FROM POSITION RISKS TO BLOCK SURVIVAL: FASTER GENERATION FOR DIFFUSION LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Diffusion language models (DLMs) can accelerate generation by predicting multiple tokens in parallel, but there is a mismatch between how these tokens are predicted and how they ultimately contribute to generation. Parallel predictions can hardly condition on the tokens selected earlier within the same block, even though their validity depends on this realized prefix. Under the popular proposal–verification decoding, this mismatch makes errors highly asymmetric: an early rejection prevents all subsequent proposals from contributing decoding progress. We introduce BRISK-DLM, a framework that addresses both mismatches by optimizing proposal learning and selection for verified progress. BRISK-DLM trains on self-generated sequences, using risk–reward weighting to dynamically prioritize positions by their impact on verified progress and decoding cost. During inference, a lightweight prefix-conditioned corrector reranks existing candidates using previously selected tokens and preferences distilled from the model's own verifier. The corrector reuses the backbone's parallel representations and requires no additional backbone evaluation, while fused execution keeps its overhead small. BRISK-DLM improves end-to-end throughput by up to 37.4% while preserving task quality, establishing a new quality–throughput frontier for DLM generation.

1 INTRODUCTION

Autoregressive language models generate one token at a time, making sequential dependence a fundamental bottleneck for inference. Diffusion language models (DLMs) offer a different computational structure: multiple future token positions can be predicted within a single model evaluation. Although this capability is realized differently across discrete diffusion, masked prediction, and block-based formulations, recent DLMs have scaled to general-purpose language modeling and reasoning while substantially narrowing the quality gap with autoregressive models (Austin et al., 2021; Sahoo et al., 2024; Lou et al., 2024; Nie et al., 2025; Arriola et al., 2025). This raises an appealing possibility: language generation need not be tied to one-token-at-a-time computation.

Yet predicting tokens in parallel is not the same as making progress in parallel. In DLMs, future positions are predicted simultaneously, before the token choices at earlier positions within the same block are known, even though the appropriate prediction at a later position depends on this realized prefix. Under the increasingly popular proposal–verification decoding(Chen et al., 2023), this mismatch becomes especially costly: proposals are verified from left to right, so an early rejection prevents all subsequent proposals from contributing immediate decoding progress. Therefore, parallel generation faces two fundamental consequences of prefix dependence: *which positions are worth improving* depends on how much downstream progress they offer, while *which tokens should be selected* depends on the prefix realized before them. This perspective exposes a gap in existing approaches to parallel DLM generation. Prior work improves model capability(Gong et al., 2024), block-parallel generation(Fu et al., 2025), specialized decoding(Arriola et al., 2025), token editing(Bie et al., 2026), caching(Wu et al., 2025), and acceptance-aware training (Wu et al., 2026). However, proposal learning and within-block token selection are typically optimized separately, even though both determine how much parallel computation becomes usable output.

Instead, we argue that DLM efficiency should be organized around a common principle: *verified progress per model evaluation*. Achieving this requires improving both the proposal distribution itself and how its candidates are selected. This raises two integral questions. During learning, how

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.

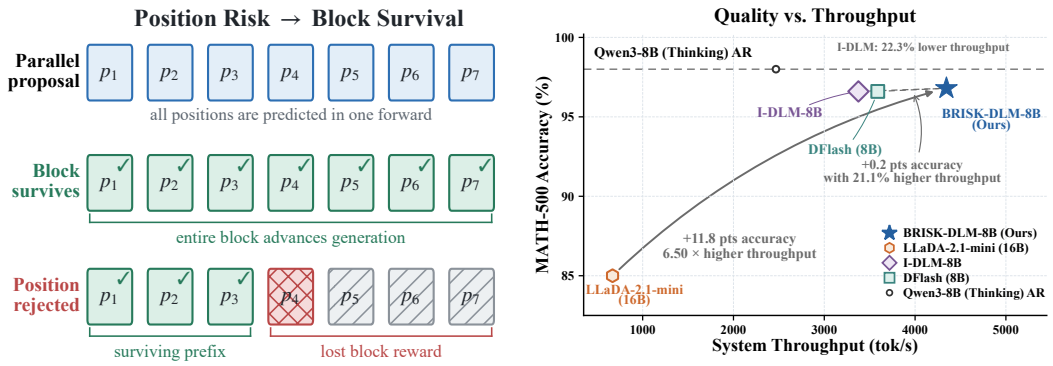

Figure 1: **From position risks to the quality–throughput frontier. Left:** Parallel proposals advance generation only through the consecutively accepted prefix. **Right:** BRISK-DLM shifts the quality–throughput frontier toward higher throughput while preserving quality.

should the model allocate its capacity across proposal positions? A position is more valuable to improve when it is likely to fail and when its survival unlocks substantial downstream progress. During decoding, which candidate should be selected? A token that is locally plausible under parallel prediction may become inappropriate once earlier choices within the block are realized.

Based on this principle, we introduce **BRISK-DLM**, a framework that aligns both proposal learning and candidate selection with verified decoding progress. During training, BRISK-DLM learns from correctness-filtered trajectories generated under the deployed decoding policy and assigns learning signal according to each proposal position's marginal effect on decoding efficiency. This risk–reward aware objective concentrates model capacity on the errors that currently limit block progress, while shifting emphasis as the acceptance profile evolves. During inference, BRISK-DLM complements stronger proposals with a lightweight prefix-conditioned corrector that reranks existing candidates using the tokens already selected within the block. The corrector is trained to reproduce preferences from the model's own verifier on states encountered by the deployed proposal policy. Importantly, the DLM backbone remains fully parallel: the corrector reuses its representations and candidate set, requires no additional backbone evaluation, and is integrated through fused execution so that improved prefix survival translates into end-to-end speedup. Figure 1 connects this position-to-block mechanism to the ultimate objective of improving throughput without sacrificing quality.

Overall, this paper makes the following contributions:

- We identify verified progress per model evaluation as a unifying objective for efficient parallel DLM generation. This view connects two challenges that are typically treated separately: how learning capacity should be allocated across proposal positions, and how parallel candidates should be selected once the within-block prefix is realized.
- We introduce BRISK-DLM to address both challenges. A risk–reward objective dynamically prioritizes proposal positions according to their marginal effect on verified progress, while a lightweight prefix-conditioned corrector distills verifier preferences on policy-reached states to assemble more compatible proposal blocks without additional backbone evaluations.
- We evaluate BRISK-DLM across different model architectures, scales, and reasoning tasks under diverse serving configurations. BRISK-DLM increases verified tokens per forward by up to 37.4% over existing advances, including DFlash (Chen et al., 2026), while preserving task quality. These results unlock a new quality–throughput frontier for DLM inference.

2 Related Work and Motivation

2.1 Background and Related Work

At each step, the DLM predicts token distributions for multiple unresolved positions in parallel, and a decoding schedule determines which predictions are committed or revised before the next step. Masked DLMs apply this process across an entire response, while block-based variants maintain a clean prefix and resolve one block (e.g., 16 tokens) at a time (Austin et al., 2021; Sahoo et al., 2024; Nie et al., 2025; Arriola et al., 2025). While converting one model evaluation into progress on mul-

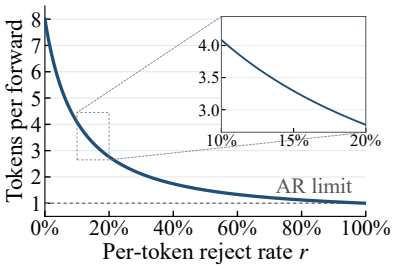

Figure 2: **Rejection limits effective parallelism.** Theoretical TPF at stride $N = 8$.

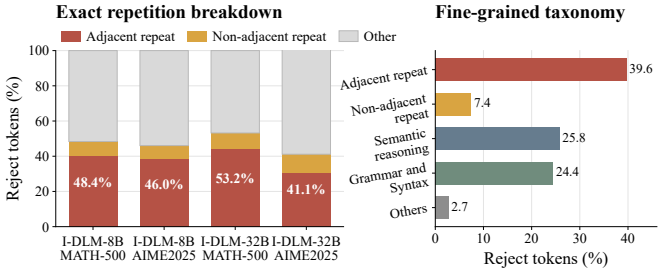

Figure 3: **Empirical patterns of proposal rejection.** Repetition accounts for a substantial fraction of rejected tokens, alongside cases involving semantic reasoning and grammar.

tiple tokens, parallel prediction also reduces causal information: later positions are computed before the actual tokens selected at earlier positions are known. This creates an opportunity for lightweight prefix-conditioned refinement at inference, without sacrificing parallel backbone computation.

Verified parallel generation. A growing class of DLM uses proposal–verification to convert parallel predictions into output, including DFlash (Chen et al., 2026; Wu et al., 2026), self-speculative block-diffusion method such as SSD, FreeDave, and S2D2 (Gao et al., 2025; Wu & Zhang, 2025; Han et al., 2026), and I-DLM with introspective strided decoding (ISD) (Yu et al., 2026). Although implementations differ, they share a common structure: multiple future tokens are proposed in parallel, while a verifier determines how much can be committed. An early rejection prevents later proposals from contributing immediate progress, making positions unequal in their effect on decoding efficiency. Meanwhile, verification evaluates each candidate under a realized prefix unavailable when the parallel proposal was produced. We provide a detailed related work in Appendix A.

2.2 Rejection Severely Limits Effective Throughput

Block size measures *nominal* parallelism, but realized decoding progress depends on how much of the proposed block survives verification. Consider a verifier that checks M parallel proposals from left to right, and let a_i denote the probability that position i is accepted conditioned on the preceding prefix. The probability that verification reaches through position k is $A_k = \prod_{i=1}^{k} a_i$. Hence, the expected accepted-prefix length is $\mathbb{E}[L] = \sum_{k=1}^{M} A_k$. This simple relation captures the central inefficiency of verified parallel generation: acceptance compounds across positions. Even moderate per-position rejection can eliminate a large fraction of the useful parallelism available in a block.

The effect can be stronger when rejection also changes the number of model evaluations required by the decoder. For example, under the ISD execution rule (Yu et al., 2026), a stride of N yields

$$\mathrm{TPF}_N(r) = \frac{2 + \sum_{i=1}^{N-2}(1-r)^i}{2 - (1-r)^{N-1}} \tag{1}$$

where the denominator accounts for the additional forward-pass cost induced when a proposal block does not fully survive. Figure 2 plots this relation for $N = 8$. TPF drops from 8 with no rejection to 4.09 at 10% rejection, 3.29 at 15%, and 2.77 at 20%. Thus, seemingly modest changes in proposal reliability can translate into large differences in realized throughput. Figure 3 further shows that these rejections are structured rather than uniform: exact repetition accounts for 41.1%–53.2% of rejected tokens, with adjacent repetition forming the largest fine-grained category.

These observations motivate optimizing the acceptance profile rather than relying on block width or average token accuracy as proxies for decoding efficiency. Moreover, because progress depends on prefix survival, improvements at different positions need not have equal value. This motivates the decoder-aware credit assignment developed in Section 3.

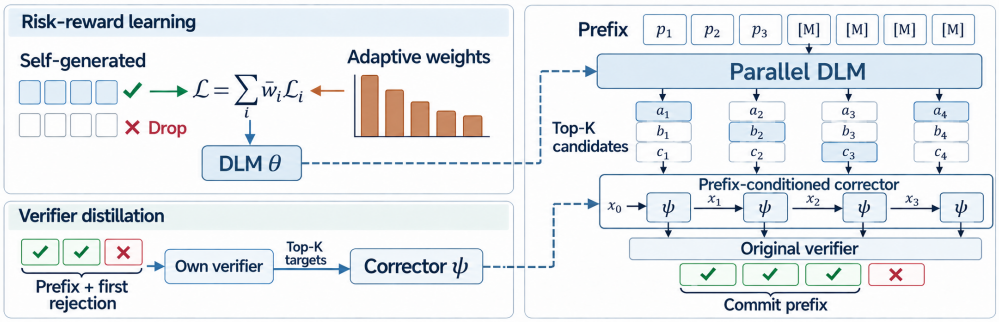

Figure 4: BRISK-DLM overview. Left: risk–reward learning and verifier distillation. Right: prefix-conditioned proposal selection followed by verification.

3 BRISK-DLM: Optimizing Effective Block Progress

3.1 Overview

In this paper, we consider the class of verified parallel decoders, in which a model proposes M future tokens in parallel and a prefix-conditioned verifier determines how much of the proposal can be committed. This abstraction encompasses both DLM-drafter systems with a separate autoregressive verifier, such as DFlash (Chen et al., 2026) and D-Pace (Wu et al., 2026), and self-verifying designs such as I-DLM (Yu et al., 2026), where parallel proposal and causal verification are provided by the same model. BRISK-DLM operates on this shared proposal–verification structure.

This popular structure exposes two places where parallel prediction is misaligned with realized decoding progress. First, the proposal model is typically trained position-wise, although improving different positions can have very different effects on the amount of verified output obtained from a model evaluation. Second, the backbone computes all proposal positions before the token choices at earlier positions are known, even though candidates at a later position depend on this realized prefix.

BRISK-DLM addresses these two mismatches at their respective interfaces. As shown in Figure 4, *Risk-reward proposal learning* allocates model capacity according to each position's marginal contribution to verified progress (§3.2). *Prefix-conditioned proposal selection* then reuses the parallel backbone representations while conditioning the final token choices on the prefix already selected within the block (§3.3). Together, BRISK-DLM improves candidate quality and assemble those candidates into lower-rejection blocks, while leaving the underlying DLM architecture, verifier, rejection correction, and stopping rule unchanged, and requiring no additional backbone forward pass.

3.2 Risk–Reward Proposal Learning

Inference-time selection can only choose among candidates supplied by the proposal model, so improving verified progress first requires improving the proposal distribution itself. BRISK-DLM aligns proposal learning with the model's realized decoding behavior along two dimensions: (i) *which states it learns from* and (ii) *where learning signal is allocated*. Fixed offline sequences do not reproduce the contexts and continuation patterns generated by the model itself, while uniform token-wise losses ignore that errors at different positions have different consequences for effective block progress. We address these issues as a single learning process: self-generated trajectories expose the proposal states and failure patterns encountered during decoding, and position-aware weights determine how the resulting training signal should be allocated.

Specifically, we sample multiple responses from the initial model checkpoint using the deployed decoding procedure and retain trajectories whose final answers are correct. Such filtering avoids reinforcing unsuccessful generations while preserving diverse solution trajectories, bringing the training distribution closer to inference. We analyzed the effect of data selection in Section 4.2.

Marginal verified-progress weighting. Within each retained trajectory, we adjust the contribution of every proposal position according to its effect on expected block progress. Improving an early position protects every subsequent proposal, whereas later positions become increasingly important

once the preceding prefix is reliable. Hence, the relevant bottleneck changes as the model improves and cannot be captured by uniform weighting or a fixed positional schedule.

For a block of M proposal positions, let a_i be the probability that position i is accepted conditioned on the preceding prefix, and define its survival probability as $A_k = \prod_{j=1}^{k} a_j$ with $A_0 = 1$. If L denotes the accepted-prefix length, its distribution and the resulting decoding efficiency are

$$\begin{aligned} \Pr(L = k) &= \begin{cases} A_k(1 - a_{k+1}), & 0 \le k < M, \\ A_M, & k = M, \end{cases} \\ \mathrm{TPF}(\mathbf{a}) &= \frac{\sum_{k=0}^{M} \Pr(L = k) g_k}{\sum_{k=0}^{M} \Pr(L = k) c_k}, \qquad \mathrm{FPT}(\mathbf{a}) = \mathrm{TPF}(\mathbf{a})^{-1}, \end{aligned} \tag{2}$$

where g_k and c_k denote the number of committed tokens and the forward-pass cost when $L = k$, respectively. This general form accommodates both conventional speculative decoding, where a DLM drafter is verified by a separate AR target, and self-speculative DLM decoding such as I-DLM, where proposal and verification are carried out within the same model. For conventional speculative decoding and I-DLM with unit verification cost and one guaranteed committed token per forward pass, it reduces to$\mathrm{TPF} = 1 + \sum_{k=1}^{M} A_k$. In practice, we estimate a_i as one minus the empirical rejection rate at position i on self-generated training data, computed according to each architecture's native verification rule and conditioned on verification reaching that position.

We measure the importance of position i by its marginal effect on FPT under the log-odds parameterization $z_i = \log(a_i/(1-a_i))$. The raw importance, its mean-one normalization, and the resulting proposal objective are summarized as:

$$\begin{aligned} w_i^{\mathrm{raw}} &= -\frac{\partial \,\mathrm{FPT}(\mathbf{a})}{\partial z_i} = -\frac{\partial \,\mathrm{FPT}(\mathbf{a})}{\partial a_i} a_i(1 - a_i), \\ \bar{w}_i &= \frac{M w_i^{\mathrm{raw}}}{\sum_{j=1}^{M} w_j^{\mathrm{raw}} + \epsilon}, \qquad \mathcal{L}_{\mathrm{BRISK}} = \frac{1}{M} \sum_{i=1}^{M} \bar{w}_i \mathcal{L}_i^{\mathrm{prop}}. \end{aligned} \tag{3}$$

The normalized weights preserve the overall loss scale and are treated as fixed coefficients during backpropagation. Recomputing them from the current acceptance profile shifts emphasis toward the positions that most constrain block progress, without requiring an additional model forward pass. Figure 8 illustrates this adaptive behavior during training: the weighting initially emphasizes early proposal positions, which are more likely to truncate the block, and gradually shifts toward later positions as the earlier prefix becomes more reliable.

3.3 Prefix-Conditioned Proposal Selection

Risk–reward learning improves marginal proposals, but their parallel representations are computed before the actual within-block choices $x_{<i}$ are known. A candidate can therefore be plausible in isolation but poorly matched to the selected prefix. At first rejections, the verifier's top-ranked token is already in the base Top-16 in 96.97% of states (Table 7). This suggests that recomputing the backbone is often unnecessary, and what is missing is a lightweight mechanism that conditions candidate selection on the realized prefix. Existing work has introduced lightweight causal heads to condition later decisions on earlier token choices (Huang et al., 2026; Cheng et al., 2026). This restores missing prefix information, but leaves a complementary question: *how should such a selector be supervised for verified progress?*

BRISK addresses both selection and supervision at this interface. The backbone still produces all candidate representations in parallel, while a lightweight corrector sequentially reranks only the existing Top-K candidates using previously selected tokens. We train this selector by distilling the model's own verifier on policy-reached decisions through the first rejection. Thus, selected-token feedback provides the realized-prefix information needed for refinement, while verifier distillation specifies which candidate should be preferred at the decisions that determine usable block progress.

Rejection-aware candidate selection. For proposal position i, we reuse the frozen backbone representation m_i, proposal logits ℓ_i^q, and cached context representation h_c. Let $\mathcal{C}_i = \mathrm{TopK}(q_i, K)$ denote the candidate set obtained from the normalized base proposal distribution q_i. We initialize a

compact state as $u_0 = \mathrm{Init}_\psi(h_c, E(x_0))$, where x_0 is the token immediately preceding the proposal block. Each selected token then conditions the next candidate decision:

$$u_i = f_\psi\left(u_{i-1}, h_c, m_i, E(x_{i-1}), \phi_i\right), \tag{4}$$

where E is the frozen token embedding and ϕ_i contains position and proposal-score features. For each candidate $v \in \mathcal{C}_i$, the corrector predicts a gated low-rank residual on top of the base logit:

$$\tilde{\ell}_{\psi,i}(v) = \ell_i^q(v) + g_{\psi,i}\left\langle W_s u_i, W_c E_{\mathrm{LM}}(v)\right\rangle, \tag{5}$$

where $E_{\mathrm{LM}}(v)$ is the frozen output-vocabulary row, and the projections and gate are learned. We select x_i from refined logits and feed it into the next state update. Let π_i be the resulting proposal law and p_i the frozen causal verifier distribution under the valid prefix and target sampling configuration. Standard speculative sampling accepts with probability $\min\{1, p_i(x_i)/\pi_i(x_i)\}$ and resamples from the normalized positive part $[p_i - \pi_i]_+$ upon rejection, preserving p_i (Leviathan et al., 2023; Chen et al., 2023). Top-K restricts proposals only.

Putting it into practice. The remaining question is how to train the corrector. Existing causal refiners commonly train on reference or teacher-forced continuations (Huang et al., 2026; Cheng et al., 2026). While this provides valid causal supervision, it does not fully specify the learning problem faced by our selector: at deployment, the corrector reranks the proposal model's actual candidates under prefixes formed by earlier proposal choices, and its utility is determined by the verification decisions that are actually reached before the block stops. We therefore align supervision with both the prefix and the reached decisions that determine block progress.

For each recorded proposal block $z_{1:M}$ with a accepted positions, we supervise $\mathcal{R} = \{1, \ldots, \min(a+1, M)\}$. At each $i \in \mathcal{R}$, the model's own verifier distribution under the recorded prefix $z_{<i}$ is restricted and renormalized to the available candidate set $\mathcal{C}_i$. Accepted positions teach candidate choices that preserve verified progress, while the first rejection supplies the preference at the decision that terminates it. Table 5 supports these design choices: richer verifier targets under teacher forcing alone do not yield consistent serving gains, whereas extending policy-reached supervision from accepted-only through the first rejection improves verified progress and throughput. The full objective is given in Appendix C.3.

Efficient system integration. BRISK-DLM adds no backbone evaluation. The corrector reuses cached proposal representations and vocabulary rows, and we fuse candidate construction, recurrent updates, residual scoring, and selection into a GPU-resident execution path. The resulting block is then passed unchanged to the original verifier. Execution details are provided in the Appendix F.

4 Evaluations

Models and benchmarks. We experiment on the 8B and 32B I-DLM checkpoints and use DAPO-Math-17K as training data. Mathematical reasoning is evaluated on GSM8K, MATH-500, Math-Bench, AIME 2024, and AIME 2025, while code generation is evaluated on HumanEval and MBPP. We use stride $N = 8$ by default. We report accuracy for mathematical reasoning and pass@1 for code generation. Efficiency is measured by rejection rate, accepted-prefix length, TPF, and end-to-end throughput. Throughput is evaluated at concurrency $C \in \{1, 2, 4, 8, 16, 32\}$ on NVIDIA H200 and A100 GPUs. Complete training and evaluation details are provided in Appendix C.

Baselines. We compare against the following representative state-of-the-art systems:

- *I-DLM* (Yu et al., 2026): an introspective DLM that uses introspective strided decoding (ISD) to verify previously proposed tokens while advancing new tokens in the same forward pass.
- *LLaDA 2* and *SDAR* (Bie et al., 2026; Cheng et al., 2025): state-of-the-art block-diffusion systems for parallel generation. LLaDA 2.1 accelerates decoding through token editing, while SDAR converts pretrained AR models into blockwise DLMs.
- *DFlash* (Chen et al., 2026): a speculative decoding system that uses a lightweight block-diffusion model to draft multiple tokens in parallel for verification by an AR target. The official repository only released Qwen3-8B drafter for non-thinking generation; we evaluate the same drafter with target thinking enabled and disabled. The main quality and throughput comparisons use thinking mode to retain reasoning-capable operating points.

	LLaDA-2.1 -mini	LLaDA-2.0 -flash	LLaDA-2.1 -flash	SDAR 8B	SDAR 30B-A3B	DFlash OOD-think.	DFlash native NT	I-DLM 8B	BRISK-DLM 8B	Qwen3 8B	I-DLM 32B	BRISK-DLM 32B	Qwen3 32B
Params	*16B*	*100B*	*100B*	*8B*	*30B*	*8B*	*8B*	*8B*	*8B*	*8B*	*32B*	*32B*	*32B*
GSM8K	89.0	**96.1**	92.9	91.7	91.4	93.0	81.7	92.9	92.3	91.8	94.9	<u>95.4</u>	94.7
MATH-500	85.0	97.6	91.0	78.6	77.8	96.6	85.2	96.6	96.8	**98.0**	97.6	96.0	<u>97.8</u>
MathBench	84.2	92.0	90.8	76.9	79.3	92.8	83.6	89.0	89.0	92.8	**95.6**	92.8	<u>95.5</u>
AIME-24	43.3	63.3	63.3	10.0	16.7	76.7	26.7	73.3	<u>80.0</u>	76.7	**83.3**	76.7	76.7
AIME-25	43.3	60.0	63.3	10.0	10.8	63.3	23.3	53.3	56.7	<u>66.7</u>	**80.0**	**80.0**	**80.0**
HumanEval	86.0	**94.5**	90.2	78.7	42.1	75.0	81.1	93.3	90.8	90.9	<u>94.3</u>	93.3	**94.5**
sanitized MBPP	82.1	88.3	88.4	72.0	32.3	75.1	75.1	92.2	88.9	89.9	<u>92.7</u>	**94.2**	92.2

Table 1: Performance on mathematical-reasoning and coding benchmarks. Bold and underlined values indicate the best and second-best results, respectively.

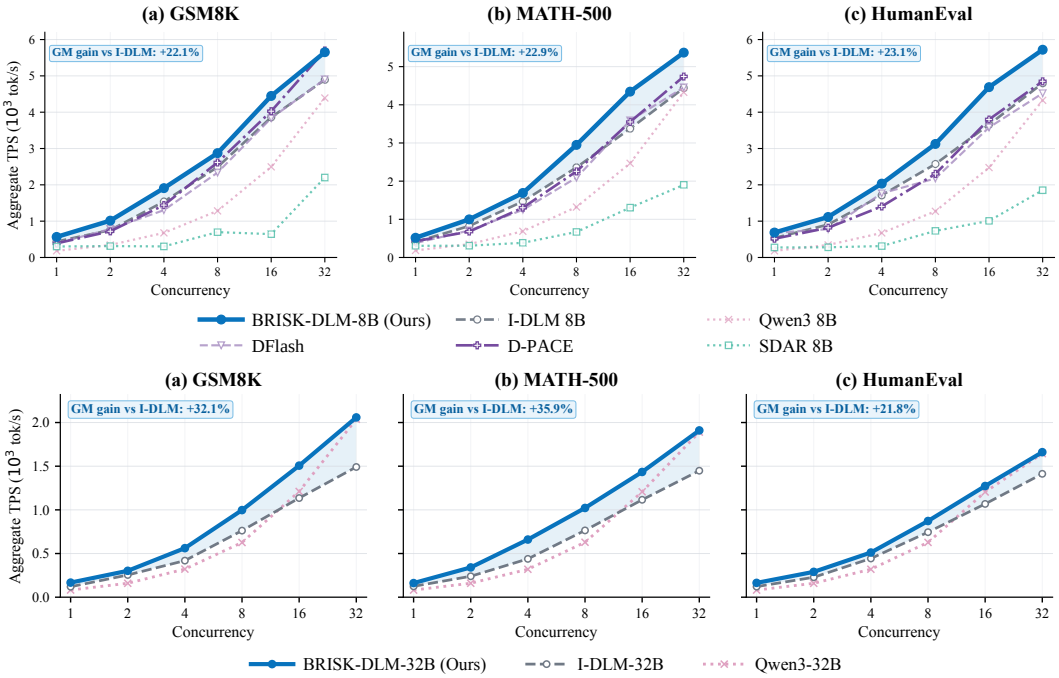

Figure 5: BRISK-DLM improves throughput across model sizes, tasks, and concurrency levels.

- *D-PACE* (Wu et al., 2026): dynamically weights the DFlash drafter's training loss for speculative decoding with AR target model; we use its official training and the same target modes.

4.1 Main Results

BRISK-DLM improves end-to-end throughput across scales. Figure 5 reports serving throughput under matched H200 allocations. BRISK-DLM outperforms the same-scale I-DLM baseline in all 18 workload–concurrency settings at both model scales. At 8B, the geometric-mean gain is 22.7%, with 15.1%–34.7% pointwise gains; task-wise gains are 22.1% on GSM8K, 22.9% on MATH-500, and 23.1% on HumanEval. At 32B, the geometric-mean gain increases to 29.8%, with 15.4%–50.4% pointwise gains; the corresponding task-wise gains are 32.1%, 35.9%, and 21.8%. The larger gain at 32B is accompanied by a greater improvement in verified tokens per forward (37.4% vs. 31.1%), suggesting that increased verified progress contributes to the stronger throughput improvement. These results measure the complete serving path, including the execution overhead of the prefix-conditioned corrector.

BRISK-DLM also compares favorably with external systems. At 8B, relative to the thinking configurations of DFlash and D-PACE, BRISK-DLM achieves 1.29× and 1.27× geometric-mean TPS, respectively. Relative to Qwen3, it achieves 2.28× the TPS at 8B and 1.52× the aggregate TPS at 32B. Complete H200 results, including native NT baselines, are reported in Appendix (Table 10).

BRISK-DLM preserves strong task quality. Table 1 shows that BRISK-DLM remains broadly competitive with state-of-the-art, same-scale DLMs across reasoning and code-generation bench-

Configuration	GSM8K	MATH-500	HumanEval	Avg. gain
I-DLM-8B	2.897	2.794	3.710	–
BRISK-DLM	4.264	3.497	4.534	+31.1%
I-DLM-32B	1.967	2.048	1.990	–
BRISK-DLM	2.805	2.734	2.713	+37.4%

Table 2: TPF for same-scale I-DLM and BRISK pairs.

Training data	W.	Head	Acc. (%)	Reject (%)	TPF	TPS
Open-source	D	Off	83.4	19.02	2.91	1436.2
Self-gen. (all)	D	Off	91.4	13.87	3.41	1893.6
Self-gen. (corr.)	U	Off	96.0	15.55	3.13	1589.0
Self-gen. (corr.)	D	Off	96.2	14.03	3.35	1847.2
Self-gen. (corr.)	**D**	**On**	**96.2**	**11.10**	**3.63**	**1948.8**

Table 3: Training choices and head impact (MATH-500). U and D denote uniform and dynamic weights.

Drafter	$C=1$	$C=2$	$C=4$	$C=8$	$C=16$	$C=32$	GM gain
Original DFlash	878.4	1612.5	2784.2	4531.7	6611.0	10046.0	–
BRISK-DFlash	**925.8**	**2217.1**	**3568.7**	**5755.6**	**7141.1**	**10788.0**	**+18.3%**

Table 4: Training-recipe transfer to DFlash on GSM8K. Aggregate TPS (model tokens/s) on one H200, with exactly 2,048 generated model tokens per request.

marks. At 8B, average accuracy across the five mathematical reasoning benchmarks increases from 81.0 to 83.0. BRISK-DLM reaches 96.8 on MATH-500 and 56.7 on AIME 2025, the latter improving by 3.4 points over the matched I-DLM baseline, while performance on the remaining tasks changes only moderately. At 32B, results vary across tasks but remain strong overall: BRISK-DLM attains 80.0 on AIME 2025 and 93.3 on HumanEval while improving both GSM8K and MBPP.

BRISK-DLM delivers more effective progress per model evaluation. To isolate the algorithmic source of the throughput gains, we measure committed tokens per forward (TPF) (Table 2). On MATH-500, GSM8K, and HumanEval, BRISK-DLM increases 8B TPF from 2.79, 2.90, and 3.71 to 3.50, 4.26, and 4.53, respectively. At 32B, TPF increases from 2.05, 1.97, and 1.99 to 2.73, 2.81, and 2.71. These correspond to geometric-mean gains of 31.1% at 8B and 37.4% at 32B over matched I-DLM. The larger gains in TPF than in wall-clock throughput are expected: BRISK-DLM commits more output per expensive model evaluation, while the prefix-conditioned corrector introduces additional lightweight work. The end-to-end gains above show that, after this overhead is included, the increase in effective block progress still translates into substantial serving acceleration.

4.2 Performance Breakdown and Ablation Studies

Performance Breakdown. Table 3 separates the contributions of self-generated data, dynamic position weighting, and prefix-conditioned selection. The results show that data selection and position weighting address different failure modes. With dynamic weighting fixed, training on self-generated trajectories yields higher TPF and throughput than training only on correctness-filtered trajectories, but reduces accuracy by 4.8 points. Correctness filtering recovers task quality at only a 2.5% throughput cost. Replacing self-generated trajectories with open-source data is substantially worse: accuracy drops by 12.8 points, rejection rises from 14.03% to 19.02%, and throughput decreases by 22.3%. This highlights the need for training on states induced by the decoding policy.

Holding correctness-filtered self-generated data fixed, dynamic weighting reduces rejection by 1.52 points relative to uniform weighting and improves TPF and throughput by 7.0% and 16.3%, respectively, without reducing accuracy. Thus, policy-matched data determines which states the model learns from, while marginal-progress weighting determines where learning capacity is spent within those states. The prefix-conditioned corrector provides a further inference-time gain on top of the trained proposal model. With the backbone checkpoint fixed, enabling the corrector reduces rejection from 14.03% to 11.10%, increases TPF from 3.35 to 3.63 (+8.4%), and raises throughput from 1847.2 to 1948.8 tok/s (+5.5%).

Cross-backbone transfer. To test whether proposal learning is specific to I-DLM, we apply the BRISK training recipe to the DFlash drafter while keeping the Qwen3-8B target, serving path, and numeric path fixed. Under native non-thinking GSM8K requests, Table 4 shows a 18.3% geometric-mean TPS gain, with improvements at all six concurrency levels.

Residual proposal errors and repair. Even after proposal training, many first rejections occur despite a verifier-preferred alternative remaining available within the restricted candidate support: the verifier's top-ranked token is already contained in the base Top-16 at 96.97% of factual first-

Supervision	Mean TPS / request TPF ↑		
	$C = 1$	$C = 8$	$C = 16$
No corrector	91.6/3.42	704.8/3.46	1055.1/3.38
TF token labels	96.1/3.68	753.6/3.74	1115.8/3.68
TF verifier targets	95.4/3.66	743.1/3.71	1152.6/3.71
Policy-reached, accepted only	98.1/**3.76**	744.4/3.74	1123.1/3.73
BRISK: through first rejection	**98.2/3.76**	**780.3/3.76**	**1198.7/3.79**

Table 5: Selector supervision at $N = 8$ on A100. Cells show mean TPS / mean request TPF. TF denotes teacher forcing.

K	First breakers Coverage/ mass (%)	Natural EOS Prefix/ TPF
4	80.12/80.03	4.83/4.12
8	91.50/91.40	4.84/4.12
16	96.56/96.49	**5.04/4.36**
32	**98.71/98.67**	4.79/4.07

Table 6: Top-K sensitivity on MATH-500. Coverage and mass describe candidate availability, not an oracle rescue rate.

breaker states. Under same-state replay, prefix-conditioned refinement rescues 28.77 decisions per 100 first breakers; separately, it introduces 1.60 rejections per 100 accepted positions. Together, the diagnostics support the view that a substantial part of the residual error lies in selecting among available candidates under the realized prefix. Appendix B provides the full analysis, including replay definitions, position-wise intervention and rescue, and whole-block verifier-likelihood outcomes.

What should supervise the candidate selector? Table 5 asks two questions: *under which prefix should the selector learn verifier preferences*, and *which reached decisions should be supervised*. We first ask whether richer targets are sufficient under teacher forcing. The teacher-forced controls reflect reference-conditioned supervision used by representative causal refiners (Huang et al., 2026; Cheng et al., 2026), without reimplementing those systems. Replacing hard token labels with soft verifier targets does not yield consistent serving gains, showing that target richness alone is insufficient. BRISK instead trains on policy-reached candidate decisions with verifier targets, and achieves 780.3 and 1198.7 TPS at $C = 8$ and 16, outperforming both teacher-forced controls at these loads.

We next ask how far supervision should extend along a policy-reached trajectory. Accepted-only targets teach the selector how to preserve an already successful prefix, but omit the first reached decision that terminates usable block progress. BRISK retains this position and distills the verifier's preference over its available candidates. Relative to accepted-only supervision, BRISK improves TPS from 744.4 to 780.3 at C=8 and from 1123.1 to 1198.7 at C=16.

Top-K sensitivity analysis. We vary $K \in 4, 8, 16, 32$ with the 8B BRISK backbone fixed. At head-off first rejections, $K = 16$ already contains the verifier's top-1 choice in 96.56% of states and captures 96.49% of its probability mass (Table 6); increasing to $K = 32$ adds only about 2.2 points, while ranks 17–32 are selected in just 0.07% of cases. On MATH-500, $K = 16$ achieves the highest measured accepted-prefix length and TPF, and the highest A100 throughput at all three evaluated concurrency levels, $C \in \{1, 8, 16\}$. We use $K = 16$ as the default.

Impact of Stride Size. We sweep $N \in 4, 8, 16$ on MATH-500 to test whether BRISK-DLM's gains depend on the default stride. Across all strides and concurrency levels, BRISK-DLM consistently outperforms matched I-DLM, with geometric-mean TPS gains of 14.7%, 22.9%, and 20.2% for $N = 4, 8, 16$, respectively (Appendix Table 13). Thus, BRISK's benefit is robust to proposal width.

System optimization. For the same correcter checkpoint, a reference implementation improves TPF but reduces end-to-end TPS relative to head-off decoding. Optimized execution instead delivers an 8.2–9.5% TPS gain over head-off decoding on A100 across $C \in \{1, 8, 32\}$ (Appendix Table 15).

5 Conclusion

BRISK-DLM aligns parallel proposal learning and selection with the prefix-dependent progress that ultimately determines decoding efficiency. It combines correctness-filtered self-generated training, marginal-progress weighting, and prefix-conditioned selection while preserving efficient parallel backbone computation. Across AR–DLM and DLM decoding settings, BRISK-DLM consistently improves generation efficiency by 22.7% at 8B and 29.8% at 32B, while maintaining task quality.

AI USE STATEMENT

In this work, we used generative AI tools to assist with English translation and provide feedback on the presentation of the research methodology. We have not used generative AI tools to generate experimental data, formulate mathematical claims, or interpret experimental results, and the remaining required disclosure tasks are not applicable to this work. Additionally, we used generative AI tools to draft and edit portions of the paper, suggest its structure, summarize and identify relevant literature. We have reviewed all AI-assisted work and manually verified the technical claims, citations, and reported results. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work studies methods for improving the decoding efficiency of diffusion language models and does not involve human subjects or the collection of private user data. Our experiments use established public benchmarks and model-generated training trajectories. As with general-purpose language models, improvements in generation efficiency may broaden both beneficial and potentially harmful uses; our work does not introduce new capabilities or application domains beyond those of the underlying models.

REPRODUCIBILITY STATEMENT

We provide the algorithmic details, training objectives, model and decoding configurations, datasets, evaluation protocols, and serving setups needed to reproduce our results in the main paper and appendices. The appendices additionally document implementation details, hyperparameters, ablation settings, hardware configurations, and measurement procedures for the reported quality, TPF, and throughput results.

REFERENCES

Marianne Arriola, Aaron Gokaslan, Justin T. Chiu, Zhihan Yang, Zhixuan Qi, Jiaqi Han, Subham Sekhar Sahoo, and Volodymyr Kuleshov. Block diffusion: Interpolating between autoregressive and diffusion language models. *arXiv preprint arXiv:2503.09573*, 2025. URL https://arxiv.org/abs/2503.09573.

Jacob Austin, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne Van Den Berg. Structured denoising diffusion models in discrete state-spaces. In *Advances in Neural Information Processing Systems*, volume 34, pp. 17981–17993, 2021. URL https://proceedings.neurips.cc/paper_files/paper/2021/hash/958c530554f78bcd8e97125b70e6973d-Abstract.html.

Tiwei Bie, Maosong Cao, Kun Chen, Lun Du, Mingliang Gong, Zhuochen Gong, Yanmei Gu, Jiaqi Hu, Zenan Huang, Zhenzhong Lan, et al. LLaDA 2.0: Scaling up diffusion language models to 100b. *arXiv preprint arXiv:2512.15745*, 2025. URL https://arxiv.org/abs/2512.15745.

Tiwei Bie, Maosong Cao, Xiang Cao, Bingsen Chen, Fuyuan Chen, Kun Chen, Lun Du, Daozhuo Feng, Haibo Feng, Mingliang Gong, et al. LLaDA 2.1: Speeding up text diffusion via token editing. *arXiv preprint arXiv:2602.08676*, 2026. URL https://arxiv.org/abs/2602.08676.

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774*, 2024. URL https://arxiv.org/abs/2401.10774.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023. URL https://arxiv.org/abs/2302.01318.

Jian Chen, Yesheng Liang, and Zhijian Liu. DFlash: Block diffusion for flash speculative decoding. *arXiv preprint arXiv:2602.06036*, 2026. URL https://arxiv.org/abs/2602.06036.

Shuang Cheng, Yihan Bian, Dawei Liu, Linfeng Zhang, Qian Yao, Zhongbo Tian, Wenhai Wang, Qipeng Guo, Kai Chen, Biqing Qi, et al. SDAR: A synergistic diffusion-autoregression paradigm for scalable sequence generation. *arXiv preprint arXiv:2510.06303*, 2025. URL https://arxiv.org/abs/2510.06303.

Xin Cheng, Xingkai Yu, Chenze Shao, Jiashi Li, Yunfan Xiong, et al. DSpark: Confidence-scheduled speculative decoding with semi-autoregressive generation. *arXiv preprint arXiv:2607.05147*, 2026. URL https://arxiv.org/abs/2607.05147.

Justin Deschenaux and Caglar Gulcehre. Beyond autoregression: Fast llms via self-distillation through time. *arXiv preprint arXiv:2410.21035*, 2024. URL https://arxiv.org/abs/2410.21035.

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

Yifeng Gao, Ziang Ji, Yuxuan Wang, Biqing Qi, Hanlin Xu, and Linfeng Zhang. Self speculative decoding for diffusion large language models, 2025.

Fabian Gloeckle, Badr Youbi Idrissi, Baptiste Roziere, David Lopez-Paz, and Gabriel Synnaeve. Better & faster large language models via multi-token prediction. *arXiv preprint arXiv:2404.19737*, 2024. URL https://arxiv.org/abs/2404.19737.

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. *arXiv preprint arXiv:2410.17891*, 2024. URL https://arxiv.org/abs/2410.17891.

Ligong Han, Hao Wang, Han Gao, Kai Xu, and Akash Srivastava. S2d2: Fast decoding for diffusion llms via training-free self-speculation. *arXiv preprint arXiv:2603.25702*, 2026.

Shijing Hu, Jingyang Li, Xingyu Xie, Zhihui Lu, Kim-Chuan Toh, and Pan Zhou. GRIFFIN: Effective token alignment for faster speculative decoding. *arXiv preprint arXiv:2502.11018*, 2025. URL https://arxiv.org/abs/2502.11018.

Jianuo Huang, Yaojie Zhang, Qituan Zhang, Hao Lin, Hanlin Xu, and Linfeng Zhang. Domino: Decoupling causal modeling from autoregressive drafting in speculative decoding. *arXiv preprint arXiv:2605.29707*, 2026. URL https://arxiv.org/abs/2605.29707.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning*, pp. 19274–19286. PMLR, 2023. URL https://proceedings.mlr.press/v202/leviathan23a.html.

Xiang Lisa Li, John Thickstun, Ishaan Gulrajani, Percy Liang, and Tatsunori B. Hashimoto. Diffusion-lm improves controllable text generation. In *Advances in Neural Information Processing Systems*, volume 35, pp. 4328–4343, 2022. URL https://openreview.net/forum?id=3s9IrEsjLyk.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE: Speculative sampling requires rethinking feature uncertainty. *arXiv preprint arXiv:2401.15077*, 2024. URL https://arxiv.org/abs/2401.15077.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE-3: Scaling up inference acceleration of large language models via training-time test. *arXiv preprint arXiv:2503.01840*, 2025. URL https://arxiv.org/abs/2503.01840.

Aaron Lou, Chenlin Meng, and Stefano Ermon. Discrete diffusion modeling by estimating the ratios of the data distribution. In *Proceedings of the 41st International Conference on Machine Learning*, 2024. URL https://arxiv.org/abs/2310.16834.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, et al. SpecInfer: Accelerating generative large language model serving with tree-based speculative inference and verification. *arXiv preprint arXiv:2305.09781*, 2023. URL https://arxiv.org/abs/2305.09781.

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. URL https://arxiv.org/abs/2502.09992.

Subham Sekhar Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T. Chiu, Alexander Rush, and Volodymyr Kuleshov. Simple and effective masked diffusion language models. In *Advances in Neural Information Processing Systems*, volume 37, pp. 130136–130184, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/f59ba58cd21e7648528a0d1ab9df9064-Abstract-Conference.html.

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. URL https://arxiv.org/abs/2606.06712.

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*, 2025. URL https://arxiv.org/abs/2505.22618.

Shutong Wu and Jiawei Zhang. Free draft-and-verification: Toward lossless parallel decoding for diffusion large language models. *arXiv preprint arXiv:2510.00294*, 2025.

Tianyu Wu, Yu Yao, Zhenting Qi, Han Zheng, Zhuohan Wang, Haoran Ma, Lawrence Liao, Himabindu Lakkaraju, Ju Li, and Yilun Du. D-PACE: Dynamic position-aware cross-entropy for parallel speculative drafting. *arXiv preprint arXiv:2605.18810*, 2026. URL https://arxiv.org/abs/2605.18810.

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. URL https://arxiv.org/abs/2508.15487.

Yifan Yu, Yuqing Jian, Junxiong Wang, Zhongzhu Zhou, Donglin Zhuang, Xinyu Fang, Sri Yanamandra, Xiaoxia Wu, Qingyang Wu, Shuaiwen Leon Song, Tri Dao, Ben Athiwaratkun, James Zou, Fan Lai, and Chenfeng Xu. Introspective diffusion language models. *arXiv preprint arXiv:2604.11035*, 2026. URL https://arxiv.org/abs/2604.11035.

A Detailed Related Work

Diffusion language models. Diffusion language models (DLMs) generate text by iteratively recovering tokens from partially masked states, offering a route to parallel generation beyond left-to-right autoregression. Early discrete diffusion formulations established principled denoising objectives in discrete spaces (Austin et al., 2021), while Diffusion-LM (Li et al., 2022), masked diffusion language models (Sahoo et al., 2024), and SEDD (Lou et al., 2024) developed these ideas for language generation. Recent systems have substantially expanded the scale and decoding design space of DLMs, including LLaDA (Nie et al., 2025), Block Diffusion (Arriola et al., 2025), DREAM (Ye et al., 2025), SDAR (Cheng et al., 2025), and LLaDA 2 (Bie et al., 2025; 2026). A complementary line converts pretrained autoregressive models into DLMs to reduce training cost (Gong et al., 2024; Deschenaux & Gulcehre, 2024; Fu et al., 2025; Su et al., 2026). In contrast to iterative denoising methods, I-DLM uses introspective strided decoding (ISD) to generate new tokens while verifying previously proposed tokens in the same forward pass (Yu et al., 2026). BRISK-DLM builds on this setting: our focus is not a new DLM architecture or a new denoising schedule, but improving the proposal acceptance profile that determines how much ISD parallelism is realized in practice.

Parallel drafting and acceptance-aware training. Speculative decoding accelerates autoregressive generation by verifying a fast model's draft with a target model while preserving the target distribution (Leviathan et al., 2023; Chen et al., 2023). Subsequent work improves drafting through multiple heads (Cai et al., 2024), feature-level proposals (Li et al., 2024; 2025), tree verification (Miao et al., 2023), and multi-token prediction (Gloeckle et al., 2024). Parallel block drafters such as DFlash (Chen et al., 2026) make accepted-prefix length especially important because all draft positions are proposed jointly. Most closely related, D-PACE uses prefix-dependent acceptance to derive dynamic position weights(Wu et al., 2026). BRISK shares the observation that proposal positions have unequal effects on accepted progress, but differs in three respects. First, our objective accounts for forward-pass cost in addition to accepted progress. Second, we treat prefix dependence as an inference-time selection problem as well as a training-time weighting problem, using a lightweight corrector to rerank parallel candidates under the realized proposal prefix. Third, BRISK targets the proposal–verification interface internal to the DLM itself rather than a separate diffusion-drafter/AR-target pipeline. GRIFFIN instead mitigates training–decoding token misalignment through token alignment and loss masking (Hu et al., 2025). These methods address a separate drafter–target configuration. Our work studies the proposal and verification paths internal to DLM, and couples acceptance-aware optimization with correct self-generated trajectories and proposal-side block consistency.

B Mechanism Analysis

Setup. We analyze 100 GSM8K and 100 MATH-500 requests at $N = 8$, with seven speculative positions and Top-16 support. The factual head-off policy uses the Online+Dynamic checkpoint; the head-enabled policy uses the same backbone. Local substitutions are evaluated at factual head-off states. Whole-block replay starts both selectors from shared producer states. These diagnostic runs are separate from throughput measurement; Table 7 summarizes the main findings.

B.1 First-breaker repair and block compatibility

A first breaker is the first speculative token rejected by ISD. The analysis contains 62,079 factual first breakers. Candidate coverage in Table 7 asks whether the clean verifier's full-vocabulary top-ranked token is in the base Top-16. Coverage describes available support; it is not an oracle acceptance rate.

Same-state replay. We fix the factual prefix, backbone representations, Top-16 support, full-vocabulary-normalized base q_i, clean verifier p_i, and acceptance draw u_i, changing only the token selected at the intervention position. The evaluated diagnostic rule is

$$\alpha_i(x) = \min\{1, p_i(x)/q_i(x)\}, \qquad J_i(x) = \mathbf{1}[u_i < \alpha_i(x)]. \tag{6}$$

A rescue is $J_i(x^{\text{base}}) = 0$, $J_i(x^{\text{head}}) = 1$ at a factual first breaker. A regression is the reverse outcome at a factual accepted position. Their denominators differ, so subtracting their displayed conditional rates is not a net rescue rate. Table 7 adds the intervention rate; Figure 6 provides

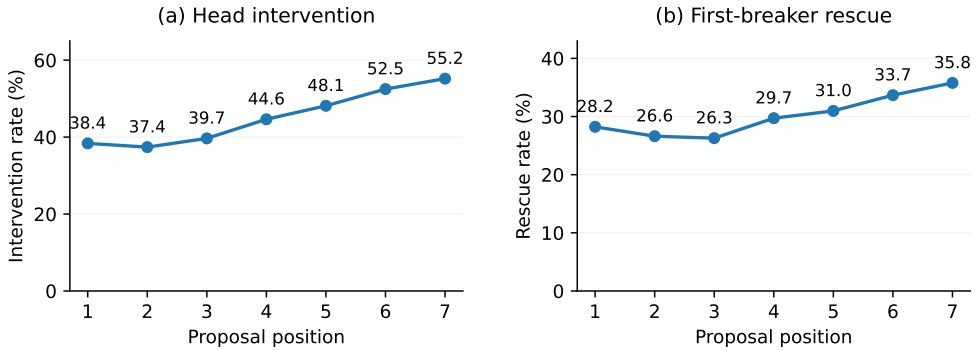

Figure 6: **First-breaker repair by proposal position.** (a) Head intervention and (b) same-draw rescue, each as a fraction of factual first breakers at that position. Intervention rises from 38.4% at position 1 to 55.2% at position 7, while rescue rises from 28.2% to 35.8%. The overall upward trend suggests that later proposals offer more opportunity for prefix-conditioned correction. The rescue curve shows that the head's additional interventions translate into more accepted local decisions, rather than merely changing token choices.

Table 7: Mechanism diagnostics for residual selection errors. Candidate-availability and local-substitution statistics are equal-request averages over 100 GSM8K and 100 MATH-500 requests.

Quantity	Population / unit	Value
Candidate availability and same-state repair		
Verifier top-1 in base-q Top-16	First breakers	96.97%
Corrector changes selected token	First breakers	43.07%
Corrector rescues rejection	Per 100 first breakers	28.77
Corrector introduces rejection	Per 100 accepted positions	1.60
Whole-block verifier preference		
Mean $\Delta\mathcal{L}$	Nats per seven-token block	+7.63
Refined block has higher likelihood	Paired blocks	33.6%
Equal recorded likelihood	Paired blocks	52.3%
Base block has higher likelihood	Paired blocks	14.2%

position-wise results. Events from the same request are correlated, and local summary rates use equal-request averaging rather than treating every event as an independent request.

Rescue and regression quantify candidate substitutions under this fixed-base-q diagnostic rule. A gain in $p_i(x)/q_i(x)$ need not by itself imply a gain in $p_i(x)$; clean-verifier likelihood is therefore assessed separately.

Whole-block replay. The head selects all seven positions sequentially. Base and head blocks are then scored in separate clean-verifier evaluations from their shared producer prefix c:

$$\mathcal{L}(b \mid c) = \sum_{i=1}^{7} \log p(x_i \mid c, x_{<i}), \qquad \Delta\mathcal{L} = \mathcal{L}(b^{\text{head}} \mid c) - \mathcal{L}(b^{\text{base}} \mid c). \tag{7}$$

Each block is scored under its own selected predecessor path. The mean gain uses equal-request averaging; outcome fractions pool 109,197 paired blocks. Improved, tied, and regressed blocks are all retained in Table 7. Equal likelihood does not establish token identity, and higher likelihood does not establish semantic or final-answer correctness.

B.2 Local repetition

Figure 7 compares equal-request repeat prevalence over verified positions and first breakers. Adjacent repetition increases from 2.22% to 10.02%, whereas non-adjacent repetition changes from

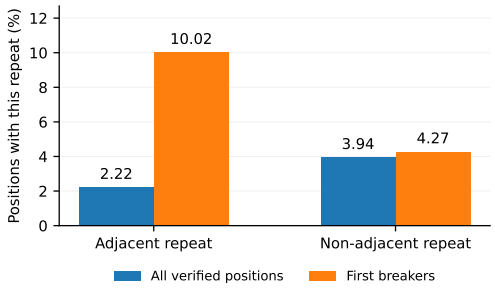

Figure 7: **Repeat prevalence among verified positions and first breakers.** Rates are equal-request averages.

3.94% to 4.27%. This is an association in the trained head-off policy, not evidence that repeated tokens are necessarily invalid or are the primary cause of rejection.

Under the same-draw diagnostic rule, the head rescues 57.32% of adjacent-repeat first breakers and 34.80% of non-adjacent-repeat first breakers. Paired whole-block replay shows adjacent repeats decreasing from 0.404 to 0.101 tokens per block; non-adjacent repeats change from 0.348 to 0.364. The selective change is consistent with prefix-dependent refinement rather than uniform repetition suppression. These paired-block counts are not the same quantity as the rejection-conditioned percentages in the earlier baseline taxonomy.

C Implementation Details

C.1 Brisk Backbone

We use DAPO-Math-17K as the source of training prompts and reference answers. For each prompt, we generate candidate responses once from the initial I-DLM checkpoint using the deployed decoding procedure. We generate the training data offline rather than continuously regenerating it during training to mitigate potential overfitting. We retain only trajectories whose final answers match the reference under normalized answer matching, reducing the risk of reinforcing incorrect reasoning paths. The resulting correctness-filtered self-generated trajectories are used for continued training. All runs use the original I-DLM masking construction with stride $N = 8$, yielding $M = N - 1 = 7$ proposal positions. The position-risk weights are applied to these seven proposal losses according to Eq. equation 3. Controlled ablations use the same initialization, training-token budget, and optimization schedule.

We estimate a_i from position-wise rejection statistics on the self-generated training data, following the native introspective strided decoding (ISD) verification rule. At position i, the rejection rate is the number of proposals rejected at that position divided by the number of blocks in which verification reaches it; a_i is one minus this rate. Positions after the first rejection are excluded from the statistics because they are not reached by verification.

Using this acceptance profile, the general renewal-reward formulation in Eq. equation 2 specializes to

$$\mathrm{TPF}(\mathbf{a}) = \frac{2 + \sum_{k=1}^{M-1} A_k}{2 - A_M}, \qquad M = N - 1. \tag{8}$$

This corresponds to $g_k = 2 + \min(k, M - 1)$, with $c_k = 2$ for $k < M$ and $c_M = 1$. In all experiments, we use Eq. equation 8 to compute $\mathrm{FPT}(\mathbf{a}) = 1/\mathrm{TPF}(\mathbf{a})$, from which the raw position sensitivities and normalized weights are obtained using Eq. equation 3. The normalized weights are treated as fixed coefficients during backpropagation. Table 8 summarizes the remaining training and inference settings.

Table 8: Training and inference configurations.

Setting	8B	32B
Training data	All valid	All valid
Optimizer / LR	AdamW / 2×10^{-5}	AdamW / 10^{-5}
Global batch size	32	32
Sequence length	4096	8192
Training epochs	5	5
Training GPUs	8×H20	8×H20
Tensor parallelism	1	4
Precision	BF16	BF16
CUDA / PyTorch	12.1 / 2.5.1	12.1 / 2.5.1
SGLang / Triton	0.4.6 / 3.1.0	0.4.6 / 3.1.0

C.2 Adaptive Position-Weight Dynamics

Figure 8 shows how the normalized position weights evolve during training. Early proposal positions receive larger weights initially, reflecting their stronger influence on prefix survival. As their acceptance improves, the weighting gradually shifts toward later positions, which become the remaining bottlenecks to verified progress. This behavior illustrates how the proposed objective adapts its optimization focus as the acceptance profile evolves.

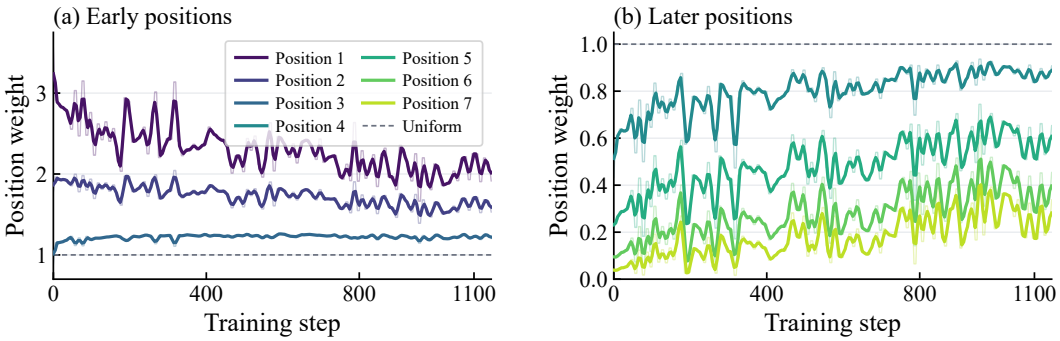

Figure 8: Evolution of the seven normalized proposal-position weights in our experimental setting. Early positions (1–3) and later positions (4–7) are separated for clarity. Faint curves show logged values, solid curves show a centered three-point moving average, and the dashed line denotes uniform weighting ($\bar{w}_i = 1$). Emphasis initially favors early positions and gradually shifts toward later positions as the acceptance profile evolves. The panels use different vertical scales.

C.3 Candidate-Refinement Head Details

We augment the frozen BRISK backbone with a lightweight recurrent head that refines the base proposal scores at the $M = N - 1 = 7$ speculative positions. At position j, the head operates on the ordered candidate set

$$\mathcal{S}_j = \mathrm{TopK}(q_j), \qquad K = 16, \tag{9}$$

where q_j is the base proposal distribution. The head only reranks these candidates and never expands the candidate set.

The recurrent state is initialized from the anchor hidden state and anchor token embedding. At each proposal position, its input concatenates a 64-dimensional projection of the MASK hidden state, a 64-dimensional projection of the previous-token embedding, a 16-dimensional position embedding, and a 16-dimensional projection of five base-proposal confidence features. These features contain the Top-1–Top-2 and Top-1–Top-4 logit gaps, indicators of whether the previous token is the Top-1 token or belongs to the Top-4, and the Top-1–previous-token logit gap. The resulting 160-dimensional input is processed by a bias-free minimal GRU with a 128-dimensional state.

For each candidate $v_{j,k} \in \mathcal{S}_j$, the head predicts a gated, rank-64 residual over the base logit,

$$d_{j,k} = \langle W_s s_j,\, W_c E_{\text{out}}(v_{j,k}) \rangle\,, \tag{10}$$

$$\gamma_j = \sigma(\alpha)\,\sigma\big(w_g^\top [s_j; \phi_j] + b_g\big)\,, \tag{11}$$

$$\widetilde{\ell}_{j,k} = \ell^q_{j,k} + \gamma_j d_{j,k}, \tag{12}$$

where E_{out} denotes the frozen LM-head embedding and ϕ_j denotes the confidence features. The input embedding, LM head, and BRISK backbone remain frozen. The refinement head contains 1.89M trainable parameters.

We train the head on the correctness-filtered self-generated trajectories described above. Previous-token feedback follows the actual base proposal trajectory. The target at each reached position is the verifier distribution restricted to and renormalized over the same Top-16 candidate set. If the first rejected proposal occurs at position b, the loss covers the accepted prefix and position b; for an all-pass event, it covers all seven proposal positions. Positions after the first rejection are excluded.

Let $p_{\mathcal{S},j}$ be the verifier distribution, $q_{\mathcal{S},j}$ the base distribution, and $r_j = \mathrm{softmax}(\widetilde{\ell}_j)$ the refined distribution, all conditional on $\mathcal{S}_j$. The distribution r_j is used for training; verification uses the actual proposal law induced by the configured inference selection rule. The training objective is

$$\begin{aligned} \ell_{e,j} &= \mathrm{CE}(p_{\mathcal{S},e,j}, r_{e,j}) \quad + \beta D_{\mathrm{KL}}(q_{\mathcal{S},e,j} \,\|\, r_{e,j}), \\ \mathcal{L}_{\text{head}} &= \frac{\sum_{e,j} m_{e,j} \ell_{e,j}}{\sum_{e,j} m_{e,j}}, \qquad \beta = 0.01. \end{aligned} \tag{13}$$

Here CE is cross-entropy and $m_{e,j}$ is the reached-position mask. We take one global mean over valid positions, use uniform position weights, and apply no additional weight to the first rejected position. Table 9 summarizes the implementation settings.

Table 9: Candidate-refinement head configuration.

Setting	Value
Proposal positions / candidate size	7 / 16
Recurrent cell / state dimension	minimal GRU / 128
Hidden and token projection dimensions	64 / 64
Position / feature projection dimensions	16 / 16
Candidate residual rank	64
Trainable parameters	1.89M
Training data	326 prompts / 165,515 events
Supervision	accepted prefix + first rejection
Optimizer / learning rate	AdamW / 3×10^{-4}
Anchor KL coefficient	0.01
Trainable / forward precision	FP32 / BF16
Training hardware	1×A100

C.4 Algorithm

C.5 Evaluation Details

BRISK-DLM and I-DLM use identical serving configurations, with inference implemented in SGLang and block selection in Triton. Main-text comparisons use aggregate TPS; Appendix D.2 additionally reports median request TPS and the complete hardware configuration. Figure annotations summarize paired throughput ratios using the geometric mean across concurrency.

D Detailed Serving Results

D.1 Protocols and comparison scope

Main serving results use aggregate generated-model-token throughput, equal to generated model tokens divided by elapsed wall-clock time. Each request generates exactly 2,048 model tokens and

Algorithm 1 Overview of BRISK-DLM training and inference.

(a) Training

Require: Initial DLM θ_0, prompt–answer pairs $\mathcal{D}$

$\mathcal{D}_{\text{on}} \leftarrow \emptyset$, $\theta \leftarrow \theta_0$
for $(c, y) \in \mathcal{D}$ **do**
    Generate responses $\mathcal{S}_c$ with θ_0
    Add correct responses in $\mathcal{S}_c$ to $\mathcal{D}_{\text{on}}$
while not converged **do**
    Sample $\mathcal{B} \subset \mathcal{D}_{\text{on}}$
    Compute proposal losses and accept profile $\mathbf{a}$
    Compute $\bar{\mathbf{w}}$ from $\mathbf{a}$ using Eq. (3)
    Update θ with $\mathcal{L}_{\text{BRISK}}$
Verifier-distilled corrector
Freeze θ and collect policy-reached blocks
Obtain verifier targets on reached positions
Train corrector ψ using Eq. equation 13
return θ, ψ

(b) Inference

Require: Trained DLM θ, corrector ψ, prefix c

Compute $\{m_i, \ell_i^q, q_i\}_{i=1}^{M}$ and h_c **in parallel**
$x_0 \leftarrow$ token preceding the proposal block
$u_0 \leftarrow \text{Init}_\psi(h_c, E(x_0))$
for $i = 1, \ldots, M$ **do**
    $\mathcal{C}_i \leftarrow \text{TopK}(q_i, K)$
    $u_i \leftarrow f_\psi(u_{i-1}, h_c, m_i, E(x_{i-1}), \phi_i)$
    Refine logits on $\mathcal{C}_i$ using Eq. (5)
    $x_i \leftarrow \text{Select}(\mathcal{C}_i, \tilde{\ell}_{\psi,i})$
Verify $x_{1:M}$ using (p_i, π_i) as in §3.3
Commit the accepted prefix and correction; follow the ISD stopping rule
return updated prefix

concurrency is $C \in \{1, 2, 4, 8, 16, 32\}$. Geometric means average paired ratios, not raw TPS values. Quality is assessed separately using the task-specific metrics in Table 1; a fixed-output throughput benchmark is not an accuracy or time-to-correct-answer measurement.

H200 comparisons use one GPU/TP1 for 8B I-DLM, BRISK, and Qwen3, and two GPUs/TP2 for the corresponding 32B models. LLaDA-2.0/2.1-flash uses four H200s/TP4; other listed H200 systems use one H200/TP1. The panel therefore compares native serving instances. DFlash OOD-thinking and native NT are separate operating modes. D-PACE uses the same target, runtime, and numeric path as the corresponding DFlash operating point, replacing only the drafter. The system optimization study in Appendix F uses a different client-visible-token counter and includes prefill exposures in its TPF denominator.

D.2 Complete H200 Results

Table 10 reports the aggregate TPS values used in the main figure. The 8B geometric-mean gains relative to I-DLM are 22.1%, 22.9%, and 23.1% on GSM8K, MATH-500, and HumanEval; the 32B gains are 32.1%, 35.9%, and 21.8%.

D.3 Tokens per Forward

Table 11 reports C=1 TPF over the same fixed 2,048-token measurement window. For DFlash and D-PACE-DFlash, TPF counts target-verification cycles and excludes draft-model forward cost, so it is a mechanism diagnostic rather than a complete measure of end-to-end cost. Qwen3 has TPF 1 by autoregressive definition.

Table 10: **Complete H200 serving results.** GPU/TP denotes the number of GPUs and tensor-parallel degree; each result cell reports aggregate TPS / median request TPS in generated model tokens/s. Rankings use the aggregate-TPS component: * denotes the best result in each column, underlining denotes the second-best result, and gray rows denote BRISK-DLM.

(a) GSM8K Method	GPU/TP	$C=1$	$C=2$	$C=4$	$C=8$	$C=16$	$C=32$
I-DLM-8B	1/1	438.4/438.4	752.8/450.2	1539.0/403.5	2489.4/340.4	3858.4/293.9	4893.2/181.8
BRISK-DLM-8B (Ours)	1/1	566.1/566.1	1014.3/517.1	1909.0/464.9	2873.3/408.3	4447.6/337.1	5652.0/192.4
DFlash OOD-thinking	1/1	392.5/392.5	782.2/447.6	1303.1/371.5	2337.2/330.1	3835.1/297.1	4906.2/195.9
DFlash native NT	1/1	878.4/878.4*	1612.5/898.3	2784.2/887.0	4531.7/823.6	6611.0/696.3	10046.0/454.9
D-PACE-DFlash OOD-thinking	1/1	389.1/389.1	721.0/427.3	1441.5/384.0	2616.9/354.8	4026.7/323.0	5696.7/207.7
D-PACE-DFlash native NT	1/1	803.2/803.2	1976.2/1107.3*	3467.8/1000.4*	5604.2/838.8*	8451.3/619.3*	11074.1/441.5*
Qwen3-8B	1/1	183.0/183.0	344.1/172.1	677.6/169.4	1282.8/160.4	2495.0/156.1	4391.1/137.4
SDAR-8B	1/1	302.6/302.6	313.7/240.6	304.3/101.2	698.0/125.2	642.9/62.1	2201.0/86.7
LLaDA-2.1-mini (16B)	1/1	476.9/476.9	501.0/388.8	213.5/60.2	453.9/67.1	950.2/68.0	1367.6/46.4
SDAR-30B-A3B	1/1	309.1/309.1	276.0/187.3	85.1/22.7	173.5/23.2	406.0/27.5	767.6/25.9
LLaDA-2.0-flash (100B)	4/4	686.4/686.4	863.1/778.1	919.1/280.9	684.7/103.6	577.2/40.9	1160.1/37.4
LLaDA-2.1-flash (100B)	4/4	852.6/852.6	1429.2/714.7	1168.8/388.6	1494.5/225.9	1143.4/78.0	1264.8/40.7
I-DLM-32B	2/2	119.9/119.9	254.3/127.8	418.8/113.1	762.3/99.2	1135.1/75.4	1491.0/49.8
BRISK-DLM-32B (Ours)	2/2	167.1/167.1	302.1/158.5	561.5/158.2	997.3/146.5	1506.5/105.3	2058.5/72.3
Qwen3-32B	2/2	82.2/82.2	159.5/79.8	321.5/80.4	626.4/78.3	1210.9/75.7	2037.9/63.7
(b) MATH-500 Method	GPU/TP	$C=1$	$C=2$	$C=4$	$C=8$	$C=16$	$C=32$
I-DLM-8B	1/1	406.7/406.7	824.4/422.5	1467.7/408.2	2359.2/336.6	3375.9/247.5	4437.6/161.7
BRISK-DLM-8B (Ours)	1/1	516.9/516.9	998.4/481.6	1689.5/437.2	2950.0/399.3	4345.2/280.9	5367.3/182.9
DFlash OOD-thinking	1/1	377.8/377.8	706.3/372.8	1248.8/368.8	2090.4/313.2	3589.1/260.7	4454.7/167.8
DFlash native NT	1/1	808.0/808.0	1563.7/973.0*	2950.6/874.7*	3857.8/685.5*	4992.5/504.8	7995.3/395.7*
D-PACE-DFlash OOD-thinking	1/1	426.7/426.7	677.7/347.9	1303.4/384.9	2256.1/319.0	3550.7/251.4	4742.5/172.8
D-PACE-DFlash native NT	1/1	842.4/842.4*	1507.8/765.8	2814.4/799.7	3300.1/715.5	5487.4/510.4*	7865.5/399.6
Qwen3-8B	1/1	184.3/184.3	348.9/174.5	686.7/171.7	1321.1/165.2	2467.9/154.4	4327.8/135.4
SDAR-8B	1/1	304.7/304.7	308.9/232.9	383.8/137.3	668.7/114.2	1305.3/111.1	1902.3/78.5
LLaDA-2.1-mini (16B)	1/1	501.8/501.8	331.5/332.6	303.5/89.1	324.6/46.1	668.1/46.6	766.1/23.9
SDAR-30B-A3B	1/1	306.6/306.6	236.4/199.9	81.0/21.7	167.3/22.4	410.0/27.7	753.0/25.9
LLaDA-2.0-flash (100B)	4/4	475.1/475.1	272.8/136.4	543.9/154.1	461.3/66.1	484.4/32.7	883.9/28.5
LLaDA-2.1-flash (100B)	4/4	596.0/596.0	749.5/374.8	676.6/189.7	528.3/99.5	467.8/30.6	852.2/27.4
I-DLM-32B	2/2	124.6/124.6	239.6/123.4	438.8/118.5	764.8/101.1	1115.4/74.0	1448.3/48.0
BRISK-DLM-32B (Ours)	2/2	161.6/161.6	340.9/172.9	660.1/167.6	1021.7/140.1	1434.2/100.3	1909.7/71.7
Qwen3-32B	2/2	81.6/81.6	159.7/79.9	319.7/79.9	629.2/78.7	1207.1/75.5	1890.6/59.1
(c) HumanEval Method	GPU/TP	$C=1$	$C=2$	$C=4$	$C=8$	$C=16$	$C=32$
I-DLM-8B	1/1	536.3/536.3	905.0/498.1	1711.9/481.2	2573.4/389.1	3657.6/285.6	4798.3/170.5
BRISK-DLM-8B (Ours)	1/1	686.4/686.4	1114.9/561.1	2033.6/518.8	3125.5/399.4*	4689.6/309.3*	5720.1/181.0
DFlash OOD-thinking	1/1	676.2/676.2	780.0/408.3	1765.9/476.5	2176.0/447.7	3579.7/305.2	4521.6/189.9
DFlash native NT	1/1	215.4/215.4	415.5/819.1	858.1/1263.3	1544.6/1128.2	2573.8/851.5	3967.8/545.4
D-PACE-DFlash OOD-thinking	1/1	516.3/516.3	817.6/468.9	1406.2/498.5	2301.4/418.3	3801.0/318.6	4850.1/198.2
D-PACE-DFlash native NT	1/1	657.6/657.6	1304.4/678.5	1458.2/1362.8	2838.8/1228.8	4373.9/532.0	6468.0/533.0*
Qwen3-8B	1/1	182.7/182.7	344.3/172.2	676.0/169.0	1271.7/159.0	2475.2/154.8	4327.0/135.4
SDAR-8B	1/1	275.8/275.8	280.5/212.3	314.0/109.4	731.8/136.0	1006.8/120.7	1851.7/57.9
LLaDA-2.1-mini (16B)	1/1	403.9/403.9	449.4/363.7	204.0/58.3	377.0/53.6	720.7/50.5	1076.2/37.1
SDAR-30B-A3B	1/1	254.8/254.8	244.2/177.8	89.7/24.6	171.0/23.3	398.0/27.6	761.1/26.6
LLaDA-2.0-flash (100B)	4/4	1581.5/1581.5	1586.9/1165.1	2162.4/741.0	2683.6/402.4	2054.1/139.3	3999.0/135.2
LLaDA-2.1-flash (100B)	4/4	1882.5/1882.5*	1754.6/1346.0*	2380.7/859.4*	3055.6/636.9	2343.0/159.0	4466.8/152.0
I-DLM-32B	2/2	121.2/121.2	228.2/120.4	443.2/118.1	744.5/108.9	1068.0/72.0	1412.7/47.6
BRISK-DLM-32B (Ours)	2/2	163.9/163.9	290.1/157.4	511.5/183.8	872.4/131.8	1273.0/87.3	1660.8/55.8
Qwen3-32B	2/2	81.5/81.5	159.2/79.6	319.9/80.0	627.5/78.4	1204.7/75.3	1644.2/51.4

Rows are grouped with the 8B systems first, followed by larger-scale references.

Table 11: H200 C=1 tokens per forward. A dash denotes an unavailable matched measurement.

Method	MATH-500	GSM8K	HumanEval
LLaDA-2.0-flash (100B)	3.814	5.404	11.636
LLaDA-2.1-flash (100B)	4.697	6.628	13.838
SDAR-8B	1.824	1.770	1.618
SDAR-30B-A3B	1.680	1.817	1.411
DFlash OOD-thinking	3.089	3.141	5.404
DFlash native NT	6.502	7.087	1.724
D-PACE-DFlash OOD-thinking	3.448	3.070	4.096
D-PACE-DFlash native NT	6.759	6.502	9.421
I-DLM-8B	2.794	2.897	3.710
BRISK-DLM-8B (Ours)	**3.497**	**4.264**	**4.534**
Qwen3-8B	1.000	1.000	1.000
I-DLM-32B	2.048	1.967	1.990
BRISK-DLM-32B	2.734	2.805	2.713
Qwen3-32B	1.000	1.000	1.000

Table 12: A100 MATH-500 throughput (aggregate TPS / median request TPS). GPU/TP denotes GPU count and tensor-parallel degree. † denotes TP2 at $C \leq 8$ and TP4 at $C \geq 16$.

Method	GPU/TP	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$
SDAR-8B	1/1	128.94 / 128.94	130.25 / 96.86	160.89 / 57.78	309.67 / 64.86	521.80 / 32.61	950.81 / 39.90
SDAR-30B-A3B	2/2	125.12 / 125.13	128.83 / 98.25	61.66 / 17.57	97.06 / 13.42	194.36 / 13.16	354.39 / 12.10
DFlash OOD-thinking	1/1	160.50 / 160.50	319.18 / 160.18	592.32 / 168.67	976.71 / 142.05	1466.81 / 106.81	1773.39 / 65.67
DFlash native NT	1/1	573.60 / 573.60	1008.56 / 531.07	1142.62 / 385.19	1413.40 / 328.30	2411.27 / 254.63	2937.99 / 146.33
I-DLM-8B	1/1	182.19 / 182.19	333.99 / 169.59	579.19 / 160.42	965.88 / 137.14	1491.94 / 103.58	1723.26 / 59.89
BRISK-DLM-8B (Ours)	1/1	231.56 / 231.56	385.00 / 195.49	666.74 / 184.67	1207.78 / 171.49	1920.29 / 133.32	2084.28 / 72.44
Qwen3-8B	1/1	73.48 / 73.48	143.76 / 71.88	278.86 / 69.72	511.22 / 63.91	914.98 / 57.20	1579.37 / 49.37
I-DLM-32B	2/2; 4/4†	57.19 / 57.19	101.72 / 50.99	181.62 / 47.93	303.05 / 40.61	751.97 / 49.47	956.14 / 31.95
BRISK-DLM-32B (Ours)	2/2; 4/4†	75.61 / 75.61	137.15 / 68.66	256.24 / 67.65	389.63 / 56.99	970.71 / 68.52	1247.33 / 46.08
Qwen3-32B	2/2; 4/4†	34.46 / 34.46	68.23 / 34.12	134.97 / 33.75	264.11 / 33.02	794.01 / 49.63	1454.08 / 45.46

BRISK-DLM-8B exceeds same-scale I-DLM at every listed concurrency, with a 21.9% geometric-mean gain. For 32B, the means are 34.1% in the TP2 regime and 29.8% in the TP4 regime, each relative to I-DLM under the same allocation. These results do not imply superiority to every other method: at $C = 32$/TP4, Qwen3-32B achieves 1454.08 TPS versus 1247.33 for BRISK-DLM-32B. This A100 panel is separate from the 64-prompt head execution study.

D.4 Stride-size sensitivity

Table 13 reports the complete throughput sweep over $N \in \{4, 8, 16\}$ under the same H200 serving setup.

Table 13: **Stride-size sensitivity on MATH-500.** Total serving throughput (model tokens/s) for Original I-DLM and BRISK across proposal widths and concurrency levels. GM gain denotes the geometric mean of the BRISK/I-DLM throughput ratio over all six concurrency levels.

Method	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$	GM gain
I-DLM $N = 4$	427.6	772.8	1484.1	2430.8	4157.8	6239.1	–
BRISK $N = 4$	457.1	904.2	1705.0	2982.9	4736.8	7057.7	+14.7%
I-DLM $N = 8$	406.7	824.4	1467.7	2359.2	3375.9	4437.6	–
BRISK $N = 8$	516.9	998.4	1689.5	2950.0	4345.2	5367.3	**+22.9%**
I-DLM $N = 16$	398.7	737.8	1290.2	1891.4	2247.3	2619.9	–
BRISK $N = 16$	482.9	828.5	1477.6	2306.3	2791.6	3347.7	+20.2%

D.5 Impact of Hardware

We compare MATH-500 aggregate throughput on H200 and A100 (Table 10 and Table 12). With one GPU, BRISK-DLM-8B outperforms I-DLM at all six concurrency levels, with geometric-mean gains of 22.9% on H200 and 21.9% on A100. For 32B, restricting the comparison to the common TP2 regime ($C \in \{1, 2, 4, 8\}$) yields geometric-mean gains of 38.8% on H200 and 34.1% on A100.

E Top-K Sensitivity Analysis

Evaluation setting. We evaluate inference support $K \in \{4, 8, 16, 32\}$ with a frozen 8B BRISK backbone. Fixed-state candidate statistics and natural-EOS generation are evaluated on the full MATH-500 benchmark. Serving is evaluated on an A100 at $C \in \{1, 8, 16\}$ with a fixed-output request bank.

Marginal utility beyond $K = 16$. Expanding the support from $K = 16$ to $K = 32$ changes the selected-token $\log p$ only from -11.671 to -11.669, while ranks 17–32 are selected in 0.071% of first-breaker states. We therefore report this as a direct $K = 16 \to 32$ comparison rather than as a sweep. These results suggest that $K = 16$ already contains nearly all candidates that affect the corrector's decision: the additional ranks increase candidate availability but rarely change the selected token. Consequently, expanding to $K = 32$ offers little room for further gains with the fixed corrector used in this evaluation.

Table 14: **Top-K serving throughput on A100 (TPS).**

K	$C = 1$	$C = 8$	$C = 16$
4	91.70	706.19	1085.16
8	92.90	682.29	1076.39
16	**94.22**	**741.38**	**1100.17**
32	92.47	707.18	1012.90

Serving throughput. Across $C = 1$, $C = 8$, and $C = 16$, $K = 16$ attains the highest throughput, reaching 94.22, 741.38, and 1100.17 TPS, respectively. It is 1.4–5.0% faster than the best smaller-support setting at each concurrency and 1.9–8.6% faster than $K = 32$. Thus, $K = 16$ gives the highest measured throughput across all evaluated loads.

F Inference-Head System Optimization

Setup. We evaluate 64 MATH-500 prompts on one NVIDIA A100-SXM4-40GB (TP=1), with exactly 2,048 client-visible tokens per request, stride $N = 8$, and $C \in \{1, 8, 32\}$. All settings share the trained backbone; both head-enabled settings use the same causal-head checkpoint. We report medians over six counterbalanced fresh-server runs per setting, computing request p95 within each run first. TPF divides client-visible tokens by request-level backbone forward exposures, including prefill and actual active batch sizes in decoding.

From proposal efficiency to throughput. Both head implementations improve accepted-prefix length and TPF (Table 15), but reference execution overhead outweighs this benefit and reduces TPS. Optimized execution increases TPS by 8.2%, 9.5%, and 9.1% over head off at $C = 1, 8, 32$. The optimized head retains similar proposal efficiency while exceeding head-off TPS and lowering request p95 at all three concurrency levels.

Head execution cost. On the same recorded inputs, fused recurrent execution followed by CUDA Graph replay reduces head-invocation latency by 23.5×, 16.0×, and 8.5× at $C = 1, 8, 32$, respectively (Figure 9). These are head-runtime speedups; Table 15 measures end-to-end serving performance.

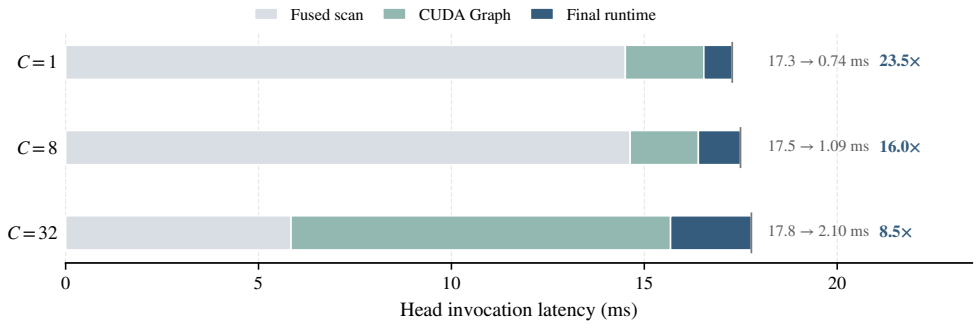

Figure 9: **Cumulative reduction in head execution cost.** Each bar reconstructs reference latency: two differences between successive measured paths (reference to fused scan, then to CUDA Graph replay), followed by the remaining runtime. These segments are not independent kernel timings. Labels show reference-to-optimized latency and speedup.

Table 15: **End-to-end inference-head system ablation.** Absolute medians over six fresh-server runs per setting on 64 MATH-500 prompts; request p95 is computed per run. Both head implementations use the same checkpoint.

C	Setting	Mean prefix ↑	TPF ↑	Total TPS ↑	Request p95 (s) ↓
1	Head Off	4.20	3.37	90.0	30.20
	Causal Head (Reference)	4.58	3.74	60.4	40.35
	Causal Head (Optimized)	4.62	3.75	**97.4**	**25.00**
8	Head Off	4.23	3.39	682.1	24.72
	Causal Head (Reference)	4.58	3.71	436.3	38.18
	Causal Head (Optimized)	4.60	3.73	**747.0**	**21.91**
32	Head Off	4.26	3.43	1237.3	49.67
	Causal Head (Reference)	4.57	3.71	1007.7	64.00
	Causal Head (Optimized)	4.57	3.69	**1350.3**	**47.42**