INTO THE CONTINUUM: EFFICIENT SEMI-CONTINUOUS DIFFUSION LLMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Continuous diffusion language models retain partially resolved token information across denoising steps. We study whether pretrained autoregressive (AR) LLMs can acquire this generation mechanism with limited additional data and training. Our efficiency focus is this conversion budget; cross-model inference speed remains to be measured. We present **Semi-Continuous Diffusion Language Model (SC-dLLM)**, the first low-data conversion of pretrained AR LLMs to blockwise continuous-state generation. Each block starts from masks, evolves through continuous states, and yields discrete tokens. A brief continuous endpoint initialization prepares the backbone for same-position prediction. The main adaptation then trains on independently corrupted intermediate states while retaining the vocabulary prediction objective. During generation, token predictions guide stochastic updates and determine when confident positions are committed. Training uses analytically sampled states without an online teacher or unrolled generation trajectories. The 7B conversion uses approximately 1.1B additional training tokens, about $1/41$ of the 45.2B effective tokens reported for ELF's OpenWebText training. Across code generation, mathematics, instruction following, and knowledge-intensive QA, SC-dLLM reaches nine-score averages of 45.2 at 1.5B and 61.2 at 7B, exceeding the reported AR-initializer and Fast-dLLM v2 averages at both scales. At 7B, all four HumanEval and MBPP Base/Plus scores exceed both the same-data AR fine-tuning control and Fast-dLLM v2. These results establish the feasibility of adapting AR backbones to blockwise continuous generation with a small additional training budget using the pretrained vocabulary head.

1 INTRODUCTION

Continuous-state diffusion language models maintain a vector-valued representation of each unresolved token across denoising steps. Unlike a mask that contains no token-specific information, an intermediate vector can carry a partial prediction forward while allowing later predictions to change before discrete commitment (Li et al., 2022; Chen et al., 2026; Hu et al., 2026). This lets subsequent evaluations use the information accumulated in an unresolved position instead of restarting from a mask or treating an early token guess as final (Azangulov et al., 2026). LangFlow and ELF show that embedding-space generation can rival strong discrete diffusion language models (DLMs), but their generators require substantial training (Chen et al., 2026; Hu et al., 2026; Yang et al., 2026). On OpenWebText, ELF reports 45.2B effective training-token exposures (Hu et al., 2026). LangFlow's 1M-step schedule corresponds to 524.3B (Chen et al., 2026). We ask whether a pretrained autoregressive (AR) LLM can acquire continuous-state generation with much less additional data.

AR initialization has made discrete diffusion conversion considerably cheaper. Masked and blockwise methods replace clean next-token inputs with discrete corruption states while retaining much of the pretrained backbone (Sahoo et al., 2024; Wu et al., 2026; Su et al., 2026). Fast-dLLM v2, for example, converts an AR LLM to a block-diffusion model with approximately one billion fine-tuning tokens (Wu et al., 2026). This result does not directly settle the continuous case. A discrete model only needs to recognize token embeddings and designated corruption symbols. A continuous generator must make useful predictions from intermediate vectors at many noise levels. Those vectors are neither clean tokens nor ordinary masks, and their distribution changes throughout generation. The adaptation problem therefore includes learning a new input-state distribution while keeping the strong token predictions inherited from AR pretraining.

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.

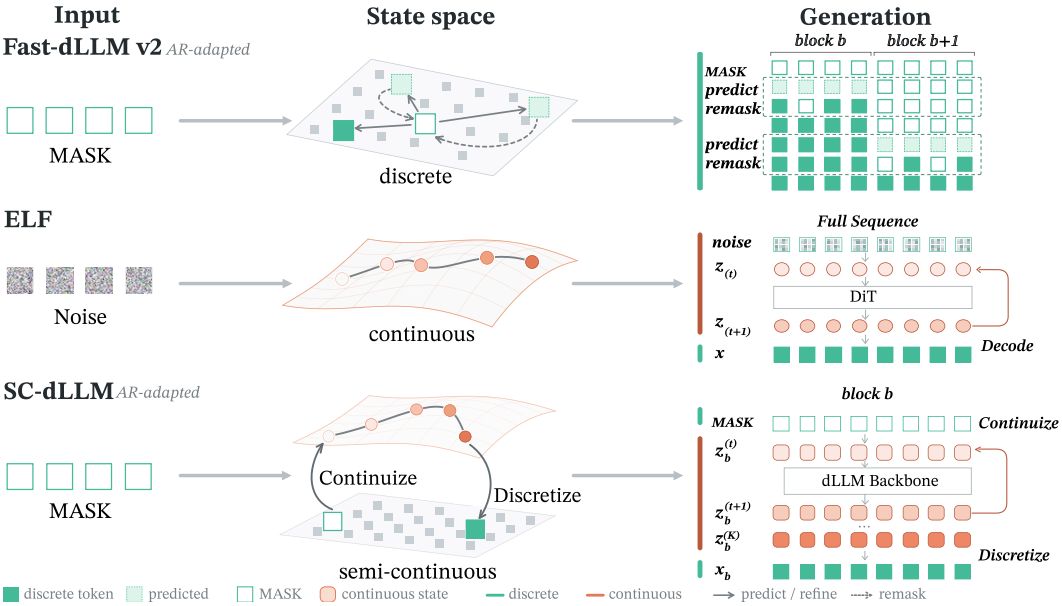

Figure 1: **Discrete, continuous, and semi-continuous generation.** Fast-dLLM v2 refines discrete predictions within each block through prediction and remasking. ELF starts from noise and evolves a full sequence in continuous space before decoding tokens. SC-dLLM adapts an AR backbone to start each block from masks, refine unresolved states continuously, and commit discrete tokens. Completed blocks provide context for subsequent blocks. Orange symbols denote continuous states. Filled and outlined green squares denote discrete tokens and masks, respectively.

Existing AR-to-continuous approaches address different regimes. TESS 2 adapts an AR model to full-sequence simplex diffusion using approximately 45B adaptation tokens (Tae et al., 2025). CADD augments discrete diffusion with a continuous latent process and reports 65B training tokens in its code-generation setting (Zheng et al., 2026). L2D adds a separate diffusion path to an AR model for iterative next-token prediction, leaving the original backbone frozen (Cetin et al., 2025). EvoToken-DLM instead starts from a pretrained masked DLM and trains on unrolled soft-token trajectories (Zhong et al., 2026). SC-dLLM adapts the AR backbone itself to generate continuous states within blocks from analytically sampled training inputs. It requires neither a separate next-token diffusion path nor a coupled discrete–continuous process. To our knowledge, this is the first low-data conversion of a pretrained AR backbone into a blockwise continuous-state generator. Efficiency concerns the additional conversion budget, excluding inherited AR pretraining; matched inference-cost comparisons under common hardware and decoding conditions remain open.

The central difficulty is that AR and blockwise continuous generation ask different questions of the same network. An AR model sees clean prefix tokens under causal attention and predicts the next token. Within an active diffusion block, a position must predict its own target token from a noisy representation while also using information from other positions at different stages of denoising. Simply changing the attention pattern or replacing tokens with Gaussian noise need not preserve a useful prediction interface. The model must distinguish noise level from token identity, exploit less-corrupted neighbors, and remain reliable when a few confident positions have already become clean tokens. With a small conversion budget, these behaviors must be learned without rebuilding the linguistic and reasoning capabilities of the initializer.

We address this problem with **Semi-Continuous Diffusion Language Model (SC-dLLM)**. Figure 1 summarizes its generation structure. SC-dLLM retains the pretrained token embeddings and vocabulary prediction head while adapting the backbone to tokenwise continuous inputs. A brief continuous endpoint initialization prepares the backbone to predict tokens at the mask and clean endpoints. The main adaptation then introduces independently corrupted intermediate states under the same token prediction objective. These states are sampled directly from dataset tokens, without

an online teacher or unrolled generation trajectories. This separates learning to interpret continuous inputs from learning a new output representation for token decoding.

At inference, unresolved positions in each new block start from exact mask embeddings. They then evolve through stochastic continuous states rather than alternating between discrete guesses and masks. Model predictions guide these updates until a position is committed to a token, either when sufficiently confident or at the end of its denoising schedule. We use *semi-continuous* for this mask-to-continuous-to-token trajectory: its starting and committed states are discrete, while the evolving intermediate states are continuous within each active block.

The 1.1B-token conversion budget is approximately $1/41$ of TESS 2's reported 45B adaptation tokens (Tae et al., 2025) and $1/59$ of the 65B tokens used by CADD in its code-generation setting (Zheng et al., 2026). These ratios compare reported additional token exposure across different training regimes and exclude inherited AR pretraining. We assess the resulting models on code generation, mathematics, instruction following, and knowledge-intensive QA. SC-dLLM reaches nine-score averages of 45.2 at 1.5B and 61.2 at 7B, above the corresponding AR initializers, same-data fine-tuned controls, and Fast-dLLM v2. At 7B, all four code scores exceed those of Fast-dLLM v2, with HumanEval Base and Plus reaching 65.7 and 61.2. The results show that a small adaptation budget can support continuous generation while retaining competitive downstream performance.

The paper makes three contributions to low-data continuous-state generation:

- We demonstrate the first low-data conversion of a pretrained 7B AR LLM to blockwise continuous-state diffusion generation using approximately 1.1B additional training tokens.
- We adapt the AR backbone to independently corrupted continuous inputs under its existing token prediction objective, then generate through continuous within-block states before discrete commitment, with committed tokens serving as clean context.
- We show competitive downstream performance at 1.5B and 7B, with higher nine-score averages than the corresponding AR initializers, same-data fine-tuned controls, and Fast-dLLM v2, and gains across all four code scores over Fast-dLLM v2 at 7B.

2 Related Work

Continuous-state language generation. Diffusion-LM and SSD-LM generate through continuous word-embedding or simplex representations (Li et al., 2022; Han et al., 2023). More recent LangFlow, ELF, and RePlaid show that continuous-state models can reach strong language-generation quality under modern training recipes (Chen et al., 2026; Hu et al., 2026; Yang et al., 2026). Other approaches learn a separate latent space (Liang et al., 2026). Masked Language Flow Models and x-Prediction Flow introduce continuous states into pretrained masked diffusion models (Azangulov et al., 2026; Wang et al., 2026), while EvoToken-DLM trains on soft-token trajectories from such models (Zhong et al., 2026). SC-dLLM instead adapts an AR backbone and trains directly on analytically sampled continuous states.

Conversion from AR models. DiffuGPT and DiffuLLaMA establish AR-to-diffusion adaptation through continued training (Gong et al., 2025). Fast-dLLM v2 converts AR models to discrete block diffusion with approximately 1B tokens (Wu et al., 2026). FLUID and OPDLM address causal alignment and on-policy training in related conversion settings (Ma et al., 2026; Su et al., 2026). Continuous or hybrid AR-based precedents operate at a different budget or generation granularity. TESS 2 uses full-sequence simplex diffusion and approximately 45B adaptation tokens (Tae et al., 2025). CADD couples discrete diffusion to continuous latents and reports 65B training tokens for its code-generation setting (Zheng et al., 2026). PlaidQ repurposes a pretrained AR backbone as a bidirectional continuous denoiser for parallel code completion and distills its denoising trajectory to few or one step (Peng et al., 2026). L2D adds a separate continuous diffusion path for iterative next-token prediction while keeping the AR backbone frozen (Cetin et al., 2025). At 7B, SC-dLLM converts the backbone itself for continuous generation within blocks using approximately 1.1B additional training tokens, excluding inherited AR pretraining.

3 METHOD

SC-dLLM converts a pretrained AR model into a blockwise generator (Figure 2). Each active block starts from masks, evolves through continuous states, and yields discrete tokens. Training adapts the AR backbone to these states while retaining its vocabulary prediction objective. During generation, token predictions guide the continuous updates and determine when positions are committed.

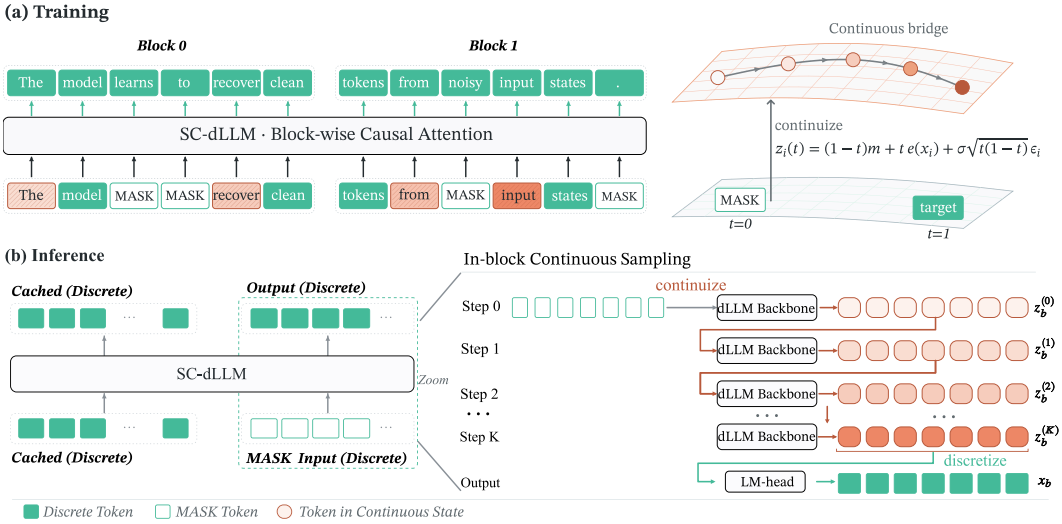

Figure 2: **Training and generation in SC-dLLM.** (a) Blockwise attention supports same-position reconstruction. The schematic summarizes states from the continuous endpoint and interior training branches, omitting the duplicated clean stream. (b) Completed blocks provide cached discrete context as the active block evolves from masks through continuous states to tokens. The LM head is used at every denoising evaluation, although shown only at output. Position-wise early commitment and block-boundary cache updates are omitted in this schematic.

3.1 CONTINUOUS STATES AND TOKEN PREDICTION

Blockwise formulation. Let $x = (x_1, \ldots, x_L)$ be a token sequence, partitioned into contiguous blocks $\mathcal{B}_1, \ldots, \mathcal{B}_M$ of size B. We generate blocks from left to right, conditioning each block on the completed prefix $x_{<\mathcal{B}_b}$, with bidirectional attention inside the current block. We adopt the block-wise attention and complementary training layout of Fast-dLLM v2 (Wu et al., 2026). Completed blocks provide discrete context, while the active block evolves through continuous states. The model weights are initialized from the pretrained AR model. Earlier blocks are no longer denoised, so their discrete tokens provide ordinary context for the active block. Continuous generation is confined to positions whose outputs are still unknown. The model therefore need not represent the entire generated sequence as simultaneous noisy states. Let $\mathcal{V}$ denote the model vocabulary, including the special mask token, and let $E \in \mathbb{R}^{|\mathcal{V}| \times d}$ be the token embedding matrix. We write $e(x_i) = E_{x_i}$ and $m = E_{\texttt{[MASK]}}$ for clean-token and mask embeddings, respectively.

From endpoints to continuous states. We adopt the mask-to-clean Brownian-bridge marginal introduced by Azangulov et al. (2026) in Masked Language Flow Models and adapt it to AR-initialized blockwise generation by sampling each response position's corruption time independently. For each response position i, we define a noisy bridge from the mask embedding to its clean token embedding:

$$z_i(t_i) = (1 - t_i)m + t_i e(x_i) + \sigma\sqrt{t_i(1 - t_i)}\,\epsilon_i, \qquad \epsilon_i \sim \mathcal{N}(0, I_d), \quad t_i \in [0, 1]. \tag{1}$$

Here, σ controls the noise scale in the bridge interior. The exact endpoints $z_i(0) = m$ and $z_i(1) = e(x_i)$ place the generation start at the mask embedding and completed tokens at their clean embeddings, as shown in Figure 2(a). Training samples bridge states analytically without unrolling trajectories. Section 3.3 gives the stochastic generation dynamics.

Time conditioning and token prediction. The model receives $z_i(t_i) + a_\phi(t_i)$ at position i, where the time-dependent input residual is parameterized as

$$a_\phi(t) = \begin{cases} 4t(1-t)\, f_\phi(\gamma(t)), & 0 < t < 1, \\ 0, & t \in \{0, 1\}, \end{cases} \qquad \gamma(t) = \log \frac{\sigma^2(1-t)}{t}. \tag{2}$$

Here, $f_\phi : \mathbb{R} \to \mathbb{R}^d$ maps the interior corruption level to a conditioning vector. The gate makes the residual zero at both endpoints, so the model receives the original mask or clean-token embedding there and a time-conditioned input in the interior. The mapping's parameterization is given in Appendix A.2. Given the current block states Z_b and times $\mathbf{t}_b$, the model predicts a vocabulary distribution over all tokens, including the special mask token

$$q_{\theta,i}(v) = p_\theta(x_i = v \mid x_{<\mathcal{B}_b}, Z_b, \mathbf{t}_b), \qquad i \in \mathcal{B}_b, \quad v \in \mathcal{V}, \tag{3}$$

where θ includes the time-conditioning parameters ϕ. The output at position i predicts x_i itself, with no next-token shift. Fu et al. (2025) show that removing token shift improves average accuracy in Efficient-DLM's discrete block-diffusion conversion setting. In SC-dLLM, it aligns bridge states with their token posteriors and clean-endpoint estimates.

3.2 Progressive Adaptation to Continuous Inputs

The main adaptation trains the AR backbone to predict tokens from continuously corrupted inputs with independent noise levels across positions (Figure 2(a)). A short continuous endpoint initialization prepares same-position blockwise prediction before this training.

A common reconstruction objective. For a sampled training batch, let $\mathcal{S}$ index the response targets across its corrupted views. Both stages minimize same-position token cross-entropy,

$$\mathcal{L}_{\text{rec}}(\theta) = \mathbb{E}\left[-\frac{1}{|\mathcal{S}|} \sum_{i \in \mathcal{S}} \log q_{\theta,i}(x_i) \right]. \tag{4}$$

The expectation is over training batches and the chosen corruption distribution. At every noise level, the target is the original token. The predicted distribution also determines the clean-embedding estimate used for generation (Section 3.3). No separate velocity-regression loss is used. This use of cross-entropy is consistent with the Bregman-divergence flow-matching formulation of LangFlow introduced by Chen et al. (2026). Bridge states are sampled analytically from dataset tokens, so training needs neither an online teacher nor trajectory distillation.

Each corrupted block attends to its own states and the clean prefix, excluding clean targets in its current and future blocks. Appendix A.2 specifies the full attention layout and loss masking.

Stage I: continuous endpoint initialization. This stage trains at the two endpoints of the continuous process. Using the complementary masking layout, it reconstructs masked response tokens ($t_i = 0$) from visible tokens ($t_i = 1$) and the completed prefix. The two views supervise every response token under the same-position rule in Equation 3. The main adaptation subsequently introduces interior states while retaining continuous endpoint updates in its training mixture.

Stage II: tokenwise interior denoising. We then mix continuous endpoint updates with interior updates. Each interior update replaces all eligible response positions with bridge states and supervises their original tokens. We use tokenwise independent Brownian-bridge corruption (BB IID), sampling times and Gaussian perturbations independently across response positions. Independent times place positions with different amounts of target information in the same training block. The model can use clearer neighboring states while predicting noisier ones. Continuous endpoint updates separately expose it to visible clean tokens, the context available after positions are committed during generation. Both corruption branches supervise the original tokens through the same vocabulary head, which remains trainable together with the AR backbone.

Let $\mathcal{L}_{\text{end}}$ and $\mathcal{L}_{\text{iid}}$ denote Equation 4 under continuous endpoint and independent-time corruption, respectively. At optimization step s, branch sampling implements the mixture

$$\mathcal{L}_s = (1 - \lambda_s)\mathcal{L}_{\text{end}} + \lambda_s \mathcal{L}_{\text{iid}}. \tag{5}$$

The schedule starts with $\lambda_s = 0$ and increases the frequency of interior updates while retaining continuous endpoint updates. Together, the branches cover independently evolving continuous states and clean-token context used during generation (Section 3.3). The corruption distributions and schedule values are specified in Appendices A.2 and A.3.

3.3 Semi-Continuous Blockwise Generation

At inference, each new block starts with exact mask embeddings at unknown positions and clean embeddings at any prompt positions. Within the active block, unresolved states evolve along the bridge and high-confidence predictions are committed to tokens (Figure 2(b)). Appendix A.4 provides the time schedule, blockwise execution, caching, and stopping details.

Continuous generation. At each evaluation, the model's vocabulary distribution defines an estimated clean embedding $\hat{e}_i = \sum_{v \in \mathcal{V}} q_{\theta,i}(v) E_v$. For a position at bridge time $0 \le t_i < 1$, stochastic generation takes the form of the following stochastic differential equation

$$\mathrm{d}z_i = \frac{\hat{e}_i - z_i}{1 - t_i}\,\mathrm{d}t_i + \sigma\,\mathrm{d}W_i(t_i), \tag{6}$$

where W_i is a standard Brownian motion and the endpoint estimate is recomputed from the current block states. The drift moves the state toward the predicted clean embedding, while the stochastic term retains the Brownian-bridge formulation. We simulate this process with finite Gaussian bridge transitions, holding the predicted endpoint fixed within each update (Appendix A.4). The state carries information between evaluations, while Gaussian perturbations can move it beyond probability-weighted mixtures of token embeddings. Starting this process from the mask endpoint corresponds to the continuize operation in Figure 1 and Figure 2(b).

Confidence-based early commitment. Confidence-based early commitment stops generation of a position once its token prediction is sufficiently certain (Azangulov et al., 2026; Wu et al., 2026). In discrete masked decoding, confidence selects which masks become tokens. Here it determines when a continuous trajectory can end before reaching the bridge endpoint. The committed token provides clean context, while unresolved positions carry their continuous states into the next denoising step. For the unresolved positions $\mathcal{U}$ in the active sub-block, we choose the most probable non-mask token and its confidence from the same vocabulary distribution,

$$\hat{x}_i = \arg \max_{v \neq \texttt{[MASK]}} q_{\theta,i}(v), \qquad c_i = q_{\theta,i}(\hat{x}_i). \tag{7}$$

Positions with $c_i \ge \tau$ are committed in parallel, setting their states to $e(\hat{x}_i)$ and their bridge time to 1. Before the final scheduled evaluation, positions that remain unresolved advance to the next denoising time step, including when none reaches the threshold. This continuous update uses the same model prediction as the confidence test. Once all positions in the active sub-block are committed, its remaining scheduled denoising evaluations are skipped. Each sub-block uses at most 32 denoising evaluations. At the final step, any positions still unresolved are forcibly committed to the non-mask argmax from that evaluation, setting $z_i = e(\hat{x}_i)$ and $t_i = 1$ regardless of confidence. This terminal step requires no additional model evaluation. Committed positions remain as clean context in later Transformer calls but receive no further state updates.

4 Experiments

We evaluate SC-dLLM's downstream performance, training-token budget, and quality–throughput trade-off under confidence-based early commitment (Section 3.3).

Table 1: **Downstream results at 1B and 7B+ scales.** Published baselines follow Fast-dLLM v2 (Wu et al., 2026). Qwen2.5-FT and SC-dLLM are our own same-protocol results. MLFM's GSM8K score (†) uses its source protocol (Azangulov et al., 2026). Base and Plus are original and extended EvalPlus tests. Scores are percentages (↑). Avg. is the unweighted mean of nine displayed scores for complete rows. SmolLM 2's reported 40.7 is corrected to 40.2 from its displayed scores. Bold and underline mark the highest and second-highest available averages per scale.

Model	#Params	HumanEval		MBPP		GSM8K	MATH	IFEval	MMLU	GPQA	Avg.
		Base	Plus	Base	Plus						
1B-scale Models											
LlaMA-3.2	1.2B	34.1	31.1	34.1	29.4	43.0	23.8	58.9	44.4	24.1	35.9
SmolLM 2	1.7B	34.1	28.7	50.6	46.0	47.7	21.1	55.1	49.1	29.2	40.2
Qwen2.5-1.5B	1.5B	42.1	37.2	48.1	41.3	57.0	46.8	41.2	54.6	30.6	44.3
Qwen2.5-1.5B-FT	1.5B	39.4	35.3	51.4	46.8	58.4	42.8	40.8	57.7	29.8	44.7
Fast-dLLM v2	1.5B	43.9	40.2	50.0	41.3	62.0	38.1	47.0	55.1	27.7	<u>45.0</u>
MLFM	1.35B	–	–	–	–	31.2†	–	–	–	–	–
SC-dLLM (ours)	1.5B	44.0	41.1	49.8	42.3	59.8	39.2	46.7	56.1	27.5	**45.2**
7B+ Models											
LLaDA	8B	35.4	31.7	31.5	28.6	78.6	26.6	59.9	65.5	31.8	43.3
LLaDA-1.5	8B	52.4	–	42.8	–	83.3	42.6	58.2	66.0	36.9	–
LLaDA-MoE	7B	61.6	–	70.0	–	82.4	58.7	59.3	67.2	–	–
Dream	7B	57.9	53.7	68.3	56.1	81.0	39.2	62.5	67.0	33.0	57.6
Qwen2.5-7B	7B	51.2	47.6	57.7	49.5	71.4	73.3	70.8	68.7	33.5	58.2
Qwen2.5-7B-FT	7B	53.5	50.2	59.1	52.2	82.9	69.3	70.2	67.8	32.7	59.8
Fast-dLLM v2	7B	63.4	58.5	63.0	52.3	83.7	61.6	61.4	66.6	31.9	<u>60.3</u>
SC-dLLM (ours)	7B	65.7	61.2	64.3	52.8	82.2	65.0	62.2	66.0	31.3	**61.2**

4.1 Experimental Setup

Model and conversion data. We initialize the 1.5B and 7B SC-dLLM models from the corresponding Qwen2.5-Instruct checkpoints (Yang et al., 2024) and retain their tokenizers. Our implementation discussion focuses on the 7B model. The conversion configuration uses a reasoning-off subset of the Llama-Nemotron post-training data (Bercovich et al., 2025), packed into sequences of 2,048 tokens. Examples remain intact at block-aligned boundaries. Prompt and padding tokens are excluded from the reconstruction loss, and packed examples cannot attend to one another. The conversion uses approximately 1.1B tokens beyond AR pretraining, counted as cumulative non-padding exposure across optimization updates for the conversion run.

Implementation Details. The 7B configuration uses AdamW with a learning rate of 2×10^{-5}, 100 warmup updates, and a global batch size of 128 for 6,000 updates. Parameters and optimizer states are stored in FP32, with BF16 forward computation and gradient checkpointing. Training uses BB IID corruption with $\sigma = 0.2$ after a brief continuous endpoint initialization, following the curriculum in Section 3.2. Generation uses finite Gaussian bridge transitions with blocks of 32 tokens, sub-blocks of 8, at most 32 denoising evaluations per sub-block, and a default early-commitment confidence threshold of $\tau = 0.9$. The evaluation configuration allows at most 1,024 generated tokens within a total context limit of 2,048 tokens. Appendices A.2, A.4, and A provide implementation, sampling, and evaluation details, respectively.

Comparisons and evaluation scope. At each scale, Qwen2.5-Instruct provides the AR reference, and Fast-dLLM v2 Wu et al. (2026) provides the discrete block-diffusion comparison with published main results at $\tau = 1.0$. Our Qwen2.5-FT controls retain the AR next-token objective with the same data and optimization settings as the corresponding SC-dLLM models, accounting for additional exposure. We follow Fast-dLLM v2's benchmark evaluation protocol, selecting $\tau = 0.9$ for SC-dLLM's continuous sampler from Figure 4. The AR-initializer, Fast-dLLM v2, and additional baseline scores in Table 1 are published results grouped by scale. Appendix A.5 gives generation settings, including prompt formatting and output-length and context limits.

4.2 Downstream Performance

The main comparison in Table 1 spans mathematical reasoning on GSM8K (Cobbe et al., 2021) and MATH (Hendrycks et al., 2021), code generation on HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), instruction following on IFEval (Zhou et al., 2023), and knowledge-intensive question answering on MMLU (Hendrycks et al., 2020) and GPQA (Rein et al., 2023). Base and Plus denote original and extended EvalPlus tests on the same generated code (Liu et al., 2023). Metrics are answer accuracy for mathematics, pass@1 for code, prompt-level strict accuracy for IFEval, and accuracy for MMLU and GPQA. Appendix A.6 provides scoring and aggregation details, and Appendix B shows paired examples from the official evaluation dataset. The average weights the nine displayed scores equally, so HumanEval and MBPP each contribute twice through Base and Plus. We therefore interpret this code-weighted aggregate alongside individual task results. MLFM (Azangulov et al., 2026) provides a related continuous-state reference, although it adapts a pretrained masked diffusion model rather than AR weights. Its row counts a 1.028B-parameter backbone and 319M trainable adapters. The reported GSM8K result uses a different prompt and sampling protocol and is included only as context. ELF and LangFlow both support conditional generation. Their published evaluations report translation and summarization for ELF (Hu et al., 2026), and perplexity and zero-shot corpus transfer for LangFlow (Chen et al., 2026). Neither reports this instruction, reasoning, and code suite under a matched protocol, so we do not insert their published metrics into Table 1, which aggregates the nine benchmark scores above.

The most consistent gains over the AR initializers occur in code generation: both SC-dLLM scales improve on all four code scores. At 7B, the gains also exceed those of same-data AR fine-tuning, suggesting that additional data exposure alone does not explain the observed code performance. One plausible explanation is that within-block conditioning helps reconcile local code dependencies while the inherited vocabulary head preserves token-level predictions. The current comparisons evaluate the complete conversion recipe and do not isolate this mechanism.

Relative to discrete blockwise conversion, the 7B model improves all four code scores over Fast-dLLM v2 and reaches a nine-score average of 61.2 versus 60.3. The gains persist under the extended Plus tests, which evaluate the same generated programs more strictly. At 1.5B, the averages are close (45.2 versus 45.0), with improvements on HumanEval Plus, MBPP Plus, and MATH offset in part by lower GSM8K accuracy. Thus, the relative gains over discrete conversion depend on model scale and task, with lower GSM8K accuracy at both scales.

Capability retention also varies by task. Both scales trail their AR initializers on MATH and GPQA, and the 7B model loses IFEval accuracy despite stronger code results. Against same-data AR fine-tuning, the 7B code gains coexist with lower mathematics, instruction-following, and knowledge scores, so the aggregate ranking depends on task weights. The central result is successful AR-to-semi-continuous conversion with limited additional training and competitive downstream performance; the causes of individual capability gains and losses remain unresolved.

4.3 Training Tokens and Downstream Performance

Figure 3 compares all eight 7B+ models on HumanEval Base, which every row reports, against training tokens after initialization: zero for Qwen2.5-7B and approximately 1.1B cumulative non-padding tokens for SC-dLLM and Qwen2.5-FT. Reported budgets are 1.31B nominal packed-token slots for Fast-dLLM v2 (Wu et al., 2026), 580B pretraining tokens for Dream (Ye et al., 2025), 2.3T for LLaDA (Nie et al., 2026), and 21T for LLaDA-MoE (Zhu et al., 2025). LLaDA-1.5 inherits LLaDA's 2.3T-token base and adds preference training (Zhu et al., 2026). SC-dLLM reaches 65.7 on HumanEval Base, versus 63.4 for Fast-dLLM v2 and 53.5 for our same-data AR fine-tuning control. The budget axis excludes inherited AR pretraining but includes reported pretraining for scratch-trained diffusion models. Later fine-tuning for Dream and the LLaDA variants is not fully accounted for, so those positions are lower bounds. The different token-counting conventions make this a contextual comparison rather than a matched data-efficiency ratio. Token exposure differs from processed positions and training FLOPs; cross-model inference costs need separate measurement.

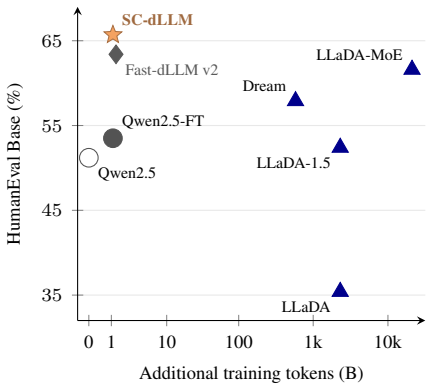

Figure 3: **Training tokens and code accuracy.** All eight 7B+ rows of Table 1 are shown. The horizontal coordinate is $\log_{10}(1+T)$; ticks give token count T in billions. Inherited AR pretraining is excluded, while scratch diffusion pretraining is included. Triangles mark budgets that omit later tuning. Token-counting conventions differ across the published sources.

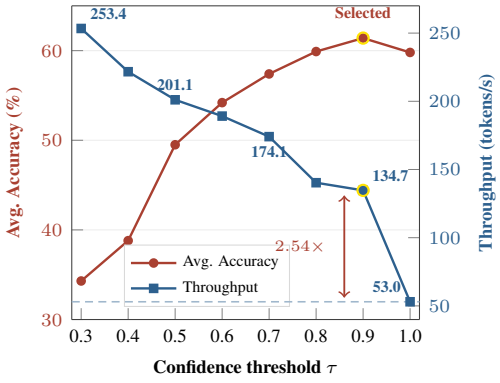

Figure 4: **Early-commitment threshold.** Red circles show Avg. Accuracy, the nine-score mean for one generation seed; blue squares show tokens/s. Yellow rings mark the selected $\tau = 0.9$, and the blue dashed line marks the full-schedule ($\tau = 1.0$) throughput, with early commitment disabled.

4.4 Early-Commitment Threshold

Figure 4 supports $\tau = 0.9$ as the default for our continuous sampler. From $\tau = 0.3$ to 0.9, the nine-score mean rises from 34.3 to 61.4 as throughput falls from 253.39 to 134.69 tokens/s. At $\tau = 1.0$, the mean is 59.8 at 53.02 tokens/s. Thus, $\tau = 0.9$ gives the highest observed mean, 1.6 percentage points above SC-dLLM's full schedule, with $2.54\times$ its throughput. Without early commitment at $\tau = 1.0$, sub-block evaluations cannot be skipped and clean context arrives later. Extra stochastic updates and delayed context may contribute to the lower score.

5 Conclusion

SC-dLLM demonstrates AR-to-semi-continuous conversion with limited additional training. Continuous endpoint initialization prepares same-position prediction before adaptation to independently corrupted interior states under the same vocabulary objective. During generation, unresolved states evolve continuously, and committed tokens provide clean context.

Competitive downstream performance at both evaluated scales validates this conversion paradigm. Code generation shows the most consistent gains over the AR initializers, and same-data AR controls account for additional data exposure. The results establish a practical route to continuous-state language generation with task-dependent capability trade-offs.

6 Limitations & Future Directions

SC-dLLM generates one block at a time and cannot revise committed tokens. An incorrect early commitment can therefore persist as context for later predictions. Future work could study revisable commitment and cross-block refinement, together with their effects on quality and computation.

Capability retention is task dependent: improvements in code generation coexist with lower scores than the AR initializer on some mathematics, instruction-following, and knowledge tasks. The current experiments evaluate the complete conversion recipe at 1.5B and 7B. Controlled studies of the corruption distribution and curriculum could clarify their individual contributions, while larger models and longer contexts would test the range over which the conversion remains effective. We also plan to continue investigating training methods for converting pretrained autoregressive language models into fully continuous diffusion language models.

AI USE STATEMENT

We used generative AI tools to draft and polish text, find related work, check mathematical and methodological consistency, and provide feedback on experimental presentation and result interpretation. We also used these tools for LaTeX compilation and layout checks. We did not use generative AI tools to originate the research idea, run the model-training or benchmark experiments, generate synthetic datasets, or prove mathematical claims. We reviewed the AI-assisted text and checked the retrieved references. We take responsibility for the final content, including text, claims, and artifacts produced with generative AI assistance.

REPRODUCIBILITY STATEMENT

To support reproducibility, we provide training details in Appendix A.1 and implementation details in Appendix A.2. Upon acceptance, we will publicly release the code and trained model weights for SC-dLLM to facilitate reproduction of our results and further research.

REFERENCES

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.

Iskander Azangulov, Kianoosh Ashouritaklimi, Leo Zhang, Simon Vary, and Patrick Rebeschini. Masked language flow models. *arXiv preprint arXiv:2606.27617*, 2026.

Akhiad Bercovich, Itay Levy, Izik Golan, Mohammad Dabbah, Ran El-Yaniv, Omri Puny, Ido Galil, Zach Moshe, Tomer Ronen, Najeeb Nabwani, et al. Llama-nemotron: Efficient reasoning models. *arXiv preprint arXiv:2505.00949*, 2025.

Edoardo Cetin, Tianyu Zhao, and Yujin Tang. Large language models to diffusion finetuning. *arXiv preprint arXiv:2501.15781*, 2025.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

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.

Yonggan Fu, Lexington Whalen, Zhifan Ye, Xin Dong, Shizhe Diao, Jingyu Liu, Chengyue Wu, Hao Zhang, Enze Xie, Song Han, et al. Efficient-dlm: From autoregressive to diffusion language models, and beyond in speed. *arXiv preprint arXiv:2512.14067*, 2025.

Shansan Gong, Shivam Agarwal, Yizhe Zhang, Jiacheng Ye, Lin Zheng, Mukai Li, Chenxin An, Peilin Zhao, Wei Bi, Jiawei Han, et al. Scaling diffusion language models via adaptation from autoregressive models. In *International Conference on Learning Representations*, volume 2025, pp. 5046–5073, 2025.

Xiaochuang Han, Sachin Kumar, and Yulia Tsvetkov. Ssd-lm: Semi-autoregressive simplex-based diffusion language model for text generation and modular control. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 11575–11596, 2023.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

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.

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

Jiajun Liang, Yucheng Liao, Yukang Cao, Jiazhe Wei, Ken Li, Wende Tan, Jiankun Zhang, ZY Cui, Jingkang Yang, Liucheng Guo, et al. Aurora-lm: Autoencoding unified representation for continuous-latent diffusion language modeling. *arXiv preprint arXiv:2608.02602*, 2026.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *NeurIPS*, 36:21558–21572, 2023.

Xiangyu Ma, Teng Xiao, Zuchao Li, and Lefei Zhang. From ar to diffusion: Efficiently adapting large language models with strictly causal and elastic horizons. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 20914–20927, 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. *Advances in Neural Information Processing Systems*, 38:50608–50646, 2026.

Fred Zhangzhi Peng, Kaiwen Zheng, and Anru R Zhang. Distilled continuous diffusion language models can write code in few steps—or one. *arXiv preprint arXiv:2609.04531*, 2026.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. Gpqa: A graduate-level google-proof q&a benchmark. *arXiv preprint arXiv:2311.12022*, 2023.

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

Xingyu Su, Jacob Helwig, Shubham Parashar, Atharv Chagi, Lakshmi Jotsna, Degui Zhi, James Caverlee, Dileep Kalathil, and Shuiwang Ji. Data-efficient autoregressive-to-diffusion language models via on-policy distillation. *arXiv preprint arXiv:2606.06712*, 2026.

Jaesung Tae, Hamish Ivison, Sachin Kumar, and Arman Cohan. Tess 2: A large-scale generalist diffusion language model. In *ACL*, pp. 21171–21188, 2025.

Weitian Wang, Lianlei Shan, Shubham Rai, Cecilia De La Parra, and Akash Kumar. Masked diffusion decoding as x-prediction flow. *arXiv preprint arXiv:2606.29066*, 2026.

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. In *ICLR*, volume 2026, pp. 128353–128370, 2026.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

Zhihan Yang, Wei Guo, Shuibai Zhang, Subham Sekhar Sahoo, Yongxin Chen, Arash Vahdat, Morteza Mardani, and John Thickstun. Continuous diffusion scales competitively with discrete diffusion for language. *arXiv preprint arXiv:2605.18530*, 2026.

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

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. In *ICLR*, volume 2026, pp. 34212–34242, 2026.

Linhao Zhong, Linyu Wu, Bozhen Fang, Tianjian Feng, Chenchen Jing, Wen Wang, Jiaheng Zhang, Hao Chen, and Chunhua Shen. Beyond hard masks: Progressive token evolution for diffusion language models. In *ACL*, pp. 6936–6959, 2026.

Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. Instruction-following evaluation for large language models. *arXiv preprint arXiv:2311.07911*, 2023.

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

Fengqi Zhu, Rongzhen Wang, Shen Nie, Xiaolu Zhang, Chunwei Wu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Llada 1.5: Variance-reduced preference optimization for large language diffusion models. In *ACL*, pp. 11425–11460, 2026.

A Experimental Details

A.1 Model, Data, and Optimization

Tables 2 and 3 list the 1.5B and 7B conversion configurations, respectively. The tokenizer is inherited from the AR checkpoint, with a dedicated mask token registered for corrupted inputs. The time-conditioning module is initialized with a zero output projection, while the AR weights initialize the backbone, token embeddings, and vocabulary prediction head. All these parameters remain trainable. Training examples use same-position reconstruction targets with prompt and padding labels excluded. For gradient accumulation, the loss is normalized by the total number of prediction targets across the full optimizer window, including both complementary endpoint views when present.

Table 2: **1.5B conversion configuration.**

Setting	Value
AR initialization	Qwen2.5-1.5B-Instruct
Packed sequence length	2,048
Training block size	32
Global batch size	128
Optimizer	AdamW
Learning rate	1×10^{-5}; initial linear warmup of 100 updates
Weight decay / gradient norm clipping	0 / 1.0
Parameter and optimizer precision	FP32
Bridge noise scale	$\sigma = 0.2$
Interior noise variable	$\gamma \sim \mathcal{U}[-10, 6]$
Time-conditioning network	64 sinusoidal features, 256 hidden units, SiLU
Completed optimizer updates	6,000
Reference device allocation	8 NVIDIA H200 GPUs

Table 3: **7B conversion configuration.**

Setting	Value
AR initialization	Qwen2.5-7B-Instruct
Packed sequence length	2,048
Training block size	32
Global batch size	128
Optimizer	AdamW
Learning rate	2×10^{-5}; initial linear warmup of 100 updates
Weight decay / gradient norm clipping	0 / 1.0
Parameter and optimizer precision	FP32
Bridge noise scale	$\sigma = 0.2$
Interior noise variable	$\gamma \sim \mathcal{U}[-10, 6]$
Time-conditioning network	64 sinusoidal features, 256 hidden units, SiLU
Completed optimizer updates	6,000
Reference device allocation	8 NVIDIA H200 GPUs

The AR fine-tuning controls use the same data and optimization settings as the corresponding 1.5B and 7B SC-dLLM models, retaining the AR next-token objective. The configured dataset is the reasoning-off Dataset-A export of the Llama-Nemotron post-training collection. The conversion dataset is strictly decontaminated and deduplicated against all benchmarks used in our evaluation. Examples are packed at block-aligned boundaries without truncation. The total packed-token budget is approximately 1.5B, of which approximately 1.1B are non-padding tokens. We use cumulative non-padding exposure for the 7B adaptation budget reported in the paper. This count excludes inherited AR pretraining and is distinct from the number of supervised response targets.

A.2 Implementation Details

Bridge representation and time conditioning. Bridge operations use the pretrained embedding coordinates without normalization, and gradients propagate to the token and mask embeddings. The time-conditioning map is $f_\phi(\gamma) = \mathrm{MLP}_\phi(\mathrm{PE}(\gamma))$, with the dimensions in Table 3. Its final layer is initialized to zero. The endpoint branch bypasses this map, so the divergent values of $\gamma(t)$ at $t \in \{0, 1\}$ are never evaluated.

Parallel block reconstruction. We concatenate corrupted and clean streams to reconstruct all blocks in parallel. Corrupted queries attend to their own corrupted block and strictly earlier clean blocks. Clean queries use block-causal attention, and both streams restrict attention to the same packed example. Thus, corrupted queries cannot access the clean copy of their current or future blocks. Only selected response positions in the corrupted stream contribute to the loss.

Endpoint and interior corruption. For a continuous endpoint update, each block draws $r_b \sim \mathcal{U}(0.001, 1)$ and selects response positions independently with probability r_b. Two complementary views mask the selected and remaining positions, respectively, so each response token is supervised exactly once. Masked positions have $t_i = 0$ and visible positions have $t_i = 1$. For an interior update, all eligible response positions are corrupted and supervised, and we sample time through

$$\gamma_i \overset{\text{iid}}{\sim} \mathcal{U}(-10, 6), \qquad t_i = g(\gamma_i) = \mathrm{sigmoid}(\log \sigma^2 - \gamma_i). \tag{8}$$

Gaussian noise is also independent across positions. Each optimizer update samples one branch and applies the same normalized reconstruction loss.

A.3 Curriculum and Training Schedule

The initial schedule uses continuous endpoint reconstruction exclusively for 1,000 updates. The probability of selecting the BB IID interior branch then increases linearly to 0.5 by update 2,000 and remains at 0.5.

A.4 Sampling Implementation

Finite bridge updates. Equation 1 specifies the marginal distribution at a sampled time. For a fixed clean endpoint, we use a Brownian-bridge coupling with conditional covariance $\mathrm{Cov}(z_i(u), z_i(v) \mid x_i) = \sigma^2(\min\{u, v\} - uv)I_d$ for $u, v \in [0, 1]$. For an unresolved position, $0 \le t_i < 1$. We advance its time to $t_i' \in [t_i, 1]$ and update its state by

$$z_i' = (1 - \alpha_i)z_i + \alpha_i \hat{e}_i + \sigma\sqrt{\frac{(t_i' - t_i)(1 - t_i')}{1 - t_i}}\,\xi_i, \qquad \alpha_i = \frac{t_i' - t_i}{1 - t_i}, \quad \xi_i \sim \mathcal{N}(0, I_d). \tag{9}$$

With $\hat{e}_i = e(x_i)$ fixed, this is the exact conditional Gaussian transition of the bridge. At generation time, the clean endpoint is unknown and $\hat{e}_i$ is its mean embedding under the predicted token distribution. Holding this estimate fixed within an update gives the finite transition used to simulate Equation 6. It is not an exact sample from the mixture of endpoint-conditioned transitions.

Prediction and early-commitment ordering. Sub-blocks are processed from left to right, with each evaluation running the full current block against the cached prefix. The vocabulary distribution is computed once and reused for commitment and the continuous update. Its softmax includes the mask entry, which is suppressed only for discrete token selection. Before the final step, positions in the active sub-block that pass the confidence threshold are committed first. The remaining positions receive a bridge update from the same pre-commit prediction before the next evaluation, even when no position reaches the threshold. At the final step, all remaining positions are forcibly committed using the non-mask argmax of that step's prediction, setting $z_i = e(\hat{x}_i)$ and $t_i = 1$. This terminal discretization replaces the continuous update and requires no additional model evaluation. Committed positions remain as clean context in subsequent full-block evaluations. Once the sub-block is fully committed, its remaining evaluations are skipped.

Time schedule. We use blocks of 32 tokens and sub-blocks of 8. Let U be the number of initially unknown positions in the physical block, indexed by $j = 0, \ldots, U-1$. Each sub-block has $K = 32$ scheduled denoising steps, with model evaluations indexed by $k = 0, \ldots, K-1$. The fixed per-position clock is

$$T_j(k) = \begin{cases} 0, & k = 0, \\ g(6 - 16(k/(K-1))^{\rho_j}), & 1 \le k < K, \\ 1, & k = K, \end{cases} \qquad \rho_j = 0.8 + 0.4\frac{j}{U-1}, \tag{10}$$

with $\rho_0 = 1$ when $U = 1$ and g defined in Equation 8. Evaluation k reads unresolved positions at $T_j(k)$. For $k < K-1$, those still unresolved after the confidence test advance to $T_j(k+1)$ through Equation 9. The last evaluation, $k = K-1$, forcibly commits any remaining positions at the terminal time $T_j(K) = 1$. Thus, K counts model evaluations, while the clock includes the initial state and the final endpoint. The counter restarts at zero for each sub-block. Future sub-blocks remain masked until activated. Each sub-block uses at most 32 denoising evaluations, giving at most 128 across the four sub-blocks of a full 32-token block. This count excludes prompt prefill and block-boundary cache recomputation. The default confidence threshold is $\tau = 0.9$.

Cache updates and termination. Before the next physical block, we recompute the completed block from its committed tokens and append its keys and values to the prefix cache. Intermediate continuous states are never written to this cache, and the terminal block needs no cache update. Generation stops when an end-of-sequence token has all preceding positions resolved, or the output-length budget is exhausted.

A.5 Generation Settings

Generation applies the model's chat template with a generation prompt. Math prompts request step-by-step reasoning and a final answer in `\boxed{...}`. The implementation processes one prompt at a time. For a prompt of P tokens, the output budget is $\min(1024, 2048 - P)$.

A.6 Scoring and Aggregation

Code evaluation uses EvalPlus with HumanEvalPlus version v0.1.10. Base and Plus scores apply the original and extended tests to the same generated program. For our results in Table 1, correctness is averaged over three generation seeds from one trained checkpoint and then across questions, estimating pass@1 for code. Table 1 reports the unweighted mean of its nine task scores, so the four code columns receive four of the nine weights.

MMLU and GPQA likelihood scoring. We follow Fast-dLLM v2's candidate-answer construction and likelihood-based multiple-choice protocol 2509.26328, Fast-dLLM v2. The question and option texts are visible as context, while the answer label being scored is masked and never supplied as a visible target. For an answer target at position j, SC-dLLM uses the model's raw vocabulary logits at position j, consistent with same-position reconstruction. AR scoring uses its next-token alignment. This evaluation reads the masked-endpoint prediction directly without running the continuous generation sampler, so the early-commitment threshold does not affect these scores. Writing $s_\theta(a; c)$ for the conditional log-likelihood score of candidate label a under context c, we predict $\hat{a} = \arg\max_{a \in \mathcal{A}} s_\theta(a; c)$ over the shared answer set $\mathcal{A}$ and report accuracy against the gold label. These scores evaluate candidate answers at the masked endpoint, not the joint sequence likelihood of the continuous sampling process. Comparing accuracy on the same task requires matched visible information and answer choices; SC-dLLM and AR need not have numerically identical sequence-likelihood definitions.

B Qualitative Generation Examples

The examples below come from the official evaluation dataset. Two examples are shown for each of the nine score columns, selected as correct under both confidence thresholds, $\tau = 1.0$ and $\tau = 0.9$. IFEval correctness denotes strict instruction-format validation. MMLU and GPQA are scored by option likelihood, so their entries report the selected options and likelihoods rather than free-form generations. These selected cases illustrate outputs and are not a representative sample.

B.1 HUMANEVAL

HumanEval/97 (Base)

Prompt.

```
def multiply(a, b):
    """Complete the function that takes two integers and returns
    the product of their unit digits.
    Assume the input is always valid.
    Examples:
    multiply(148, 412) should return 16.
    multiply(19, 28) should return 72.
    multiply(2020, 1851) should return 0.
    multiply(14,-15) should return 20.
    """
```

SC-dLLM, $\tau = 1.0$.

```
def multiply(a, b):
    """Complete the function that takes two integers and returns
    the product of their unit digits.
    Assume the input is always valid.
    Examples:
    multiply(148, 412) should return 16.
    multiply(19, 28) should return 72.
    multiply(2020, 1851) should return 0.
    multiply(14,-15) should return 20.
    """
    # Extract the unit digits of both numbers
    digit_a = a % 10
    digit_b = b % 10

    # Calculate the product of the unit digits
    product = digit_a * digit_b

    return product
```

SC-dLLM, $\tau = 0.9$.

```
def multiply(a, b):
    """Complete the function that takes two integers and returns
    the product of their unit digits.
    Assume the input is always valid.
    Examples:
    multiply(148, 412) should return 16.
    multiply(19, 28) should return 72.
    multiply(2020, 1851) should return 0.
    multiply(14,-15) should return 20.
    """
    # Extract the unit digits of the numbers
    digit_a = a % 10
    digit_b = b % 10

    # Calculate the product of the unit digits
    product = digit_a * digit_b

    return product
```

HumanEval/46 (Base)

Prompt.

```
def fib4(n: int):
```

```
    """The Fib4 number sequence is a sequence similar to the Fibbonacci
    sequnece that's defined as follows:
    fib4(0) -> 0
    fib4(1) -> 0
    fib4(2) -> 2
    fib4(3) -> 0
    fib4(n) -> fib4(n-1) + fib4(n-2) + fib4(n-3) + fib4(n-4).
    Please write a function to efficiently compute the n-th element of
    the fib4 number sequence.  Do not use recursion.
    >>> fib4(5)
    4
    >>> fib4(6)
    8
    >>> fib4(7)
    14
    """
```

SC-dLLM, $\tau = 1.0$.

```
def fib4(n: int):
    """The Fib4 number sequence is a sequence similar to the Fibbonacci
    sequnece that's defined as follows:
    fib4(0) -> 0
    fib4(1) -> 0
    fib4(2) -> 2
    fib4(3) -> 0
    fib4(n) -> fib4(n-1) + fib4(n-2) + fib4(n-3) + fib4(n-4).
    Please write a function to efficiently compute the n-th element of
    the fib4 number sequence.
    Do not use recursion.

    >>> fib4(5)
    4
    >>> fib4(6)
    8
    >>> fib4(7)
    14
    """
    if n < 0:
        raise ValueError("Input should be a non integer.")

    fib4 = [0] * (n + 1)
    fib4[0] = 0
    fib4[1] = 0
    fib4[2] = 2
    fib4[3] = 0

    for i in range(4, n + 1):
        fib4[i] = fib4[i - 1] + fib4[i - 2] + fib4[i - 3] + fib4[i - 4]

    return fib4[n]
```

SC-dLLM, $\tau = 0.9$.

```
def fib4(n: int):
    """The Fib4 number sequence is a sequence similar to the Fibbonacci
    sequnece that's defined as follows:
    fib4(0) -> 0
    fib4(1) -> 0
    fib4(2) -> 2
    fib4(3) -> 0
    fib4(n) -> fib4(n-1) + fib4(n-2) + fib4(n-3) + fib4(n-4).
    Please write a function to compute the n-th element of the fib4
    number sequence. Do not use recursion.
```

```
    >>> fib4(5)
    4
    >>> fib4(6)
    8
    >>> fib4(7)
    14
    """
    if n <= 0:
        return 0
    elif n == 1:
        return 0
    elif n == 2:
        return 2
    elif n == 3:
        return 0
    else:
        return fib4(n-1) + fib4(n-2) + fib4(n-3) + fib4(n-4)
```

HumanEval/45 (Plus)

Prompt.

```
def triangle_area(a, h):
    """Given length of a side and high return area for a triangle.
    >>> triangle_area(5, 3)
    7.5
    """
```

SC-dLLM, $\tau = 1.0$.

```
def triangle_area(a, h):
    """Given length of a side and high return area for a triangle.
    >>> triangle_area(5, 3)
    7.5
    """
    return 0.5 * a * h
```

SC-dLLM, $\tau = 0.9$.

```
def triangle_area(a, h):
    """Given length of a side and high return area for a triangle.
    >>> triangle_area(5, 3)
    7.5
    """
    return 0.5 * a * h
```

HumanEval/58 (Plus)

Prompt.

```
def common(l1: list, l2: list):
    """Return sorted unique common elements for two lists.
    >>> common([1, 4, 3, 34, 653, 2, 5], [5, 7, 1, 5, 9, 653, 121])
    [1, 5, 653]
    >>> common([5, 3, 2, 8], [3, 2])
    [2, 3]

    """
```

SC-dLLM, $\tau = 1.0$.

```
def common(l1: list, l2: list):
    """Return the sorted unique common elements for two lists."""
    return sorted(set(l1) & set(l2))
```

SC-dLLM, $\tau = 0.9$.

```
def common(l1: list, l2: list):
    """Return the sorted unique common elements for two lists."""
    return sorted(set(l1) & set(l2))
```

B.2 MBPP

MBPP/427 (Base)

Prompt.

```
"""
Write a function to convert a date of yyyy-mm-dd format to dd-mm-yyyy
    format.
assert change_date_format("2026-01-02") == '02-01-2026'
"""
```

SC-dLLM, $\tau = 1.0$.

```
from datetime import datetime

def change_date_format(date_str):
    # Parse the input string into a datetime object
    date_obj = datetime.strptime(date_str, '%Y-%m-%d')
    # Format the datetime object to the desired format
    return date_obj.strftime('%d-%m-%Y')

# Test cases
assert change_date_format("2026-01-02") == '02-01-2026'
```

SC-dLLM, $\tau = 0.9$.

```
from datetime import datetime

def change_date_format(date_str):
    # Parse the input string into a datetime object
    date_obj = datetime.strptime(date_str, '%Y-%m-%d')

    # Format the datetime object to the desired format
    formatted_date = date_obj.strftime('%d-%m-%Y')

    return formatted_date

# Test cases
assert change_date_format("2026-01-02") == '02-01-2026'
```

This script uses the `datetime` module to parse the input date string into a `datetime` object, then formats it back into the desired `dd-mm-yyyy` format.

MBPP/806 (Base)

Prompt.

```
"""
Write a function to find maximum run of uppercase characters in the given
    string.
assert max_run_uppercase('GeMKSForGERksISBESt') == 5
"""
```

SC-dLLM, $\tau = 1.0$.

```
def max_run_uppercase(s):
```

```
    max_run = 0
    current_run = 0

    for char in s:
        if char.isupper():
            current_run += 1
            max_run = max(max_run, current_run)
        else:
            current_run = 0

    return max_run

# Test cases
assert max_run_uppercase('GeMKSForGERksISBESt') == 5
```

This script defines a function `max_run_uppercase` that calculates the maximum run of uppercase characters in a given string. It uses a simple loop to iterate through the string, keeping track of the current run of uppercase characters and the maximum run found so far.

SC-dLLM, $\tau = 0.9$.

```
def max_run_uppercase(s):
    max_run = 0
    current_run = 0

    for char in s:
        if char.isupper():
            current_run += 1
            max_run = max(max_run, current_run)
        else:
            current_run = 0

    return max_run

# Test cases
assert max_run_uppercase('GeMKSForGERksISBESt') == 5
```

This script defines a function `max_run_uppercase` that calculates the maximum run of uppercase characters in a given string. It uses a simple loop to iterate through the string, keeping track of the current run of uppercase characters and the maximum run found so far.

MBPP/127 (Plus)

Prompt.

```
"""
Write a function to multiply two integers.
assert multiply_int(10,20)==200
"""
```

SC-dLLM, $\tau = 1.0$.

```
def multiply_int(a, b):
    return a * b

# Test cases
assert multiply_int(10, 20) == 200
```

SC-dLLM, $\tau = 0.9$.

```
def multiply_int(a, b):
    return a * b

# Test cases
```

```
assert multiply_int(10, 20) == 200
```

MBPP/242 (Plus)

Prompt.

```
"""
Write a function to count the total number of characters in a string.
assert count_charac("python programming")==18
"""
```

SC-dLLM, $\tau = 1.0$.

```
def count_charac(s):
    return len(s)

# Test the function
assert count_charac("python programming") == 18
```

SC-dLLM, $\tau = 0.9$.

```
def count_charac(s):
    return len(s)

# Test function
assert(count_charac("python programming") == 18)
```

B.3 GSM8K

item 1108

Prompt.

> My mom went to a gardening shop. She bought a set of pots for $19 and a sack of garden soil for $26. At checkout, she used a coupon for $7 off. How much money did my mom spend at the gardening shop?

Reference answer: 38.

SC-dLLM, $\tau = 1.0$.

> To determine the total amount of money my mom spent at the gardening shop, we need to follow these steps:
> 1. Calculate the total cost of the pots and the sack of garden soil.
> 2. Subtract the value of the coupon from the total cost.
>
> First, let's find the total cost of the pots and the sack of garden soil:
>
> $$19 + 26 = 45$$
>
> Next, we subtract the value of the coupon from the total cost:
>
> $$45 - 7 = 38$$
>
> Therefore, the total amount of money my mom spent at the gardening shop is $\boxed{38}$.

SC-dLLM, $\tau = 0.9$.

> To determine the total amount of money my mom spent at the gardening shop, we need to follow these steps:
> 1. Calculate the total cost of the pots and the sack of garden soil.
> 2. Subtract the value of the coupon from the total cost.

First, let's find the total cost of the pots and the sack of garden soil:

$$19 + 26 = 45$$

Next, we subtract the value of the coupon from the total cost:

$$45 - 7 = 38$$

Therefore, the total amount of money my mom spent at the gardening shop is $\boxed{38}$.

item 957

Prompt.

Shawna's father is five times as old as Shawna. Shawna is currently three times as old as Aliya. If Aliya is 3 years old, how old is Shawna's father?

Reference answer: 45.

SC-dLLM, $\tau = 1.0$.

To determine the age of Shawna's father, we need to follow the relationships given in the problem step by step.
1. **Determine Shawna's current age:**
- Aliya is 3 years old.
- Shawna is three times as old as Aliya.

$$\text{Shawna's current age} = 3 \times 3 = 9 \text{ years old}$$

2. **Determine Shawna's father's current age:**
- Shawna's father is five times as old as Shawna.

$$\text{Shawna's father's current age} = 5 \times 9 = 45 \text{ years old}$$

Therefore, Shawna's father is $\boxed{45}$ years old.

SC-dLLM, $\tau = 0.9$.

To determine the age of Shawna's father, we need to follow the relationships given in the problem step by step.
1. **Determine Shawna's current age:**
- Aliya is 3 years old.
- Shawna is three times as old as Aliya.

$$\text{Shawna's current age} = 3 \times 3 = 9 \text{ years old}$$

2. **Determine Shawna's father's current age:**
- Shawna's father is five times as old as Shawna.

$$\text{Shawna's father's current age} = 5 \times 9 = 45 \text{ years old}$$

Thus, Shawna's father is $\boxed{45}$ years old.

B.4 MATH

prealgebra (item 512)

Prompt.

If $y^2 = 36$, what is the greatest possible value of y^3?

Reference answer: 216.

SC-dLLM, $\tau = 1.0$.

To find the greatest possible value of y^3 given that $y^2 = 36$, we start by determining the possible values of y. Since $y^2 = 36$, we have:

$$y = \pm\sqrt{36} = \pm 6$$

This means y can be either 6 or -6. Now, we need to find the greatest possible value of y^3. First, we calculate y^3 for $y = 6$:

$$y^3 = 6^3 = 216$$

Next, we calculate y^3 for $y = -6$:

$$y^3 = (-6)^3 = -216$$

Comparing the two values, we see that the greatest possible value of y^3 is 216. Therefore, the answer is:

$$\boxed{216}$$

SC-dLLM, $\tau = 0.9$.

To find the greatest possible value of y^3 given that $y^2 = 36$, we start by determining the possible values of y. Since $y^2 = 36$, we have:

$$y = \pm\sqrt{36} = \pm 6$$

This means that y can be either 6 or or -6. Now, we need to find the greatest possible value of y^3.
If $y = 6$, then $y^3 = 6^3 = 216$.
If $y = -6$, then $y^3 = (-6)^3 = -216$.
Comparing these two values, the greatest possible value of y^3 is 216. Therefore, the answer is:

$$\boxed{216}$$

algebra (item 159)

Prompt.

The function $f(n) = n^2 + n + 17$ for $0 \leq n \leq 15$ generates prime numbers. What is the value of $f(10) - f(9)$?

Reference answer: 20.

SC-dLLM, $\tau = 1.0$.

To find the value of $f(10) - f(9)$ for the function $f(n) = n^2 + n + 17$, we need to calculate $f(10)$ and $f(9)$ separately, and then find the difference between these two values.
First, let's calculate $f(10)$:

$$f(10) = 10^2 + 10 + 17$$

Calculate 10^2:

$$10^2 = 100$$

Add 10 to 100:

$$100 + 10 = 110$$

Add 17 to 110:

$$110 + 17 = 127$$

So, $f(10) = 127$.

Next, let's calculate $f(9)$:

$$f(9) = 9^2 + 9 + 17$$

Calculate 9^2:

$$9^2 = 81$$

Add 9 to 81:

$$81 + 9 = 90$$

Add 17 to 90:

$$90 + 17 = 107$$

So, $f(9) = 107$.
Now, we find the difference between $f(10)$ and $f(9)$:

$$f(10) - f(9) = 127 - 107$$

Calculate the subtraction:

$$127 - 107 = 20$$

Therefore, the value of $f(10) - f(9)$ is $\boxed{20}$.

SC-dLLM, $\tau = 0.9$.

To find the value of $f(10) - f(9)$ for the function $f(n) == n^2 + n + 17$, we need to calculate $f(10)$ and $f(9)$ separately, and then find the difference between these two values.
First, let's calculate $f(10)$:

$$f(10) = 10^2 + 10 + 17$$

Calculate 10^2:

$$10^2 = 100$$

Add 10 to 100:

$$100 + 10 = 110$$

Add 17 to 110:

$$110 + 17 = 127$$

So, $f(10) = 127$.
Next, let's calculate $f(9)$:

$$f(9) = 9^2 + 9 + 17$$

Calculate 9^2:

$$9^2 = 81$$

]
Add 9 to 81:

$$81 + 9 = 90$$

]
Add 17 to 90:

$$90 + 17 = 107$$

So, $f(9) = 107$.
Now, we find the difference between $f(10)$ and $f(9)$:

$$f(10) - f(9) = 127 - 107$$

Calculate the subtraction:

$$127 - 107 = 20$$

Therefore, the value of $f(10) - f(9)$ is $\boxed{20}$.

B.5 IFEVAL

item 2801

Prompt.

> Rewrite the following blog as a list of exactly 4 bullet points: "The world is a beautiful place. The sun is shining, the birds are singing, and the flowers are blooming. It's a perfect day to go outside and enjoy all that nature has to offer." The bullet points should be in markdown such as:
> * Bullet point 1
> * Bullet point 2

SC-dLLM, $\tau = 1.0$.

> * The world is a beautiful place
> * The sun is shining
> * The birds are singing
> * The flowers are blooming

SC-dLLM, $\tau = 0.9$.

> * The world is a beautiful place
> * The sun is shining
> * The birds are singing
> * The flowers are blooming

item 1098

Prompt.

> What is a name that people call God? Please give exactly two different responses. Separate the responses with 6 asterisk symbols: ******.

SC-dLLM, $\tau = 1.0$.

> 1. Allah
> ******
> 2. Jehovah

SC-dLLM, $\tau = 0.9$.

> 1. Allah
> ******
> 2. Jehovah

B.6 MMLU

clinical knowledge (item 184)

Question and options.

> Metabolism is determined by the:
> A. size of proteins in the cell.
> B. availability of amino acids.
> C. proteins formed as dictated by the genetic material.
> D. amino acid composition of the ribonucleic acids.

Reference choice: C.

$\tau = 1.0$**:** selected option C; log-likelihoods for (A, B, C, D): `-7.4581, -7.4581, -2.5831, -6.8331`.

$\tau = 0.9$**:** selected option C; log-likelihoods for (A, B, C, D): `-7.4581, -7.4581, -2.5831, -6.8331`.

electrical engineering (item 16)

Question and options.

> In decimal number system what is MSD
>
> A. First digit from left to right
> B. First digit from right to left
> C. Middle digit
> D. Mean of all digits

Reference choice: A.

$\tau = 1.0$**:** selected option A; log-likelihoods for (A, B, C, D): `-5.5048, -7.8798, -8.7548, -12.3173`.

$\tau = 0.9$**:** selected option A; log-likelihoods for (A, B, C, D): `-5.5048, -7.8798, -8.7548, -12.3173`.

B.7 GPQA

Chemistry / Organic Chemistry

Question and options.

> cyclooctatetraene was heated with maleic anhydride in a 1:1 ratio, forming product 1.
> 1 was heated with methnol and a small amount of sulfuric acid, forming product 2.
> 2 was heated with cyclopentadiene, forming final product 3.
> what is the structure of the major isomer of product 3?
>
> A. name: dimethyl (1R,4S,4aS,4bR,5S,8R,8aS,8bR,10S,11R)-1,4,4a,4b,5,8,8a,8b-octahydro-1,4-ethano-5,8-methanobiphenylene-10,11-dicarboxylate
> SMILES: O=C(OC)1(C=C2)3(435C=C4C5)21C(OC)=O
> B. name: dimethyl (1R,4S,4aS,4bR,5R,8S,8aS,8bR,10R,11R)-1,4,4a,4b,5,8,8a,8b-octahydro-1,4-ethano-5,8-methanobiphenylene-10,11-dicarboxylate
> SMILES: O=C(OC)1(C=C2)3(435C=C4C5)21C(OC)=O
> C. name: dimethyl (1R,4S,4aR,4bR,5R,8S,8aS,8bS,10S,11R)-1,4,4a,4b,5,8,8a,8b-octahydro-1,4-ethano-5,8-methanobiphenylene-10,11-dicarboxylate
> SMILES: O=C(OC)1(C=C2)3(435C=C4C5)21C(OC)=O
> D. name: dimethyl (1S,4R,4aR,4bR,5S,8R,8aS,8bS,10S,11R)-1,4,4a,4b,5,8,8a,8b-octahydro-1,4-ethano-5,8-methanobiphenylene-10,11-dicarboxylate
> SMILES: O=C(OC)1(C=C2)3(435C=C4C5)21C(OC)=O

Reference choice: A.

$\tau = 1.0$**:** selected option A; log-likelihoods for (A, B, C, D): `-0.4455, -2.3205, -2.3205, -2.0705`.

$\tau = 0.9$**:** selected option A; log-likelihoods for (A, B, C, D): `-0.4455, -2.3205, -2.3205, -2.0705`.

Biology / Molecular Biology

Question and options.

> A signal transduction network in living cells amplifies subtle extracellular signals to signalling intracellular storms involving thousands of molecules. This amplification ensures

proper cell responses to stimuli. Which of the examples below contains all classical components of the signal transduction process: first and second messengers, signal transducer, primary and secondary effectors?

A. Hydrophilic molecule – Receptor in the cell membrane – G-protein – Adenylate cyclase – cAMP – Protein kinase – Transcription factor – Gene expression activation
B. RBOH – ROS – ROS-sensitive proteins – Lipid peroxidation – Ion leakage – Ion channel opening
C. Self-peptide – Receptor in the cell membrane – Ca2+ ions – NADPH oxidase – Ion channel – Ca2+ ions –Transcription factor – Gene expression activation
D. Hormone molecule 1 – Ion channel – Hormone molecule 1 – Kinase 1 – Kinase 2 – Kinase 3 – Ion channel 2 opening

Reference choice: A.

$\tau = 1.0$**:** selected option A; log-likelihoods for (A, B, C, D): `-0.0421, -4.7921, -4.9171, -4.9171`.

$\tau = 0.9$**:** selected option A; log-likelihoods for (A, B, C, D): `-0.0421, -4.7921, -4.9171, -4.9171`.