments are revealed one at a time. More generally, the same procedure can reveal multiple masked segments per iteration, analogous to multi-token decoding in token-level MDMs. Decoding more than one segment per step can improve sampling speed, but introduces the usual speed–quality trade-off in diffusion language model sampling (Zheng et al., 2024; Kim et al., 2025c; Peng et al., 2025; Ben-Hamu et al., 2025; Nie et al., 2025; Wu et al., 2025b; Hayakawa et al., 2025).

Algorithm 2 gives the resulting inference procedure with k-segment parallel decoding. For simplicity, we assume that k divides L_s and define $N_s = L_s/k$. At each iteration, the LatentMDM predicts

contextual latent representations for all currently masked segment positions. The autoregressive decoder then tentatively decodes a candidate segment for each masked position, in parallel across segment positions. We score each candidate by its length-normalized token log-likelihood:

$$s^i := \frac{1}{\mathrm{len}(\hat{\mathbf{y}}^i)} \sum_{j=1}^{\mathrm{len}(\hat{\mathbf{y}}^i)} \log D_\psi\big(\hat{\mathbf{y}}^i_j \mid \hat{\mathbf{y}}^i_{<j}, \mathbf{h}^i_{\theta,\phi}(\mathbf{z})\big) .$$

We then commit the top-k masked segments according to this score, re-encode the committed segments with E_ϕ, and update the corresponding positions in the latent sequence. We use the length-normalized token log-likelihood as the default segment-selection score, and ablate both alternative scoring rules and k in Appendix E.5. KV caching is enabled for the autoregressive decoder and for the autoregressive baseline during inference. We use $L_s = 8$ segment positions at inference time, while keeping the remaining settings consistent with training. For wall-clock measurements, we use a single 80GB NVIDIA A100 GPU and a sampling batch size 1.

E.5 Additional Experiments

Table 7: Zero-shot GSM8K performance of LatentMDM under different segment-selection policies and parallel decoding sizes (k). Token sampling within each segment is performed greedily ($T = 0$).

Model	Segment-selection strategy	$k = 1$	$k = 2$	$k = 4$
LatentMDM	Avg. log-likelihood	**45.5**	**33.8**	**20.8**
	Min. log-likelihood	45.0	30.3	15.2
	First log-likelihood	41.2	28.6	12.7
	L2R	41.3	28.0	12.1
	Random	40.3	33.1	20.3

Ablation on Segment Scoring. We evaluate LatentMDM under several segment-selection policies in a zero-shot setting using greedy token sampling. Avg. log-likelihood is our default scoring rule, which ranks each candidate segment by its length-normalized token log-likelihood. First log-likelihood scores a segment using only the log-likelihood of its first generated token, following the primary slot-scoring strategy used in ReFusion (Li et al., 2025). Min. log-likelihood assigns each segment the minimum token log-likelihood within the generated segment, thereby emphasizing the least confident local decision. L2R reveals segments strictly from left to right, removing the any-order segment-selection mechanism, while Random selects masked segments uniformly at random. Table 7 shows that average log-likelihood performs best, suggesting that reliable segment-level confidence requires aggregating information across the whole generated segment.

Parallel Decoding. We further evaluate LatentMDM under parallel segment decoding, where $k \in \{1, 2, 4\}$ denotes the number of masked segments committed at each decoding iteration. As shown in Table 7, the average log-likelihood score achieves the best performance across all values of k. In contrast, the alternative scoring variants degrade more sharply as k increases, and often fall behind random segment selection in the more parallel regimes. This suggests that effective parallel segment decoding requires a robust segment-level confidence estimate; otherwise, committing multiple segments per iteration can amplify early selection errors.

E.6 Generation Trace

We further provide a generation trace of LatentMDM in Fig. 10.

Table 8: Training hyperparameters and architecture details for LatentMDM on TinyGSM. The reported LatentMDM model size separates the Transformer parameters from the tied token-embedding matrix; the additional 77.6M parameters correspond to this shared embedding matrix.

	Encoder E_ϕ	LatentMDM f_θ	Decoder D_ψ
Model size	6.8M	34.1M+77.6M	6.8M
Hidden dim	512	512	512
MLP dim	1536	1536	1536
Number of layers	2	10	2
Number of heads	8	8	8
Number of KV heads	8	8	8
Attention mask	Bidirectional	Bidirectional	Causal
RMSNorm epsilon	10^{-6}	10^{-6}	10^{-6}
Dropout	0.0	0.0	0.0
Tokenizer		Qwen/Qwen2-0.5B	
Max segment number		16	
Max segment length		32	
Optimizer		AdamW	
Learning rate		3×10^{-4}	
Warmup steps		1000	
Weight decay		0.01	
EMA value		0.9999	
Training Steps		500k	
Max grad norm		1.0	
Batch size per GPU		64	
Number of GPUs		4	

prompt:

```
# Toulouse has twice as many sheep as Charleston. Charleston has 4
# times as many sheep as Seattle. How many sheep do Toulouse,
# Charleston, and Seattle have together if Seattle has 20 sheep?
```



Figure 10: **Generation trace.** At each iteration, the model tentatively decodes candidate segments for all masked positions, scores them using the segment-selection criterion, and commits the highest-scoring segment. The highlighted row indicates the segment committed at the current iteration, while previously revealed segments are shown in gray. The row color reflects the segment score, with stronger colors corresponding to higher-scoring candidates. This example illustrates that LatentMDM generates at the level of semantic code segments rather than individual token positions, enabling a non-left-to-right segment reveal order.