SPEAR: EXACT SELF-SPECULATIVE SAMPLING FROM THE AUTOREGRESSIVE-ORDER POLICY OF DIFFUSION LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Diffusion language models (dLLMs) can decode several tokens per forward pass and in any order, yet for reasoning the autoregressive (AR) order of the same model reaches a higher Pass@k. Sampling that AR-order policy exactly costs one forward pass per token, and the parallel decoders that avoid this cost sample a different distribution. We study whether exact sampling from the AR-order policy can be made cheaper on ordinary bidirectional dLLMs, and when it is worth its cost. We propose SPEAR, a training-free self-speculative sampler that drafts a block from one forward pass, verifies it against the exact AR-order conditionals in one batched pass, and keeps or resamples each token by the speculative sampling rule. We prove that its output follows the AR-order policy at any temperature. Experiments with three dLLMs on math and code show that the models agree with most of their own drafts, and that exact samples beat lossy parallel decoding at a matched number of forward passes. Matched on GPU time, the advantage disappears on math and survives on code only against a fixed-step decoder that breaks program syntax. Code is available at `https://anonymous.4open.science/r/spear-code-iclr27-5C0B/`.

1 INTRODUCTION

Masked diffusion language models (dLLMs) generate text by filling in the masked positions of a fixed-length canvas. They grew out of discrete diffusion over tokens (Lou et al., 2024; Sahoo et al., 2024; Ou et al., 2025) and now reach billions of parameters in LLaDA (Nie et al., 2025), its preference-optimized successor LLaDA-1.5 (Zhu et al., 2025), and Dream (Ye et al., 2025). Two properties make them attractive: they can commit several tokens per forward pass, and they are not tied to a left-to-right order. For reasoning, the second property has a cost. Ni et al. (2026) show that confidence-ordered decoding skips the high-entropy tokens that decide a reasoning trajectory, the same sparse tokens that drive effective reinforcement learning in autoregressive models (Wang et al., 2025), and that this lowers the solution coverage measured by Pass@k (Yue et al., 2025). Decoding the same model in autoregressive (AR) order, always filling the leftmost mask, recovers it, and plain GRPO (Shao et al., 2024) on this AR-order policy is simpler and stronger than diffusion-specific reinforcement learning (e.g., Zhao et al., 2025).

This finding raises an immediate problem of cost. Pass@k, majority voting and on-policy reinforcement learning all need samples from the AR-order policy at a positive temperature, and drawing one exactly takes one forward pass per token, because each conditional depends on the tokens committed before it. Parallel decoders avoid this cost by committing several tokens from the one-shot marginals of a single pass, choosing how many with a confidence threshold, as Fast-dLLM does (Wu et al., 2026), or with an entropy bound (Ben-Hamu et al., 2025); fewer denoising steps even raise Pass@k at large k through diversity, at a cost in single-shot accuracy (Lamont et al., 2026). But the product of the marginals is not the AR-order distribution, and Feng et al. (2025) prove that keeping the sequence error low needs a number of steps that grows with the sequence length. On LLaDA-8B, committing eight tokens per pass this way accumulates 15.4 nats of divergence per block (Section 4.1). Evaluation must therefore separate two questions. The first asks whether a dLLM can sample its own AR-order policy exactly while committing several tokens per pass. The second asks whether doing so is worth its cost, which depends on whether the budget is counted in forward passes or in GPU

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.

time. We call the agreement between a dLLM's one-shot marginals and its AR-order conditionals *self-consistency*; it decides the answer to the first question.

Recent work makes this problem concrete, but does not yet fully solve it. Speculative sampling drafts from a cheap proposal and accepts each token by a ratio rule with residual resampling, which returns the target distribution exactly (Leviathan et al., 2023; Chen et al., 2023). For dLLMs, the self-speculative decoder of Gao et al. (2025) drafts and verifies with the same model but is exact only at temperature zero, where it reproduces one deterministic trace, and SimSD (Cui et al., 2026) drafts with a separate model and gives no exactness guarantee. Exact stochastic self-speculation exists for any-order and any-subset autoregressive models (Guo & Ermon, 2026), which are trained to score arbitrary revealed subsets, so their verification is cheap, and Agrawal et al. (2026) give a distribution-preserving speculative decoder for dLLMs that also treats exact verification at temperature. The sampling machinery is therefore not ours, and we claim neither to be first to exact self-speculative sampling nor first to consider it at temperature for dLLMs. These works design samplers. We ask a different question. Specifically, on an ordinary bidirectional dLLM, whose architecture gives no cheap prefix-conditional scoring, what does exact sampling of the AR-order policy cost, and what does the exact distribution buy that a lossy one does not?

Motivated by these questions, we propose SPEAR (Self-sPEculative AR-order sampling), a training-free sampler that uses one frozen dLLM as both drafter and verifier (Figure 1). SPEAR drafts a block of tokens from the marginals of one forward pass. It then verifies the block against the exact AR-order conditionals, which it obtains by revealing each prefix of the draft in its own row of one batched forward pass. A left-to-right scan keeps each drafted token or replaces it by the speculative sampling rule of Leviathan et al. (2023). Our key observation is that on a bidirectional dLLM this correction cannot be scored in one causal pass, so it costs an L-row batch per block, and whether that cost pays depends on the hardware as much as on the model (Section 3.2).

We validate SPEAR both theoretically and empirically. We prove that its output follows the AR-order policy exactly at any temperature, with greedy self-speculation as its temperature-zero case, and analyse a variant that accepts a whole block through a Bernoulli factory (Keane & O'Brien, 1994; Nacu & Peres, 2005), a discrete analogue of first-order rejection sampling (Chen et al., 2026), whose worst-case cost at a fixed clip level does not grow with the block length. Experiments with LLaDA-8B, LLaDA-1.5 and Dream-7B on GSM8K, MATH-500, HumanEval and MBPP confirm exactness on the real models and show that exact samples beat confidence-ordered decoding at a matched number of forward passes. Matched on GPU time, the advantage disappears on math and survives on code only against a fixed-step decoder that breaks program syntax.

Key contributions.

- We formulate exact sampling from a dLLM's AR-order policy as a cost question with two budgets, forward passes and GPU time, and identify self-consistency as the quantity that governs the first (Section 2).
- We propose SPEAR and prove that it samples the AR-order policy exactly at any temperature, and we show that a Bernoulli-factory variant with block-length-independent worst-case cost is impractical on real dLLMs because their log-ratios are heavy-tailed (Section 3).
- We show that three dLLMs accept 82 to 91% of their own drafted tokens, with rejections at the high-entropy tokens that steer a trajectory, and that exact samples beat a matched-budget confidence decoder in Pass@1 and in majority voting at every vote size (Sections 4.1 and 4.2).
- We show that at equal GPU time the advantage disappears on math, because the speedup rests on a batching discount that vanishes as the forward pass becomes compute-bound, and survives on code against a fixed-step decoder whose programs often fail to parse (Section 4.3).

The code, every logged sample and the scripts that recompute every number in the paper are released at `https://anonymous.4open.science/r/spear-code-iclr27-5C0B/`.

2 PROBLEM FORMULATION

We first define the target of interest. A masked diffusion model replaces each token of a clean sequence with a mask token independently with probability $t \in [0, 1]$ and is trained to predict the clean tokens at the masked positions (Ou et al., 2025; Sahoo et al., 2024). Generation starts from a canvas of N masks after the prompt c and unmasks positions over a schedule. Let $f_{\theta,k}(x)$ be the logits of the model at position k of an input x, and $\tau > 0$ a temperature. Always unmasking the leftmost mask defines the AR-order policy

$$\pi_{\text{AR}}(o_k \mid o_{<k}, c) = \text{softmax}\big(f_{\theta,k}([c, o_{<k}, [\texttt{MASK}], \ldots, [\texttt{MASK}]])/\tau\big), \tag{1}$$

and the chain rule $\prod_k \pi_{\text{AR}}(o_k \mid o_{<k}, c)$ is the target distribution. At $\tau = 0$ it is the point mass that greedy decoders reproduce. Drawing from Eq. 1 directly, which we call STEPWISE decoding, takes N forward passes, one per token.

A parallel decoder instead reads the one-shot marginals $q_k(\cdot) = \text{softmax}(f_{\theta,k}/\tau)$ of a single pass over $[c, o_{<p}, [\texttt{MASK}], \ldots]$ and commits several positions from their product. The first of these positions is exact, since $q_p = \pi_{\text{AR}}(\cdot \mid o_{<p}, c)$, but later ones are not, because q_{p+j} ignores the tokens drafted at $p, \ldots, p + j - 1$. We measure how far apart the two are by the acceptance probability of speculative sampling,

$$\alpha_{p+j} = \textstyle\sum_o \min\big(q_{p+j}(o),\, \pi_{\text{AR}}(o \mid o_{<p}, \tilde{o}_{p:p+j}, c)\big) = 1 - \text{TV}\big(q_{p+j}, \pi_{\text{AR}}\big), \tag{2}$$

and call a model self-consistent when α is close to one on most positions.

Two metrics read the output distribution. Pass@k is the probability that at least one of k independent samples is correct, which we estimate without bias (Chen et al., 2021). On a problem where one sample is correct with probability p it equals $1 - (1 - p)^k$, so it depends on p alone. Majority voting (Wang et al., 2023) returns the most frequent answer, so it depends on whether p exceeds the largest share w held by any single wrong answer, and two decoders with equal p can rank differently under voting. We count cost in two budgets: forward calls per sample, the hardware-independent measure of the acceleration literature (Gao et al., 2025; Ben-Hamu et al., 2025), and measured GPU time per problem. Our goal is to draw exact samples from π_{AR} with fewer forward calls than STEPWISE, and to measure whether the exact distribution beats lossy parallel decoding under each budget. Section 3 defines the sampler.

3 SPEAR

SPEAR, short for Self-sPEculative AR-order sampling, draws from the target distribution of Eq. 1 with one frozen dLLM acting as both drafter and verifier. It decodes the canvas in blocks. For each block it drafts L tokens from one forward pass, verifies them against the exact AR-order conditionals, and commits the longest prefix that the speculative sampling rule accepts, followed by one exact token. It needs no training, no auxiliary model and no change to the network. Algorithm 2 in Appendix B.3 states the procedure in full.

3.1 DRAFT, VERIFY, ACCEPT

Let $o_{<p}$ be the committed prefix and L the block length. In the draft step, one forward pass over $[c, o_{<p}, [\texttt{MASK}], \ldots]$ gives the marginals $q_p, \ldots, q_{p+L-1}$, and we draw $\tilde{o}_k \sim q_k$ independently, so the proposal is the product $\prod_k q_k(\tilde{o}_k)$. In the verify step we build L inputs, the j-th revealing $\tilde{o}_{p:p+j}$ and masking the rest, and run them as one batched forward pass. On a batch-invariant backbone, row j returns the exact conditional $\pi_{\text{AR}}(\cdot \mid o_{<p}, \tilde{o}_{p:p+j})$ at position $p + j$; where batching changes the logits, the rows must be run separately (Proposition 1, Appendix G.4). Position p needs no row, since $q_p = \pi_{\text{AR}}$ there.

In the accept step we scan left to right. The drafted token $\tilde{o}_{p+j}$ is accepted with probability $\min\big(1, \pi_{\text{AR}}(\tilde{o}_{p+j} \mid \cdot)/q_{p+j}(\tilde{o}_{p+j})\big)$. On the first rejection, that position is resampled from the normalized residual $(\pi_{\text{AR}} - q_{p+j})_+$ and the block ends. If all L tokens are accepted, one further token is drawn from the conditional given the full block, so a block commits between one and $L + 1$ tokens. Because rejections fall on high-entropy positions (Section 4.1), the entropy of q_k, which the draft pass gives for free, can also truncate a draft before its first high-entropy position. Truncation changes only where blocks end, not the output distribution.

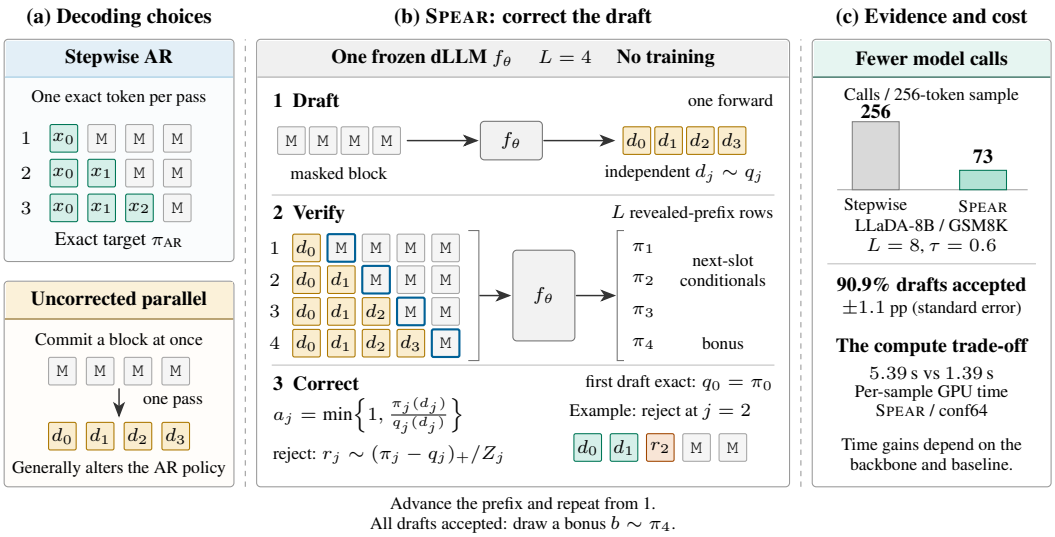


Figure 1: Overview of SPEAR. (a) stepwise and uncorrected parallel sampling; (b) one block at $L = 4$: draft, revealed-prefix verification rows and left-to-right acceptance (d_j drafted token; green accepted, orange resampled); (c) forward calls and GPU time per sample.

3.2 COST OF VERIFICATION

In an AR model one causal pass scores every prefix of a draft at once. A bidirectional dLLM does not allow this: revealing a different prefix changes the representation at every position, so each of the L revealed-prefix inputs is a distinct tensor and a block costs an L-row batch (Figure 4, Appendix B.4). Batching helps only because the rows share one pass over the weights, so while the forward pass is memory-bound the batch costs less than L single passes. Write c for the cost of one pass, $\bar{L}$ for the mean number of verified rows and t for the tokens a block commits. SPEAR then costs $(1+\alpha\bar{L})\,c$ per block against $t\,c$ for STEPWISE, where $\alpha \leq 1$ is the batching efficiency, so its speedup is $t/(1+\alpha\bar{L})$ and never exceeds $(\bar{L}+1)/(1+\alpha\bar{L})$. The speedup therefore tends to one as the pass becomes compute-bound, whatever the acceptance rate. Verification also needs logits only at drafted positions, so we apply the 126,464-way output head at those positions alone, which is mathematically identical and saves about 10% of wall-clock.

3.3 THEORETICAL GUARANTEES

We now justify the exactness of SPEAR and state what a variant that removes the L-row batch would cost. The formal statements and proofs are in Appendices B.1 and B.2.

Theorem 1 (Exactness). *For any temperature $\tau > 0$ and block length L, if every verification row returns the conditional it would return alone, the tokens committed by SPEAR are distributed exactly as the AR-order policy π_{AR} of Eq. 1.*

The proof is an induction over positions. Position p is drawn from $q_p = \pi_{AR}(\cdot \mid o_{<p})$. Conditioned on the prefix committed through $p+j-1$, the draft $\tilde{o}_{p+j}$ is accepted or replaced by the residual rule against $\pi_{AR}(\cdot \mid o_{<p}, \tilde{o}_{p:p+j})$, which returns exactly that conditional (Leviathan et al., 2023; Chen et al., 2023). On the accept path the committed tokens equal the drafted ones, so the row that reveals $\tilde{o}_{p:p+j}$ supplies the right conditional, and the committed prefix follows the chain rule. Theorem 1 makes the output independent of L, of the truncation rule and of the acceptance rate, which change only the cost. Its one condition is batch invariance, which holds exactly when the rows are run separately and which we test on the real models in Section 4.1.

The L-row batch is the cost that Section 3.2 identifies, so a sampler whose verification cost does not grow with L is the natural target. A variant accepts a whole block from its log-ratio, estimated at a random subset of positions, through a Bernoulli factory (Keane & O'Brien, 1994; Nacu & Peres, 2005) with clip level B. Theorem 2 shows that it is exact when the clipped mass vanishes, has a χ^2 error bound otherwise, and has a worst-case cost at fixed B that does not depend on L. The payoff

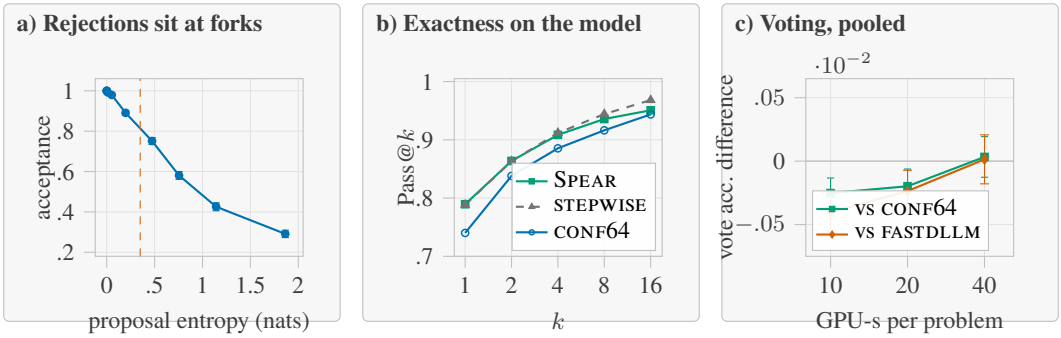


Figure 2: What verification reveals. (a) acceptance against proposal entropy on GSM8K with LLaDA-8B; (b) Pass@k of SPEAR and STEPWISE on shared problems; (c) majority-vote difference from SPEAR at matched GPU time.

holds only on paper. On LLaDA-8B the per-position log-ratio is heavy-tailed, with median 0.001 nats and maximum 36.3, so removing the clip bias at $L = 8$ needs $B \approx 254$ and about 10^{110} evaluations per block. We therefore report the variant as a construction validated on a toy oracle, not as a usable sampler.

4 EXPERIMENTS

Lossy parallel decoding gives up the target distribution for speed, and exact AR-order sampling gives up speed for the target. Our experiments test whether SPEAR removes the first trade on real dLLMs and whether the exact distribution pays under each budget of Section 2. We organize them around three research questions (RQs):

(RQ1) Is SPEAR exact on real dLLMs, and how self-consistent are they?
(RQ2) At a matched number of forward passes, does the exact distribution beat lossy parallel decoding?
(RQ3) Does that advantage survive when decoders are matched on GPU time?

We evaluate on GSM8K (Cobbe et al., 2021) and all of MATH-500 (Hendrycks et al., 2021; Lightman et al., 2024) with LLaDA-8B-Instruct (Nie et al., 2025), LLaDA-1.5 (Zhu et al., 2025) and the Qwen-based Dream-7B (Ye et al., 2025), and on all of HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) with LLaDA-8B. LLaDA-1.5 is LLaDA-8B after preference optimization, so the two share weights; Dream-7B supplies the second architecture. Unless otherwise stated, every decoder samples at $\tau = 0.6$ with generation length 256, SPEAR uses $L = 8$, math answers are graded by numeric or symbolic equivalence and programs by their unit tests, and intervals are paired bootstrap 95% intervals over problems. Sample counts, cohorts and grading are in Appendix G.1. We compare SPEAR with four decoders:

- **STEPWISE** fills the leftmost mask at one forward pass per token. It samples the same distribution as SPEAR and is the exact reference.
- **CONF**S is confidence-ordered decoding with S steps (Nie et al., 2025). CONF64 commits four tokens per pass and uses about as many forward passes as SPEAR.
- **FASTDLLM** commits every position whose confidence exceeds 0.9 (Wu et al., 2026). Our reimplementation is faster and more accurate at Pass@1 than the authors' code, with or without its key-value cache (Appendix H.6).
- **Uncorrected SPEAR** commits all L drafted tokens without verification, which isolates what the correction contributes.

Greedy self-speculation (Gao et al., 2025), the $\tau = 0$ case of SPEAR, is reported in Appendix D.1, and a toy oracle whose AR-order policy is a known joint distribution validates exactness in Appendix B.2. All runs use one NVIDIA H100 GPU.

Table 1: Exactness, self-consistency and Pass@1 across models and datasets.

				Pass@1 at matched budget		
model	dataset	accept	Δ_{STEPWISE}	SPEAR	CONF64	FASTDLLM
LLaDA-8B	GSM8K	0.909 ± 0.011	$-0.001\ [-.015, +.013]$ ✓	0.792	0.731	**0.800**
LLaDA-1.5	GSM8K	0.914 ± 0.005	$+0.009\ [-.009, +.028]$ ✓	**0.745**	0.683	0.719
Dream-7B	GSM8K	0.896 ± 0.008	$-0.018\ [-.047, +.008]$ ✓	0.766	0.520	**0.792**
LLaDA-8B	MATH-500	0.879 ± 0.010	$+0.004\ [-.031, +.043]$ ✓	0.333	0.286	**0.352**
LLaDA-8B	HumanEval	0.875 ± 0.004	$+0.004\ [-.060, +.074]$ ✓	0.345	0.157	**0.393**
LLaDA-8B	MBPP	0.823 ± 0.004	$+0.013\ [-.038, +.062]$ ✓	0.459	0.164	**0.482**

Table 2: Pass@k and forward calls per sample on GSM8K with LLaDA-8B.

method	n	fwd/sample	Pass@1	Pass@8	paired Δ vs STEPWISE
SPEAR (exact AR)	96	73	**0.792**	0.941	$-0.001\ [-0.015, +0.013]$
STEPWISE (exact AR)	32	256	0.788	0.944	n/a
FASTDLLM	96	71	0.800	0.908	not exact
CONF256	64	256	0.787	0.907	not exact
CONF64	96	64	0.731	0.942	not exact

4.1 RQ1: EXACTNESS AND SELF-CONSISTENCY ON REAL MODELS

We first answer RQ1 by comparing SPEAR with STEPWISE on the problems both were run on, and by measuring how often each model accepts its own drafts. Table 1 reports both. The column "accept" is the mean per-token acceptance probability of Eq. 2, with one standard error over prompts. The column Δ_{STEPWISE} is the paired Pass@1 difference between SPEAR and STEPWISE with its 95% interval; STEPWISE costs one pass per token, so it was run on a subset of each cell, and a ✓ marks an interval that contains zero, as exactness requires.

Every interval contains zero, in all six cells, and the Pass@k curves of the two samplers track each other on the 32 GSM8K problems both were run on (Figure 2b). A paired permutation test on the full distribution over extracted answers finds no excess total variation between SPEAR and STEPWISE beyond what two samples of one law give (-0.002, $p = 0.65$), while it separates SPEAR from CONF64 ($+0.054$, $p < 0.001$), so the test has power (Appendix D.3). The condition of Theorem 1 holds on LLaDA: each revealed-prefix row recomputed alone matches its value inside the batch to 0.000000 at every depth. It fails on Dream-7B, by 0.54 to 1.34 nats, so Dream verifies one row per pass (Appendix G.4).

The models are self-consistent. They accept 82 to 91% of their own drafted tokens, most on GSM8K and least on MATH-500 and code. Rejections are concentrated rather than uniform: in decile bins of proposal entropy, acceptance is near one at low entropy and falls to 0.29 in the top decile (Figure 2a). Split at 0.35 nats, it is 0.981 below and 0.656 above, with Spearman correlation -0.391 on LLaDA-8B and -0.487 on Dream-7B (Table 7). The rejected tokens are thus the high-entropy tokens that Ni et al. (2026) identify as steering a trajectory. The same measurement prices what a lossy decoder loses there. Committing the token at block depth j from the uncorrected marginal incurs $\mathrm{KL}(q_{p+j} \,\|\, \pi_{\text{AR}})$, which summed over a block grows from 0 at one token per pass to 15.4 nats at eight on GSM8K and 17.6 on MATH-500 (Table 9). These results answer RQ1.

Takeaway 1

SPEAR is exact on real models wherever the backbone is batch-invariant. A dLLM agrees with most of its own drafts and disagrees at the high-entropy tokens that steer a trajectory, which is exactly where lossy parallel decoding drifts.

4.2 RQ2: QUALITY AT A MATCHED NUMBER OF FORWARD PASSES

We next answer RQ2 by comparing decoders at about the same number of forward calls per sample. Table 2 gives the main GSM8K comparison; its rows cover different problem sets, given by n, so every claim below is paired on shared problems. The lossy decoders trade single-shot qual-

Table 3: Correct-answer share p and largest wrong-answer share w per decoder.

	means over 660 combinations				paired vs SPEAR		
decoder	p	w	$p-w$	correct leads	Δp	Δw	$\Delta(p-w)$
SPEAR (exact)	0.441	0.211	**0.230**	**53.5**%	n/a	n/a	n/a
CONF64	0.381	0.190	0.191	49.2%	+0.059	+0.021	+0.039
FASTDLLM	**0.456**	0.266	0.190	50.6%	−0.015	−0.055	+0.040

ity against coverage: CONF64 has the weakest Pass@1 and the strongest Pass@8, while FASTDLLM, which commits only where the model is confident, inverts this. SPEAR does not show the trade-off. Its Pass@1 is $+0.060$ above CONF64 (CI $[0.035, 0.086]$), its Pass@8 is $+0.033$ above FASTDLLM (CI $[0.003, 0.066]$), and it matches the Pass@1 of the full-budget CONF256 with 73 calls against 256. The other two pairings show no detected difference, and at $k = 64$ the most parallel decoders lead through diversity (Appendix H.9).

The comparison carries over to the other cells of Table 1. SPEAR leads CONF64 at Pass@1 by $+0.062$ on LLaDA-1.5, by $+0.047$ on MATH-500, at every difficulty level (Appendix H.8), by $+0.246$ on Dream-7B, and by $+0.187$ and $+0.295$ on HumanEval and MBPP. FASTDLLM has the better single sample in five of the six cells, while SPEAR has the higher Pass@8 in all six (Appendices D.1 and I). The comparison above also varies sampling order and step count together. Running STEPWISE against CONF256 at an identical 256 passes on 64 shared problems isolates the order. Order does not improve single-shot accuracy there ($+0.017$, CI $[-0.025, +0.061]$) but improves coverage, by $+0.053$ to $+0.056$ at $k = 4$ to 16, all excluding zero (Appendix H.2). The Pass@1 gain over CONF64 is therefore a step-count effect: SPEAR keeps a full-budget decoder's single sample at a quarter of the calls and adds the order advantage at larger k.

The correction is what makes this work. With verification removed, the same proposal at the same temperature, order and block length collapses: Pass@1 falls from 0.792 to 0.067 on GSM8K, from 0.745 to 0.148 on LLaDA-1.5, from 0.766 to 0.006 on Dream-7B and from 0.377 to 0.059 on the first 96 MATH-500 problems, although the uncorrected sampler is 8 to 10 times cheaper per sample.

Takeaway 2

At a matched number of forward passes, exact sampling keeps the single-sample accuracy of a full-budget decoder at a fraction of its calls. The accept-and-reject correction, not the draft, is what makes block drafting work.

Majority voting separates the decoders where Pass@k cannot. Pooled over the four math cells, which supply 660 model-item combinations from 596 distinct items and are bootstrapped by item, SPEAR votes better than CONF64 at every vote size, from $+0.060$ at $k = 1$ to $+0.032$ at $k = 16$, every interval excluding zero (Appendix H.4). Against FASTDLLM the sign changes with k: SPEAR is behind at $k = 1$ (-0.015, $[-0.027, -0.003]$) and ahead at $k = 16$ ($+0.025$, $[+0.005, +0.046]$). Table 3 shows how a decoder can lose the single sample and win the vote. Averaged over the combinations all three decoders cover, with paired differences in the right block, the exact sampler has a slightly lower correct-answer share p than FASTDLLM (-0.015), but its most popular wrong answer holds less of the mass (w of 0.211 against 0.266), which widens the margin $p - w$ that a large vote reads. The table is descriptive, since finite-k voting also depends on the rest of the distribution and on ties. These results answer RQ2.

Takeaway 3

Pass@k and voting read different parts of the output distribution. The threshold decoder often wins the single sample, but exact samples put less mass than it does on any one wrong answer, so they win the vote as it grows.

Table 4: SPEAR minus baseline at matched GPU time on math and code.

	math (majority vote)		HumanEval (Pass@k)		MBPP (Pass@k)	
budget	vs CONF64	vs FASTDLLM	vs CONF64	vs FASTDLLM	vs CONF64	vs FASTDLLM
10 s	-0.025 ✗	-0.040 ✗	$+0.023$	-0.178 ✗	$+0.037$ ✓	-0.109 ✗
20 s	-0.020 ✗	-0.024 ✗	$+0.102$ ✓	-0.074 ✗	$+0.067$ ✓	-0.026
40 s	$+0.003$	$+0.001$	$+0.142$ ✓	-0.010	$+0.077$ ✓	$+0.018$

4.3 RQ3: QUALITY AT MATCHED GPU TIME

To answer RQ3, we match decoders on measured GPU time instead of forward calls. A SPEAR sample issues 73 calls against 64 for CONF64, but each verification call is an L-row batch, so it does about 330 row-forwards against 64 and costs 5.39 s against 1.39 s on GSM8K (Appendix H.3). The ratio differs by cell, so we fix budgets in GPU-seconds per problem and convert each budget into a sample count at each cell's own measured cost. Every cell reaches every budget, so no comparison is censored. Table 4 summarizes the result as paired differences from SPEAR: majority vote on the four math cells and Pass@k on code. A ✓ marks an interval above zero, a ✗ an interval below zero, and a blank an interval that spans zero.

On math, the advantage does not survive. On one common cohort of 596 items, SPEAR votes worse than both baselines at 10 GPU-seconds (-0.025, $[-0.037, -0.014]$, against CONF64; -0.040, $[-0.055, -0.026]$, against FASTDLLM) and at 20 (-0.020; -0.024), and at 40 both intervals span zero (Figure 2c, 95% cluster-bootstrap intervals over items). We read the last budget as inconclusive rather than as parity, since no equivalence margin was fixed in advance. The reason is the cost structure of Section 3.2. Against STEPWISE, the exact reference, the speedup of SPEAR with entropy-adaptive truncation falls from 1.18× at 256 tokens to 1.07× at 512 and 1.00× at 1024, as the batching efficiency α rises from 0.76 to 0.97. It is also gone once four samples of a problem share each pass, and a measured cost model reproduces these speedups on LLaDA-8B and on Dream-7B (Appendix H.7). Dream-7B, which must verify one row per pass, needs 353 calls per sample against 64 and 5.3 times the wall-clock of CONF64 (Appendix G.4).

On code, the advantage survives against one baseline. CONF64 commits four tokens per pass, and only 47.5% and 42.1% of its HumanEval and MBPP programs parse, against 95.2% and 98.9% for SPEAR; the failures are duplicated tokens such as `inside = = False`. Its extra samples are then largely wasted, and at 20 and 40 GPU-seconds SPEAR leads it by $+0.102$ and $+0.142$ Pass@k on HumanEval and by $+0.067$ and $+0.077$ on MBPP, every interval excluding zero. FASTDLLM keeps 98.1% and 99.3% of its programs parseable and is ahead or level at every budget (Appendix I). These results answer RQ3.

Takeaway 4

Exactness is paid for in GPU time, not in forward passes. At equal compute it does not pay on math, and it pays on code only against a fixed-step decoder whose parallel commits break the program.

5 CONCLUSION

The AR-order policy of a diffusion language model reaches higher Pass@k than confidence-ordered decoding, but sampling it exactly costs one forward pass per token. SPEAR samples it exactly while committing several tokens per pass, by using the model as its own drafter and verifier. A dLLM agrees with most of its own drafts, and the exact samples it keeps beat lossy parallel decoding per forward pass and per vote. On a bidirectional backbone, however, verification is an L-row batch, so the saving is in calls rather than in compute. At equal GPU time the exact distribution pays only where lossy decoding breaks the output, as a fixed-step decoder does on code. Verification whose cost does not grow with the number of verified positions on a bidirectional backbone remains future work.

Reproducibility Statement

The exactness claim is proved in full in Appendix B.1, and the sublinear-verification claim in Appendix B.2, with all assumptions stated there. Appendix G.1 lists the models, prompts, decoding hyperparameters, and the answer extraction rule, and Table 20 gives the full configuration of every run. Appendix G.3 describes the Pass@k estimator, the bootstrap procedure, and the paired tests. The sampler, the confidence and greedy baselines, the GRPO code, the job script of every run, the analysis scripts and the R figure scripts are released at `https://anonymous.4open.science/r/spear-code-iclr27-5C0B/`, together with every logged sample. From those samples, one script recomputes every number in the paper on a CPU in about two minutes, and a toy-oracle unit test checks exactness on CPU. All experiments run on a single H100 GPU.

Ethics Statement

This work studies a sampling algorithm for existing open-weight language models on public math and code benchmarks. It involves no human subjects, no new data collection, and no personal data. SPEAR does not change what a model can generate; it changes how the existing AR-order distribution is sampled, so it does not add capabilities beyond those of the base model.

Use of Large Language Models

During the preparation of this paper, we used a large language model to draft and revise text and to improve its grammar, word choice, and conciseness. We also used it to help identify relevant literature. The authors independently read all suggested papers and verified each citation against its original source before inclusion.

We used Claude Code to assist with designing and running the experiments and analyzing the results. The authors reviewed the code and verified all reported numerical results against the saved outputs. All claims are the authors' own, and we take full responsibility for the content of this paper.

References

Sudhanshu Agrawal, Risheek Garrepalli, Raghavv Goel, Christopher Lott, Fatih Porikli, and Mingu Lee. Structuring the future: Diffusion LLM speculative decoding via calibrated draft graphs. In *ICML Workshop on Structured Probabilistic Inference and Generative Modeling*, 2026.

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

Heli Ben-Hamu, Itai Gat, Daniel Severo, Niklas Nolte, and Brian Karrer. Accelerated sampling from masked diffusion models via entropy bounded unmasking. In *Advances in Neural Information Processing Systems*, 2025.

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.

Fan Chen, Sinho Chewi, Constantinos Daskalakis, and Alexander Rakhlin. High-accuracy sampling for diffusion models and log-concave distributions. In *International Conference on Machine Learning*, 2026.

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.

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

Junxia Cui, Haotian Ye, Runchu Tian, Hongcan Guo, Jinya Jiang, Haoru Li, Chaojie Ren, Yiming Huang, Kaijie Zhu, Zhongkai Yu, Kun Zhou, and Jingbo Shang. SimSD: Simple speculative decoding in diffusion language models. *arXiv preprint arXiv:2606.02544*, 2026.

Guhao Feng, Yihan Geng, Jian Guan, Wei Wu, Liwei Wang, and Di He. Theoretical benefit and limitation of diffusion language model. In *Advances in Neural Information Processing Systems*, 2025.

Yifeng Gao, Ziang Ji, Yuxuan Wang, Biqing Qi, Hanlin Xu, and Linfeng Zhang. Self speculative decoding for diffusion large language models. *arXiv preprint arXiv:2510.04147*, 2025.

Gabe Guo and Stefano Ermon. Self-speculative decoding accelerates lossless inference in any-order and any-subset autoregressive models. In *International Conference on Learning Representations*, 2026.

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. In *Advances in Neural Information Processing Systems Datasets and Benchmarks Track*, 2021.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

M. S. Keane and George L. O'Brien. A Bernoulli factory. *ACM Transactions on Modeling and Computer Simulation*, 4(2):213–219, 1994. doi: 10.1145/175007.175019.

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

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, 2023.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *International Conference on Learning Representations*, 2024.

Aaron Lou, Chenlin Meng, and Stefano Ermon. Discrete diffusion modeling by estimating the ratios of the data distribution. In *International Conference on Machine Learning*, 2024.

Şerban Nacu and Yuval Peres. Fast simulation of new coins from old. *The Annals of Applied Probability*, 15(1A):93–115, 2005. doi: 10.1214/105051604000000549.

Zanlin Ni, Shenzhi Wang, Yang Yue, Tianyu Yu, Weilin Zhao, Yeguo Hua, Tianyi Chen, Jun Song, Cheng Yu, Bo Zheng, and Gao Huang. The flexibility trap: Rethinking the value of arbitrary order in diffusion language models. In *International Conference on Machine Learning*, 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. In *Advances in Neural Information Processing Systems*, 2025.

Jingyang Ou, Shen Nie, Kaiwen Xue, Fengqi Zhu, Jiacheng Sun, Zhenguo Li, and Chongxuan Li. Your absorbing discrete diffusion secretly models the conditional distributions of clean data. In *International Conference on Learning Representations*, 2025.

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

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Shenzhi Wang, Le Yu, Chang Gao, Chujie Zheng, Shixuan Liu, Rui Lu, Kai Dang, Xionghui Chen, Jianxin Yang, Zhenru Zhang, et al. Beyond the 80/20 rule: High-entropy minority tokens drive effective reinforcement learning for LLM reasoning. In *Advances in Neural Information Processing Systems*, 2025.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations*, 2023.

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. In *International Conference on Learning Representations*, 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.

Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. Does reinforcement learning really incentivize reasoning capacity in LLMs beyond the base model? In *Advances in Neural Information Processing Systems*, 2025.

Siyan Zhao, Devaansh Gupta, Qinqing Zheng, and Aditya Grover. d1: Scaling reasoning in diffusion large language models via reinforcement learning. In *Advances in Neural Information Processing Systems*, 2025.

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

Appendix

This appendix collects proofs and supplementary results; related work is discussed in Section 1. Appendices A and B give notation, background and proofs, including the sublinear variant and the algorithms. Appendices C, D and E report self-consistency, Pass@k and timing in full, and Appendix F reports the reinforcement-learning study. Appendix G gives the implementation and evaluation protocol, Appendix H the matched-budget, matched-time, voting and scaling analyses, and Appendix I code generation. Appendix J compares SPEAR with other decoders, and Appendix K gives prompts, negative results and limitations. File paths such as `scripts/vote_analysis.py` refer to the code release at `https://anonymous.4open.science/r/spear-code-iclr27-5C0B/`.

A Preliminaries and Theory

A.1 Notation

$\pi_{\text{AR}}(\cdot \mid o_{<k}, c)$ is the autoregressive-order policy of Eq. 1: the softmax at temperature τ of the model logit at position k given the prompt c, the committed prefix $o_{<k}$, and masks at all later positions. q_k is the one-shot proposal marginal at position k, that is, the softmax of the logit at position k read from a single forward pass on $[c, o_{<p}, [\texttt{MASK}], \ldots]$ where p is the block start. Table 5 lists the symbols used throughout.

The per-position log-ratio $w_k = \log \pi_{\text{AR}}(\tilde{o}_k \mid o_{<k}) - \log q_k(\tilde{o}_k)$ measures how far the one-shot marginal is from the exact conditional at a drafted token, and $W = \sum_k w_k$ is the block log-ratio that the sublinear variant estimates.

A.2 Background and Derivations

Masked diffusion models. A masked diffusion model defines a forward process that, at time $t \in [0, 1]$, replaces each token of a clean sequence x_0 by the mask symbol independently with probability t:

$$q(x_{t,k} \mid x_{0,k}) = t\, \delta_{[\texttt{MASK}]} + (1 - t)\, \delta_{x_{0,k}}. \tag{3}$$

Table 5: Symbols.

symbol	meaning
$\pi_{\text{AR}}(\cdot \mid o_{<k})$	autoregressive-order conditional at position k (Eq. 1)
q_k	one-shot proposal marginal at position k (from the draft pass)
$\tilde{o}_k$	drafted token at position k; o_k the committed token
L	block length
τ	sampling temperature
p	block start index; j the within-block offset
w_k	per-position log-ratio $\log \pi_{\text{AR}}(\tilde{o}_k \mid o_{<k}) - \log q_k(\tilde{o}_k)$
W	block log-ratio $\sum_k w_k$
B	clip level of the sublinear variant
J	Poisson count in the Bernoulli factory
NFE	number of forward passes (function evaluations)
$\text{KL}(q \,\|\, \pi_{\text{AR}})$	per-token divergence of the proposal from the conditional

A network $p_\theta(x_0 \mid x_t)$ predicts the clean tokens at masked positions. Training minimizes the negative evidence lower bound, which for masked diffusion reduces to a time-weighted cross entropy over masked positions,

$$\mathcal{L}(\theta) = -\mathbb{E}_{t\sim U[0,1],\, x_t\sim q(\cdot|x_0)}\left[\tfrac{1}{t}\sum_k \mathbb{1}[x_{t,k} = [\texttt{MASK}]] \log p_\theta(x_{0,k} \mid x_t)\right]. \tag{4}$$

At inference the model starts from a fully masked canvas and unmasks positions on a schedule. The schedule is a free choice: unmasking the leftmost masked position at each step gives autoregressive order and defines the policy π_{AR} of Eq. 1. Unmasking the most confident positions gives confidence, or arbitrary, order. Because the model reads the whole canvas at each step, the conditional at a position depends on which other positions have been revealed. This is why order matters, and why π_{AR} differs from the arbitrary-order marginals.

The AR-order conditional is a single read. For the AR-order policy the conditional at position k given a committed prefix $o_{<k}$ is read from one forward pass on the input $[c, o_{<k}, [\texttt{MASK}], \ldots]$: the logit at position k, softmaxed at temperature τ. No marginalization over unmasking trajectories is needed, because the schedule is fixed. This is what makes π_{AR} tractable and what Ni et al. (2026) exploit for reinforcement learning. The one-shot proposal marginal q_k used by SPEAR is read from the same forward pass, but at a position whose left context within the current block is still masked. The two agree exactly at the first block position. They differ, in general, at later positions, and that difference is the source of the accept and reject.

Speculative sampling, derived. We restate the correction of Leviathan et al. (2023) and derive it for completeness. Given a target p and any proposal q with $\text{supp}(p) \subseteq \text{supp}(q)$, draw $x \sim q$ and accept with probability $a(x) = \min(1, p(x)/q(x))$. The probability that the procedure outputs y by acceptance is $q(y)\, a(y) = \min(q(y), p(y))$. The total acceptance probability is $\alpha = \sum_y \min(q(y), p(y))$, so a residual draw is taken with probability $1 - \alpha = \sum_y (p(y) - q(y))_+$. If the residual is defined as $r(y) = (p(y) - q(y))_+/(1-\alpha)$, then the residual path outputs y with probability $(1-\alpha)\, r(y) = (p(y) - q(y))_+$. Adding the two paths,

$$\min(q(y), p(y)) + (p(y) - q(y))_+ = \min(q(y), p(y)) + \max(0, p(y) - q(y)) = p(y), \tag{5}$$

so the output is exactly p. The derivation makes no assumption on q beyond support, which is why any proposal, including the one-shot marginal, yields exact samples.

The Pass@k estimator. For a problem with n samples of which c are correct, the unbiased estimator of the probability that at least one of k draws (without replacement) is correct is

$$\widehat{\text{Pass@}k} = 1 - \binom{n-c}{k} \Big/ \binom{n}{k}, \tag{6}$$

which we average over problems (Chen et al., 2021). We report bootstrap confidence intervals by resampling problems with replacement, and paired differences by bootstrapping the per-problem difference on the set of shared problems, which controls for problem difficulty.

B Exactness and Sublinear Verification

B.1 Proof of the Exactness Theorem

We first restate the speculative sampling correction as a lemma, then prove Theorem 1 by induction over committed positions.

Lemma 1 (Speculative sampling correction, Leviathan et al., 2023; Chen et al., 2023)**.** *Let p and q be distributions on a finite set. Draw $x \sim q$. With probability $\min(1, p(x)/q(x))$ output x, otherwise output a fresh draw from the residual $r(\cdot) = (p(\cdot) - q(\cdot))_+ / \sum_y (p(y) - q(y))_+$. Then the output is distributed exactly as p, for any proposal q with $\mathrm{supp}(p) \subseteq \mathrm{supp}(q)$.*

Proof of Lemma 1. The output equals x iff x is drawn and accepted, which has probability $q(x)\min(1, p(x)/q(x)) = \min(q(x), p(x))$. A residual draw is taken with probability $1 - \sum_y \min(q(y), p(y)) = \sum_y (p(y) - q(y))_+$, and it returns x with probability $r(x)$, contributing $(p(x) - q(x))_+$. The total is $\min(q(x), p(x)) + (p(x) - q(x))_+ = p(x)$. □

Proof of Theorem 1. We show by induction on the committed length that the committed prefix is distributed as $\prod_k \pi_{\mathrm{AR}}(o_k \mid o_{<k})$, which is π_{AR} by the chain rule. The base case is the empty prefix. For the inductive step, suppose the prefix $o_{<t}$ is exact and consider the next block starting at $p = t$.

Position p. The draft pass reads q_p from the input $[c, o_{<p}, [\mathtt{MASK}], \ldots]$. This input has no revealed block tokens, so q_p is exactly the AR-order conditional $\pi_{\mathrm{AR}}(\cdot \mid o_{<p})$. SPEAR commits $\tilde{o}_p \sim q_p$, which is therefore an exact draw from $\pi_{\mathrm{AR}}(\cdot \mid o_{<p})$.

Position $p + j$ for $1 \le j \le L - 1$, on the accept path. Condition on the event that positions $p, \ldots, p + j - 1$ were committed with values equal to the drafted tokens $\tilde{o}_{p:p+j}$ (this is the accept path. The first rejection ends the block). SPEAR draws $\tilde{o}_{p+j} \sim q_{p+j}$ and applies Lemma 1 with proposal q_{p+j} and target $\pi_{\mathrm{AR}}(\cdot \mid o_{<p}, \tilde{o}_{p:p+j})$. The target conditional is supplied by verification row j, which reveals exactly $\tilde{o}_{p:p+j}$. On the accept path the committed tokens equal the drafted tokens, so this is the correct conditioning. By Lemma 1 the committed token is an exact draw from $\pi_{\mathrm{AR}}(\cdot \mid o_{<p}, o_{p:p+j})$, regardless of the proposal q_{p+j}.

First rejection. At the first rejected position the committed token is the residual draw, which follows the residual law r, not the target. What Lemma 1 supplies is that the *mixture* of the accept and reject branches, before conditioning on which occurred, has the target law $\pi_{\mathrm{AR}}(\cdot \mid o_{<p}, \tilde{o}_{p:p+j})$, and that mixture is what the induction needs, since the committed prefix is produced by the unconditional procedure. The block then ends, so no later position in the block conditions on an incorrect prefix.

Bonus token. If all L positions are accepted, SPEAR draws one further token directly from $\pi_{\mathrm{AR}}(\cdot \mid o_{<p}, o_{p:p+L})$, read from the bonus verification row, which is exact by definition.

In all cases each committed token has law $\pi_{\mathrm{AR}}(\cdot \mid \text{committed prefix})$, so by the inductive hypothesis the extended prefix is exact. The temperature τ enters only through the definition of π_{AR} and q and plays no role in the argument, so exactness holds at every $\tau > 0$. □

Remark on batched verification. All verification rows for a block are computed in one batched forward pass. Row j has a different revealed prefix, so it is a distinct input. Batching changes only the order of computation, not the logits, *provided the backbone is batch invariant in the sense of Proposition 1*; that is an assumption about the implementation, which holds for LLaDA and fails for Dream-7B, and which we test empirically by the reveal-zero check (Section D.2).

Proposition 1 (Batched verification is exact under batch invariance)**.** *Let $x^{(1)}, \ldots, x^{(L)}$ be the revealed-prefix inputs, where $x^{(j)}$ reveals $\tilde{o}_{p:p+j}$ and masks the remaining generation positions. Assume the backbone is* batch invariant*: every kernel it runs returns, for each row, the value it would return for that row alone, so no row's logits depend on which other rows share the batch. Then running the L inputs as a batch produces, at position $p + j$ of row j, the same logits as running $x^{(j)}$ alone.*

This is a hypothesis about the implementation, not a property of masked attention, and we test it on every backbone. It holds for LLaDA and fails for Dream-7B, whose kernels shift the row logits by

0.018 total variation at batch size 8 and so bias the sampler silently; verification there runs one row per pass (Appendix G.4).

Proof. As a map on real numbers the claim is immediate: there is no cross-sequence attention or normalization, so the forward map factorizes over batch elements and the logits at any position of row j are a function of $x^{(j)}$ alone. That is why the proposition is stated as an assumption about the *implementation* rather than proved outright. A real kernel may select a different algorithm for a different batch shape, changing reduction order, tiling or the split of a matrix multiply, and the resulting discrepancy is not bounded by the working precision. Dream-7B is the case in point: its rows move by 0.018 total variation at batch size 8, orders of magnitude beyond bfloat16 rounding, on every attention backend we tried. The hypothesis must therefore be tested per backbone, not assumed. We test it at every revealed-prefix depth: LLaDA-8B agrees to a maximum absolute log-probability difference of 0.000000 at all depths, and Dream-7B does not agree at any (Section D.2), which is why verification there runs one row per forward pass. □

Proposition 2 (Support condition)**.** *The residual resampling in Theorem 1 is well defined, because* $\mathrm{supp}(\pi_{\mathrm{AR}}) \subseteq \mathrm{supp}(q_{p+j})$ *at every position: both are softmaxes of finite logits at temperature* $\tau > 0$*, so both assign positive probability to every token in the vocabulary.*

Proof. For $\tau > 0$ the softmax of any finite logit vector is everywhere positive, so $q_{p+j}(y) > 0$ and $\pi_{\mathrm{AR}}(y) > 0$ for all y. The residual $(\pi_{\mathrm{AR}} - q_{p+j})_+$ is therefore a nonnegative function with positive total mass whenever $\pi_{\mathrm{AR}} \neq q_{p+j}$, and the normalized residual is a valid distribution. When $\pi_{\mathrm{AR}} = q_{p+j}$ the acceptance probability is one and no residual draw is taken. □

B.2 Sublinear Verification: Full Analysis

Unbiased estimator. For a block of length L with per-position log-ratios $w_1, \dots, w_{L-1}$ (and $w_0 = 0$ since $q_p = \pi_{\mathrm{AR}}$ at the block start), the block log-ratio is $W = \sum_{k=1}^{L-1} w_k$. Drawing m uniformly from $\{1, \dots, L-1\}$ and returning $(L-1)\, w_m$ gives $\mathbb{E}[(L-1)w_m] = (L-1) \cdot \frac{1}{L-1} \sum_k w_k = W$, so it is unbiased and uses a single position evaluation.

Bernoulli factory. To accept a block with probability proportional to e^W using only bounded unbiased estimates, we use the Poisson construction of Keane & O'Brien (1994); Nacu & Peres (2005). Draw $J \sim \mathrm{Poisson}(2B)$ and clipped estimates $\hat{w}_r = \mathrm{ClipB}\big((L-1)w_{m_r}\big) \in [-B, B]$ for $r = 1, \dots, J$, and accept with probability $\prod_{r=1}^{J} \frac{B+\hat{w}_r}{2B}$. Writing $\bar{w} = \mathbb{E}[\hat{w}]$, the expected acceptance probability is $\mathbb{E}_J \prod_r \mathbb{E}\big[\frac{B+\hat{w}}{2B}\big] = \sum_{j \geq 0} e^{-2B} \frac{(2B)^j}{j!} \big(\frac{B+\bar{w}}{2B}\big)^j = \exp(-2B + (B + \bar{w})) = e^{\bar{w}-B}$, which is proportional to $e^{\bar{w}}$. Clipping replaces W by $\bar{w} = \mathbb{E}[\mathrm{ClipB}((L-1)w_m)]$. The resulting bias is controlled in χ^2 divergence by the level B, following the analysis of Chen et al. (2026), and vanishes as $B \to \infty$.

Theorem 2 (Sublinear verification)**.** *Fix a clip level* B*. For a block* o *let* $\widehat{W}(o) = (L-1)w_m$ *with* m *uniform on* $\{1, \dots, L-1\}$*, let* $W(o) = \mathbb{E}[\widehat{W}(o)]$ *and* $\bar{w}(o) = \mathbb{E}[\mathrm{Clip}_B(\widehat{W}(o))]$*, and define the clipping bias* uniformly over blocks*,*

$$\varepsilon_B \;=\; \sup_o \big|W(o) - \bar{w}(o)\big| \;\leq\; \sup_o \mathbb{E}\big[(|\widehat{W}(o)| - B)_+\big].$$

Assume $J \sim \mathrm{Poisson}(2B)$ *is drawn exactly, without truncation. Then: (i) if* $\varepsilon_B = 0$ *the rejection loop outputs a block distributed exactly as the AR-order block distribution; (ii) in general its law* $\tilde{\pi}$ *satisfies* $\chi^2(\tilde{\pi} \,\|\, \pi_{\mathrm{AR}}) \leq (e^{2\varepsilon_B} - 1)^2$ *and* $\mathrm{TV}(\tilde{\pi}, \pi_{\mathrm{AR}}) \leq \frac{1}{2}(e^{2\varepsilon_B} - 1)$*; (iii) at fixed* B *the expected number of single-position evaluations per accepted block is*

$$\mathbb{E}[C] \;=\; \frac{2Be^B}{\mathbb{E}_{o \sim q}\big[e^{\bar{w}(o)}\big]} \;\in\; \big[\, 2Be^B e^{-\varepsilon_B},\; 2Be^B e^{\varepsilon_B} \,\big],$$

which equals $2Be^B$ *exactly when* $\varepsilon_B = 0$*, and is* $O(Be^{2B})$ *in the worst case. At fixed* B *only the worst-case bound* $O(Be^{2B})$ *is independent of* L*. The bracket is not, because it contains* ε_B*, which grows with* L *when* B *is held fixed, and the expected cost itself depends on* L *through the distribution of* $\bar{w}$*.*

The supremum in ε_B is what the proof uses: the normalizer is controlled only because $|\delta(o)| \le \varepsilon_B$ holds for *every* block, not on average over blocks.

Proof of the χ^2 bound. The loop accepts a block o with probability proportional to $e^{\bar{w}(o)}$ rather than the required $e^{W(o)}$, so writing $\delta(o) = W(o) - \bar{w}(o)$ the stationary law of the loop is $\tilde{\pi}(o) = \pi_{\mathrm{AR}}(o)e^{-\delta(o)}/Z$ with $Z = \sum_o \pi_{\mathrm{AR}}(o)e^{-\delta(o)}$. If $|\delta(o)| \le \varepsilon_B$ for every o then $Z \in [e^{-\varepsilon_B}, e^{\varepsilon_B}]$, so the ratio $r(o) = \tilde{\pi}(o)/\pi_{\mathrm{AR}}(o)$ lies in $[e^{-2\varepsilon_B}, e^{2\varepsilon_B}]$. Hence $\chi^2(\tilde{\pi}\|\pi_{\mathrm{AR}}) = \sum_o \pi_{\mathrm{AR}}(o)(r(o)-1)^2 \le (e^{2\varepsilon_B}-1)^2$, since $e^{2\varepsilon_B} - 1 \ge 1 - e^{-2\varepsilon_B}$, and $\mathrm{TV} = \frac{1}{2}\sum_o \pi_{\mathrm{AR}}(o)|r(o)-1| \le \frac{1}{2}(e^{2\varepsilon_B}-1)$. The bound on ε_B by the expected clipping excess is immediate from $|x - \mathrm{Clip}_B(x)| = (|x| - B)_+$ and Jensen. □

What ε_B is on a real model, and why this variant is not practical there. The bound is only as useful as ε_B is small, and ε_B is set by the tail of the per-position log-ratio. On LLaDA-8B with GSM8K at $\tau = 0.6$ and $L = 8$, that tail is heavy: the median $|w_k|$ is 0.001 nats and the ninetieth percentile is 0.71, but the ninety-ninth percentile is 23.1 and the maximum we observed is 36.3 (3203 verified positions). Since $\widehat{W} = (L-1)w_m$, clipping never binds only when $B \ge (L-1)\max_k |w_k|$, which here means $B \ge 254$ nats and an expected cost of order $e^B \approx 10^{110}$ position evaluations per accepted block. Even truncating at the ninety-ninth percentile leaves $B \approx 161$. There is no clip level that is simultaneously cheap and close to exact on this model.

We therefore report the sublinear variant as a construction with a proved cost property and a proved error bound, validated on a toy oracle whose log-ratios are $O(1)$, and *not* as a usable exact sampler for real diffusion language models. The batched verification of Algorithm 2 is what we use and what our experiments measure. The gap is a property of the heavy-tailed log-ratio rather than of the Bernoulli factory: any scheme that accepts a block from a bounded unbiased estimate of W faces the same tail, and a variance-reduced or depth-weighted estimator (Appendix J.4) changes the constant rather than the tail.

The implementation caps the Poisson draw. The identity behind part (i) assumes $J \sim \mathrm{Poisson}(2B)$ exactly. Our implementation draws J and clamps it at a maximum, so that a single unlucky draw cannot stall a block, and the clamped variable is no longer Poisson. Truncating the mixing distribution perturbs the acceptance probability in the same direction as clipping, adding a second bias term that Theorem 2 does not model: part (i) therefore describes the uncapped scheme, and the measured toy-oracle agreement in Figure 3 is evidence about the code we actually ran, not a guarantee: it is a finite-sample total variation on one small oracle, and it is confounded by the loop's exact fallback, which commits a token from q_0 after R rounds and is itself exact, so a configuration that falls back more often can look more accurate for reasons unrelated to the clip level. With $2B$ small the clamp binds rarely and the perturbation is correspondingly small, but we have not bounded it, and we note it here rather than leave the gap between the theorem and the implementation implicit. It does not affect any other result in the paper, because every reported experiment uses the batched verification of Algorithm 2 rather than this variant.

Why the theorem is stated in three parts. Part (iii) is the sublinear-verification property and part (ii) is its price, and the two interact. The block log-ratio W is a sum of $L-1$ per-position terms and is $O(L)$ in the worst case, so keeping the no-clipping condition of part (i) requires B to grow with L, and the e^B constant grows with it. The cost is independent of L only at fixed B, where the clip bias of part (ii) grows with L instead. We separate the parts because the phrasing "exact up to an error" is not a meaningful guarantee.

Rejection loop and cost. On rejection the block is redrawn from the cached proposal q and retested with fresh Poisson estimates, a proper rejection-sampling loop that targets π_{AR} over the block. The expected number of position evaluations per accepted block is the mean Poisson count $2B$ times the expected number of rounds. A block drawn from q is accepted with probability $e^{\bar{w}(o)-B}$, so the average acceptance rate is $e^{-B}\,\mathbb{E}_{o\sim q}[e^{\bar{w}(o)}]$ and the expected work is $2Be^B/\mathbb{E}_{o\sim q}[e^{\bar{w}(o)}]$. Only in the unclipped case does this collapse to $2Be^B$: there $\bar{w} = W$ and $\mathbb{E}_{o\sim q}[e^{W(o)}] = \sum_o q(o)\,\pi_{\mathrm{AR}}(o)/q(o) = 1$, so the acceptance rate is exactly e^{-B}. With clipping, $|W - \bar{w}| \le \varepsilon_B$ gives $\mathbb{E}_{o\sim q}[e^{\bar{w}}] \in [e^{-\varepsilon_B}, e^{\varepsilon_B}]$, which is the bracket in part (iii). The unclipped figure is $2Be^B$, not the $2Be^{2B}$ an earlier version of

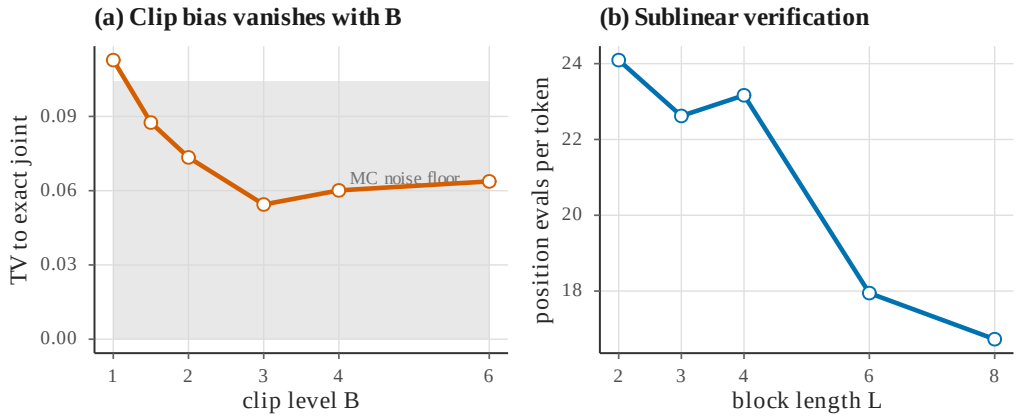


Figure 3: Sublinear verification on the toy oracle. (a) total variation to the exact joint against clip level B; (b) position evaluations per committed token against block length L.

this paper quoted: that figure came from bounding $1/Z$ by e^{2B} using only $\bar{w} \geq -B$, which is the worst case over blocks rather than the average. At fixed B the unclipped value $2Be^{B}$ and the worst case $O(Be^{2B})$ are both independent of the block length, which is the sublinear-verification property, and both are exponential in B. The clipped expected cost does not share that guarantee: it moves with L through ε_B.

Expected number of rounds. Each round accepts with probability $Z = e^{\bar{w}-B}$ in expectation (the Bernoulli-factory identity above), where $\bar{w} \in [-B, B]$ is the clipped mean log-ratio. The number of rounds until the first acceptance is geometric with success probability Z, so its expectation is $1/Z = e^{B-\bar{w}} \leq e^{2B}$. Each round draws $J \sim \text{Poisson}(2B)$ position evaluations in expectation, so the expected total per accepted block is $\mathbb{E}[\text{rounds}] \cdot \mathbb{E}[J] \leq 2B\, e^{2B}$, which is $O(Be^{2B})$. It has no dependence on the block length L. The clip level B therefore trades bias against cost: larger B reduces the χ^2 clip bias but raises the constant e^{2B}. A small B would suffice if the per-position log-ratio were bounded near its median, and on a real dLLM it is not: the distribution is heavy-tailed, with median $|w_k|$ of 0.001 nats against a maximum of 36.3, so ε_B is set by the tail and the clip needed to make it negligible at $L = 8$ is $B \approx 254$. An earlier version of this appendix argued the opposite from the $O(1)$ typical magnitude alone, which ignores the tail that ε_B actually depends on.

Toy-oracle validation. We validate exactness on a toy oracle masked diffusion model whose AR-order chain rule equals a known joint over three positions and vocabulary four (Dirichlet-sampled). We sample N sequences and compare to the joint in total variation. The sublinear variant gives TV $= 0.089$ at $N = 1500$ and clip level $B = 4$, within the Monte Carlo noise floor $1.3 \cdot \frac{1}{2}\sqrt{V^T/N} = 0.134$. The total variation tightens as B grows. We do not read that as confirming the clip-bias analysis: the implementation caps the Poisson draw and falls back to an exact stepwise commit after R rounds, and both make larger B look better for reasons unrelated to ε_B. The standard SPEAR sampler on the same oracle gives total variation within the same noise floor of the exact joint.

Figure 3 shows both properties empirically on the toy oracle. Panel (a) plots the total variation to the exact joint against the clip level B: it falls into the Monte Carlo noise band as B grows, which is consistent with the clip bias vanishing but does not establish it, because the Poisson cap and the exact fallback both move the same way with B. Panel (b) plots the number of position evaluations per committed token against the block length L. It decreases with L at fixed B, because the per-block cost is bounded independently of L and is amortized over more committed tokens. That holds only at fixed B, where the clip bias of Theorem 2(ii) grows with L instead.

Algorithm 1 states the sublinear-verification block step.

Algorithm 1 Sublinear-verification block acceptance (FORS-mode)

input: prefix $o_{<p}$, block length L, clip level B, head weight H, max rounds R
one draft forward; set q_k for $k = 0, \ldots, L-1$
for round $= 1$ to R **do**
  draw block $\tilde{o}_{0:L} \sim \prod_k q_k$
  $J \sim \text{Poisson}(2B)$; draw positions $m_1, \ldots, m_J \sim \text{Unif}\{1, \ldots, L-1\}$
  one batched forward over the J revealed-prefix rows; compute $\hat{w}_r = \text{ClipB}((L-1)w_{m_r})$
  accept the block with probability $\prod_{r=1}^{J}(B + \hat{w}_r)/2B$
  if accepted **then** commit $\tilde{o}_{0:L}$; **return**
end for
commit one exact leftmost token from q_0 // fallback after R rounds

Algorithm 2 SPEAR block step

input: prefix $o_{<p}$, block length L, temperature τ
draft: one forward on $[c, o_{<p}, [\texttt{MASK}]...]$; set $q_k = \text{softmax}(f_{\theta,k}/\tau)$; draw $\tilde{o}_k \sim q_k$ for $k = p..p{+}L{-}1$
verify: one batched forward over the L revealed-prefix inputs; obtain $\pi_{\text{AR}}(\cdot \mid o_{<p}, \tilde{o}_{p:p+j})$
commit $\tilde{o}_p$
for $j = 1$ to $L-1$ **do**
  accept $\tilde{o}_{p+j}$ with prob. $\min(1, \pi_{\text{AR}}(\tilde{o}_{p+j})/q_{p+j}(\tilde{o}_{p+j}))$
  if rejected **then** commit a draw from $(\pi_{\text{AR}} - q_{p+j})_+$; **break**
  else commit $\tilde{o}_{p+j}$
end for
if all accepted **then** commit one draw from $\pi_{\text{AR}}(\cdot \mid o_{<p}, \tilde{o}_{p:p+L})$

Algorithm 3 SPEAR with entropy-adaptive truncation and sparse head

input: prefix $o_{<p}$, block length L, temperature τ, entropy threshold θ (optional), head weight H
$x \leftarrow [c, o_{<p}, [\texttt{MASK}], \ldots]$; $h \leftarrow \text{hidden}(x)$ // one draft forward
$q_k \leftarrow \text{softmax}(H\, h_{p+k}/\tau)$ for $k = 0, \ldots, L-1$; draw $\tilde{o}_k \sim q_k$
if θ set and $\exists j \geq 1 : \text{entropy}(q_j) > \theta$ **then**
  $L \leftarrow$ (smallest such j) // truncate before the first fork
end if
commit $\tilde{o}_0$
build L rows, row j revealing $\tilde{o}_{0:j}$; $H_b \leftarrow \text{hidden}(\text{batch})$ // one batched verify forward
for $j = 1$ to $L-1$ **do**
  $\pi_j \leftarrow \text{softmax}(H\, H_b[j]_{p+j}/\tau)$; accept $\tilde{o}_j$ w.p. $\min(1, \pi_j(\tilde{o}_j)/q_j(\tilde{o}_j))$
  if rejected **then** commit draw from $(\pi_j - q_j)_+$; **break**; **else** commit $\tilde{o}_j$
end for
if all accepted **then** commit draw from π_L (bonus row)

B.3 ALGORITHMS

Algorithm 3 states SPEAR with entropy-adaptive truncation and the sparse head. The head H is applied only at queried positions: verification needs logits at L positions out of a canvas of several hundred, so materializing the full-vocabulary logits at every position wastes memory and time. We replace the model head with the identity, read post-final-norm hidden states, and apply the head weight only at the queried rows and positions. This is mathematically identical to the model head and about 10% faster in wall-clock.

B.4 COMPLEXITY ANALYSIS

We summarize the cost of each decoder for a generation of N tokens, in forward-pass count (NFE), per-call batch size, and peak activation memory.

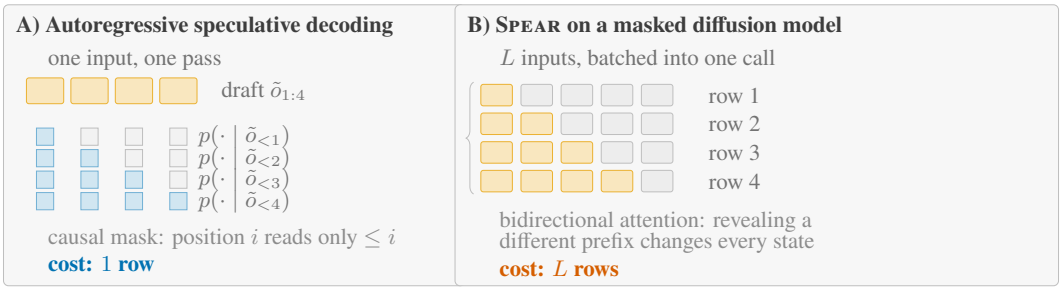


Figure 4: Single-pass verification in an AR model (A) against the L-row batch a dLLM needs (B).

Stepwise AR decoding. One forward pass per token, each on a single sequence of length $P + N$ where P is the prompt length. Total NFE is N, batch size 1, and peak activation memory is that of a single length-$(P+N)$ forward pass. It is exact.

SPEAR. Per block of length L that commits c tokens (with c close to L when acceptance is high), one draft pass on a single sequence and one verification pass on a batch of L rows. Over the generation, the number of blocks is about N/c, so the NFE is about $2N/c$ draft-plus-verify calls, and the total row-forward count (the compute-relevant quantity) is about $N(1 + L)/c$. Because $c \approx L$ at high acceptance, the row-forward count is about $N(1+L)/L \approx N$, so SPEAR does a comparable amount of compute to stepwise decoding while issuing far fewer distinct calls. Peak activation memory is that of an L-row batched forward pass, which is L times a single sequence. This is the binding resource at large L and the reason wall-clock turns over.

Sublinear variant. Per accepted block, one draft pass and an expected $2Be^B/\mathbb{E}_{o\sim q}[e^{\bar{w}(o)}]$ single-position evaluations, at most $O(Be^{2B})$, a worst-case bound that is independent of L at fixed B (Section B.2). The row-forward count per committed token therefore decreases with L (Figure 3b), at the cost of a higher constant from the Poisson intensity. This variant is preferable when memory, not compute, is the binding constraint, since it never forms an L-row batch.

Confidence decoding. With S steps, S forward passes on a single sequence, so NFE is S and batch size 1. It is lossy, with the per-token divergence of Section C.4.

C SELF-CONSISTENCY AND FIDELITY

C.1 EXTENDED SELF-CONSISTENCY

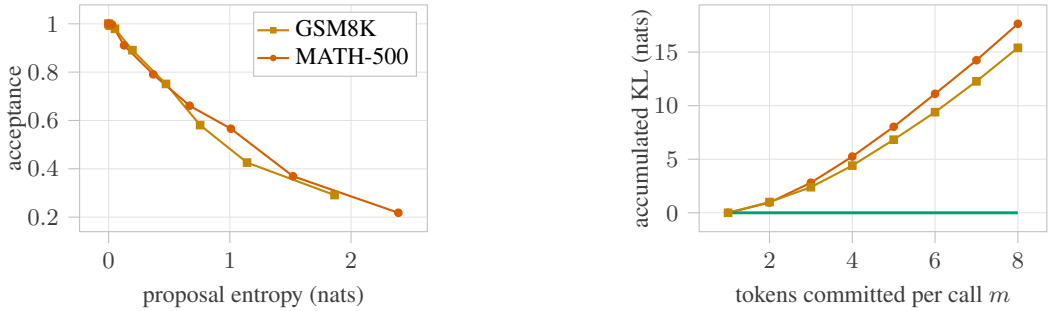


Figure 5: Self-consistency on LLaDA-8B. Left: acceptance against proposal entropy; right: divergence accumulated by committing m uncorrected tokens per call.

Standard errors are over 16 prompts for the LLaDA-8B GSM8K row and 24 for the others, and the reveal-zero exactness check was 0.0 in every cell. The MATH-500 and GSM8K intervals at $\tau = 0.6$ are about two standard errors apart.

Table 6: Mean per-token acceptance across models and datasets ($L = 8$).

model	dataset	accept (τ=0.6)	accept (τ=1.0)
LLaDA-8B-Instruct	GSM8K	0.909 ± 0.011	0.866 ± 0.012
LLaDA-1.5	GSM8K	0.914 ± 0.005	0.860 ± 0.008
Dream-7B	GSM8K	0.896 ± 0.008	0.817 ± 0.015
LLaDA-8B-Instruct	MATH-500	0.879 ± 0.010	0.799 ± 0.018

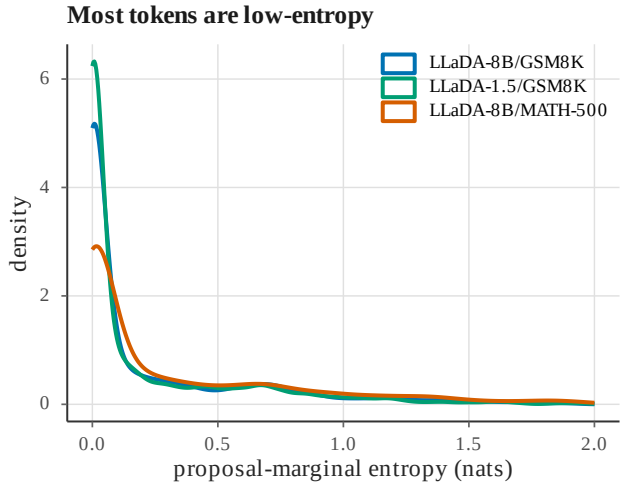


Figure 6: Distribution of proposal-marginal entropy across cells.

Table 7: Acceptance by proposal-entropy quartile.

cell	Q1	Q2	Q3	Q4	Spearman	mean KL
LLaDA-8B / GSM8K	1.000	0.999	0.946	0.666	-0.391	1.16
LLaDA-1.5 / GSM8K	1.000	0.999	0.960	0.689	-0.367	1.19
Dream-7B / GSM8K	1.000	0.991	0.859	0.562	-0.487	1.48
LLaDA-8B / MATH-500	1.000	0.995	0.911	0.603	-0.414	1.44

Run-to-run variation. Acceptance is a per-prompt average, and our sweeps use different prompt sets, so the same nominal configuration appears with slightly different values in different tables. For LLaDA-8B, GSM8K, $L = 8$, $\tau = 0.6$, the timing sweep gives 0.909 ± 0.011 over 16 prompts (Table 15), a 32-prompt run gives 0.905, and the temperature and block sweeps give 0.916 and 0.915 (Tables 8 and 17). The full spread is 0.011, about one standard error, so these are the same measurement on different prompts rather than inconsistent results. We report the standard error wherever acceptance carries a claim.

Table 7 extends Section 4.1 across all three model-dataset cells. In every cell acceptance is near one in the low-entropy quartiles and falls to 0.56 to 0.69 in the top quartile. The Spearman correlation between acceptance and proposal entropy is between -0.37 and -0.53, and it is strongest on Dream-7B, the one model outside the LLaDA lineage. The mean per-token $\mathrm{KL}(q \,\|\, \pi_{\mathrm{AR}})$ rises with task difficulty. The forking-token mechanism is therefore not specific to one model or dataset. Acceptance also decreases smoothly with temperature: at $L = 8$ on LLaDA-8B, GSM8K, acceptance is 0.909 at $\tau = 0.6$ and 0.866 at $\tau = 1.0$, with mean KL 1.15 and 1.02 respectively.

Figure 6 plots the distribution of proposal-marginal entropy for each cell. The mass concentrates near zero: most positions have a low-entropy marginal that draft and verify agree on, and the high-entropy forking tokens that drive rejections are a minority. This is why the per-token acceptance rate is high despite the large acceptance drop in the top entropy quartile.

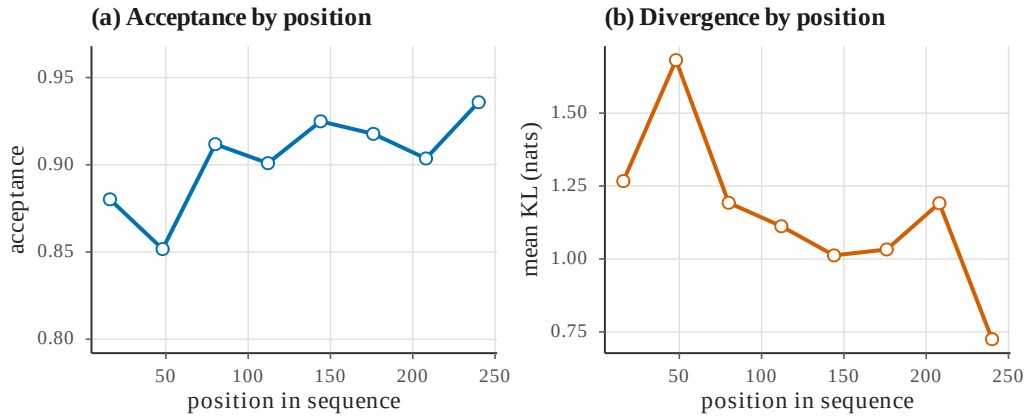


Figure 7: Acceptance (a) and per-token $\mathrm{KL}(q \,\|\, \pi_{\mathrm{AR}})$ (b) by position in the sequence.

Table 8: Temperature sweep on GSM8K ($L = 8$).

model	metric	τ=0.2	0.4	0.6	0.8	1.0	1.2
LLaDA-8B	accept	0.926	0.925	0.916	0.887	0.845	0.691
LLaDA-8B	tok/fwd	3.55	3.55	3.49	3.18	2.86	2.04
LLaDA-8B	speedup	1.08	1.02	1.00	0.91	0.82	0.58
LLaDA-1.5	accept	0.935	0.919	0.915	0.877	0.863	0.722
LLaDA-1.5	tok/fwd	3.71	3.55	3.49	3.06	2.97	2.15

C.2 Acceptance Along the Sequence

Figure 7 plots the acceptance rate and mean per-token divergence as a function of the absolute position in the generated sequence. Acceptance is high and roughly flat across the sequence, between 0.85 and 0.94, with a mild increase toward the end where continuations become more constrained. The divergence tracks the inverse: it is highest early, where the reasoning trajectory is least determined and forking tokens are densest, and lowest late. Self-consistency is therefore a property of the whole generation rather than of a particular region, which is consistent with the block sizes committed by SPEAR being stable across the sequence.

C.3 Temperature Dependence

Table 8 sweeps the sampling temperature from 0.2 to 1.2 at $L = 8$ on LLaDA-8B, GSM8K. Acceptance is highest at low temperature, above 0.92 for $\tau \leq 0.4$, and falls to 0.69 at $\tau = 1.2$. At low temperature both the one-shot marginal and the AR-order conditional concentrate on the same top token. They therefore agree on the sampled token even though their KL in nats is large. As temperature rises the distributions spread and disagreement grows, which lowers acceptance and tokens per forward pass. The practical implication is that SPEAR is most efficient in the low-temperature regime typical of reasoning evaluation, and degrades gracefully toward high temperature. The turnover is abrupt only at the highest temperature we test. Acceptance is flat from $\tau = 0.2$ to $\tau = 0.6$ and declines gently through $\tau = 1.0$ before dropping at $\tau = 1.2$. Across the range of temperatures used for reasoning ($\tau \leq 1.0$) the efficiency is stable. This matches the intuition that a well-trained masked diffusion model is close to order-agnostic on the bulk of tokens. It becomes order-sensitive only when the sampling temperature is high enough to expose the forking positions. Figure 8 plots acceptance and efficiency against temperature for both models.

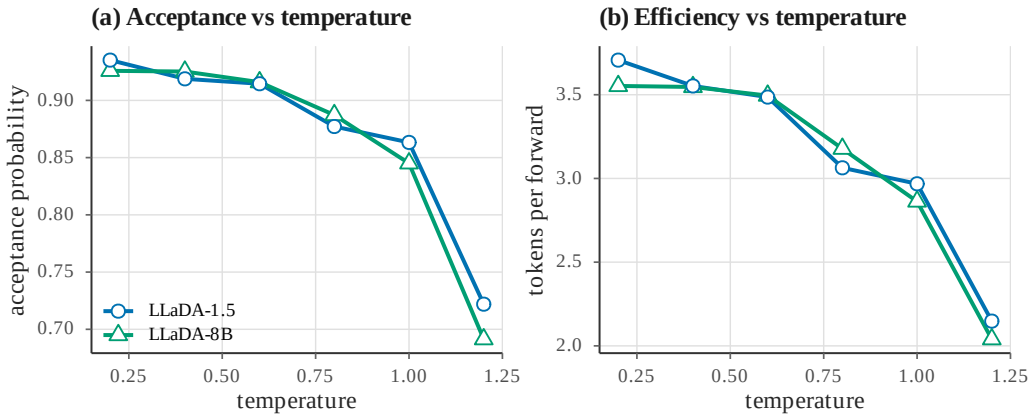


Figure 8: Temperature dependence of (a) acceptance and (b) tokens per forward pass.

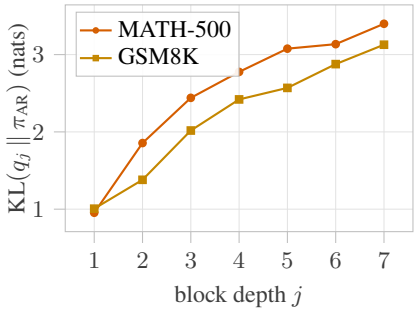


Figure 9: Per-token $\mathrm{KL}(q_j \,\|\, \pi_{\mathrm{AR}})$ by block depth j on LLaDA-8B.

C.4 Distributional Cost of Parallelism, per Cell

Table 9 reports the per-token $\mathrm{KL}(q_{p+j} \,\|\, \pi_{\mathrm{AR}})$ by block depth j for each cell, and Figure 9 plots it. There are two distinct quantities here and an earlier version of this paper conflated them. A SPEAR block stops committing at its first rejection, so if one logs a depth only when the accept path reaches it, the depth-j average is taken over blocks that already survived $j-1$ acceptance checks, which are exactly the blocks where proposal and target agreed. That conditional profile is the right thing for reasoning about SPEAR's own cost. It is the wrong thing for the question the introduction asks, which is what a *lossy* decoder loses by committing m tokens regardless of agreement.

Recovering the unconditional profile costs nothing. Verification computes all L rows in one forward pass before the accept scan begins, so every depth's divergence is already available and only the logging stopped early; an earlier version of this appendix claimed that collecting these measurements would require verification the sampler avoids, and that was simply wrong. We now record every depth and flag which ones the accept path reached. The gap is a factor of two: on GSM8K the accumulated divergence at $m = 8$ is 15.4 nats unconditionally against 7.7 on the accept path alone, and on MATH-500 17.6 against 9.0. All divergence figures quoted in this paper are the unconditional ones, which are the larger and the relevant ones. The per-token error is 0.8 to 1.3 nats at depth one and grows with depth. A decoder that commits m tokens per forward pass without correction accumulates the partial sum $\sum_{j<m}$ of these values (Figure 5, right). The error is larger throughout on MATH-500, consistent with the higher density of forking tokens on harder problems. SPEAR corrects every drafted token, so its distributional error is zero at every m in every cell.

Table 9 is measured on the accept path. The unconditional profile, which is what a lossy decoder meets, is steeper: $1.01, 1.38, 2.02, 2.42, 2.57, 2.88, 3.13$ nats on GSM8K and $0.95, 1.86, 2.44, 2.78, 3.08, 3.14, 3.40$ on MATH-500, accumulating to 15.4 and 17.6 nats at $m = 8$.

Table 9: Per-token KL($q_{p+j} \,\|\, \pi_{\mathrm{AR}}$) in nats by block depth, on the accept path.

cell	j=1	2	3	4	5	6	7
LLaDA-8B / GSM8K	0.84	1.12	1.19	1.18	1.33	1.34	1.37
LLaDA-1.5 / GSM8K	0.87	1.23	1.39	1.44	1.25	1.03	1.12
LLaDA-8B / MATH-500	1.30	1.76	1.40	1.28	1.34	1.61	1.35

D PASS@k RESULTS

D.1 FULL PASS@k TABLES

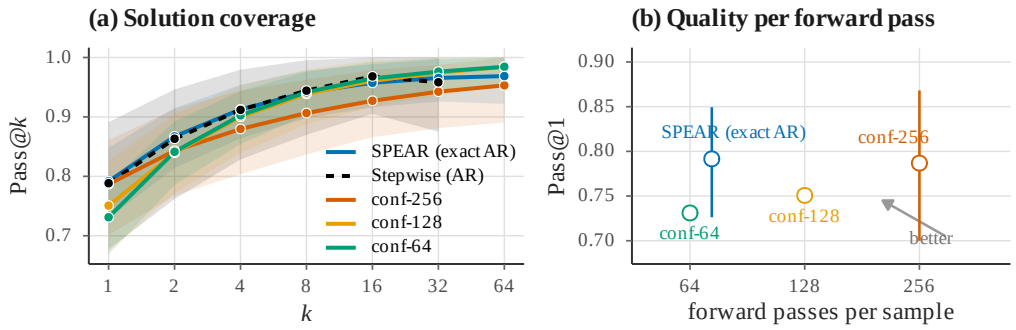


Figure 10: Pass@k against k (left) and Pass@1 against forward passes per sample (right) on GSM8K.

Table 10: Pass@1 at a matched decoding budget on the generalization cells.

model	dataset	method	n	Pass@1	fwd/sample
LLaDA-1.5	GSM8K	SPEAR (exact)	32	**0.745**	75
LLaDA-1.5	GSM8K	FASTDLLM	32	0.719	75
LLaDA-1.5	GSM8K	CONF64	32	0.683	64
LLaDA-1.5	GSM8K	SSD (greedy)	16	0.688	76
Dream-7B	GSM8K	FASTDLLM	32	**0.792**	83
Dream-7B	GSM8K	SPEAR (exact)	32	0.766	353
Dream-7B	GSM8K	CONF64	32	0.520	64
LLaDA-8B	MATH-500	FASTDLLM	500	**0.352**	98
LLaDA-8B	MATH-500	SPEAR (exact)	500	0.333	83
LLaDA-8B	MATH-500	CONF64	500	0.286	64

In Table 10, SSD is deterministic and is scored on a 16-problem subset.

Tables 11 and 12 give the full Pass@k curves for the two generalization cells, complementing Table 23, and Figure 11 plots them. Each row is computed on its own problem set, whose size is the n column, so rows with different n must not be read against each other. In particular STEPWISE, which costs 256 forward passes per sample, was run on only the first 16 problems of each cell. Read as printed, STEPWISE appears to trail SPEAR by 0.083 on LLaDA-1.5 and 0.019 on MATH-500. Both gaps reflect which problems each sampler was run on. The paired comparison on shared problems removes them: 0.671 against 0.662 on LLaDA-1.5 and 0.318 against 0.314 on MATH-500 (Table 13). The SPEAR and CONF64 rows carry the same n in both cells, so those rows are directly comparable. The paired SPEAR minus CONF64 Pass@1 difference is $+0.062$ (CI $[+0.019, +0.114]$) on LLaDA-1.5 and $+0.047$ (CI $[+0.037, +0.058]$) on MATH-500, both excluding zero.

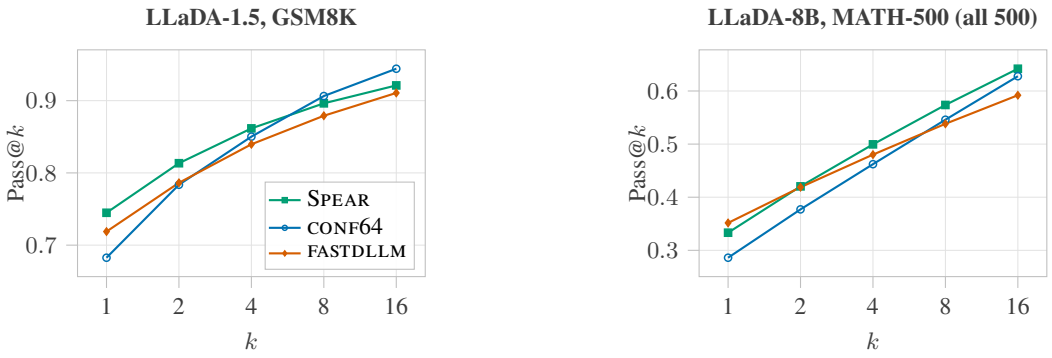


Figure 11: Pass@k on LLaDA-1.5 GSM8K and LLaDA-8B MATH-500.

Table 11: Pass@k on LLaDA-1.5, GSM8K.

method	n	P@1	P@2	P@4	P@8	P@16	P@32	fwd
SPEAR	32	0.745	0.813	0.862	0.896	0.921	0.936	75
CONF64	32	0.683	0.784	0.850	0.906	0.944	0.969	64
SSD	16	0.688	n/a	n/a	n/a	n/a	n/a	76
STEPWISE	16	0.662	0.737	0.797	0.851	0.898	0.938	256

Table 12: Pass@k on all 500 MATH-500 problems with LLaDA-8B.

method	n	P@1	P@2	P@4	P@8	P@16	fwd
SPEAR	500	0.333	0.420	0.499	0.574	0.642	83
FASTDLLM	500	0.352	0.419	0.480	0.538	0.592	98
CONF64	500	0.286	0.377	0.462	0.546	0.628	64
STEPWISE	16	0.314	0.416	0.514	0.610	0.712	256

D.2 REAL-DATA EXACTNESS CHECK

SPEAR is a proven-exact sampler of π_{AR}. On the real model we confirm this two ways. First, the reveal-zero verification row reveals no block tokens, and it reproduces the draft marginal to a maximum absolute log-probability difference of 0.0 in every configuration and every cell. This is the empirical test of the batch-invariance hypothesis that Proposition 1 assumes, and it passes on LLaDA and fails on Dream-7B; the proposition's conclusion is claimed only for backbones where it passes. It is a weak check on the sampler, however, because the reveal-zero row has the same input as the draft pass, so it cannot detect an error in the rows that reveal one or more drafted tokens. We therefore ran the stronger check directly: for every revealed-prefix depth $j = 0, \ldots, L$ we recompute that row alone and compare it with the same row inside the L-row batch. On LLaDA-8B the maximum absolute log-probability difference is 0.000000 at every depth, so batching is exact for the rows the sampler actually reads and not merely for the trivial one. On Dream-7B it fails at every depth, with maxima of 0.54 to 1.34 nats and a mean near 0.25, which is the quantitative form of the batch-invariance failure that forces one row per pass there. `scripts/verify_depths.py` reproduces both.

Second, and substantively, SPEAR matches STEPWISE on the problems the two share, which is the comparison Table 13 reports. Restricting to shared problems matters: STEPWISE was run on a subset of each cell, and comparing the two methods on their own problem sets instead produces apparent gaps of 0.083 and 0.019 (Tables 11 and 12), which reflect which problems each was run on rather than any difference between the samplers. Paired on shared problems, the differences are -0.001, $+0.009$, -0.018 and $+0.004$ Pass@1, and all four intervals include zero. An earlier version of this paper reported a MATH-500 interval that excluded zero; that was an artifact of the answer grader described in Appendix G.3, and it disappears once both samplers are graded correctly.

Table 13: Paired SPEAR minus STEPWISE Pass@1 on shared problems.

model	dataset	n	SPEAR	STEPWISE	Δ	95% CI
LLaDA-8B	GSM8K	64	0.778	0.779	-0.001	$[-0.015, +0.013]$
LLaDA-1.5	GSM8K	16	0.671	0.662	$+0.009$	$[-0.009, +0.028]$
Dream-7B	GSM8K	32	0.766	0.783	-0.018	$[-0.047, +0.008]$
LLaDA-8B	MATH-500	16	0.318	0.314	$+0.004$	$[-0.031, +0.043]$

D.3 DISTRIBUTIONAL EQUIVALENCE ON THE REAL MODEL

Pass@1 agreement compares two samplers through a single scalar. A sharper test compares the whole empirical distribution over extracted answers. For each shared problem we form the answer histogram of each sampler and compute the total-variation distance between them. Two finite samples of the *same* distribution do not give TV $= 0$, so the reference is not zero: we build it by pooling both samplers' draws for that problem, splitting the pool at random into two groups of the original sizes, and recomputing the distance. The gap between the observed distance and this permutation reference is the quantity of interest, and we test it with a paired sign-flip permutation test over problems.

Table 14 reports the result. For the two exact samplers the excess is at or below zero in all three cells, which is what Theorem 1 predicts. For SPEAR against the lossy decoders the excess is positive and significant on GSM8K with LLaDA-8B (96 problems) and on all 500 MATH-500 problems, against CONF64 and against the threshold decoder, so the test has power to detect the difference it is meant to detect. The largest excess is between STEPWISE and CONF256 at a matched budget of 256 forward passes, where sampling order is the only difference: the two orders induce measurably different answer distributions even though their Pass@1 values are close (Section 4.2).

One entry needs comment. On LLaDA-1.5 the SPEAR against STEPWISE excess is -0.021 with $p = 0.005$. A negative excess means the two samplers agree *more* than two halves of a pooled sample do, which cannot be evidence that their distributions differ. With 16 problems the permutation reference is itself noisy, and we read this as a small-sample artifact in the direction of agreement.

Table 14: Distributional equivalence of answer distributions on the real model.

cell	sampler A	sampler B	n	TV	reference	excess	p
GSM8K / LLaDA-8B	SPEAR	STEPWISE	48	0.101	0.104	-0.002	0.65
GSM8K / LLaDA-1.5	SPEAR	STEPWISE	16	0.105	0.126	-0.021	0.005
MATH-500	SPEAR	STEPWISE	16	0.226	0.223	$+0.002$	0.92
GSM8K / LLaDA-8B	SPEAR	CONF64	96	0.170	0.116	$+0.054$	< 0.001
GSM8K / LLaDA-1.5	SPEAR	CONF64	32	0.182	0.152	$+0.029$	0.067
MATH-500	SPEAR	CONF64	500	0.447	0.396	$+0.051$	< 0.001
MATH-500	SPEAR	FASTDLLM	500	0.400	0.318	$+0.081$	< 0.001
GSM8K / LLaDA-8B	STEPWISE	CONF256	48	0.191	0.085	$+0.106$	< 0.001

E TIMING AND COST STRUCTURE

E.1 TIMING AND ADAPTIVE TRUNCATION

Table 15 gives the full block-length sweep at both temperatures, and Table 16 the entropy-adaptive truncation sweep, extending Table 6. Tokens per forward pass rise with block length while wall-clock speedup falls below parity, because a block of L tokens costs an L-row batch under bidirectional attention. Adaptive truncation keeps the wall-clock gain positive by not verifying tokens that are likely to reject. Stepwise AR decoding runs at 6.12s ($\tau = 0.6$) and 5.62s ($\tau = 1.0$) per 256-token sample.

The reveal-zero exactness check in Table 15 was 0.0 in every row.

Table 15: Block-length timing sweep (LLaDA-8B, GSM8K, 16 prompts).

τ	L	accept	tok/fwd	wall speedup
0.6	4	0.918 ± 0.008	2.26	1.17×
0.6	8	0.909 ± 0.011	3.45	1.06×
0.6	16	0.903 ± 0.009	4.58	0.77×
1.0	4	0.871 ± 0.016	2.11	1.01×
1.0	8	0.866 ± 0.012	3.01	0.83×
1.0	16	0.835 ± 0.017	3.28	0.49×

Table 16: Entropy-adaptive truncation at nominal $L = 16$ (wall-clock relative to STEPWISE: 6.12 s at τ=0.6, 5.62 s at τ=1.0).

threshold (nats)	τ	accept	tok/fwd	wall (s)	speedup
0.35	0.6	0.985	3.52	4.88	1.25×
0.35	1.0	0.986	2.22	4.89	1.15×
0.80	0.6	0.946	3.99	5.44	1.12×
0.80	1.0	0.953	2.75	5.07	1.11×

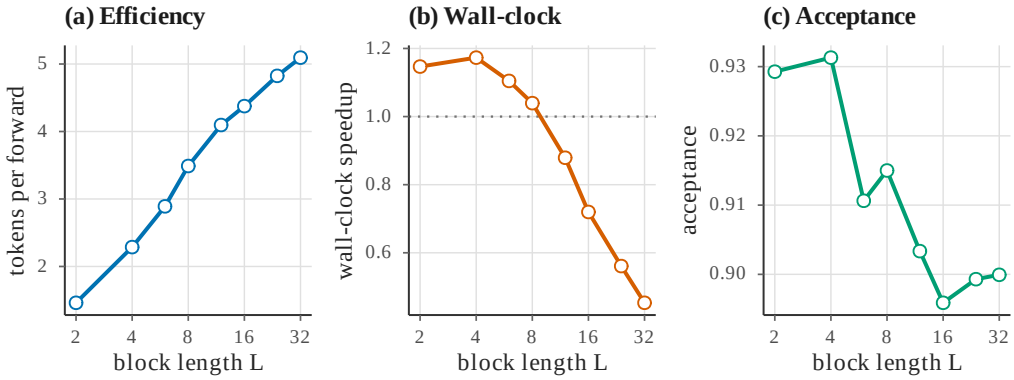


Figure 12: Block-length sweep: (a) tokens per forward pass, (b) wall-clock speedup, (c) acceptance.

E.2 BLOCK-LENGTH SCALING

Figure 12 and Table 17 sweep the block length L from 2 to 32 at $\tau = 0.6$. Three quantities move together. Tokens per forward pass rise monotonically, from 1.46 at $L = 2$ to 5.09 at $L = 32$, because a larger block commits more tokens per draft-verify pair. Wall-clock speedup rises to a peak of 1.17× near $L = 4$ and then falls below parity, crossing one near $L = 10$, because the L-row verification batch becomes compute-bound (Section E.5). Acceptance is stable throughout, between 0.90 and 0.93, so the self-consistency of the model does not degrade as the block grows. The limit on useful block length is the compute cost of verification, not a loss of agreement. The mean number of tokens committed per block rises to 10.5 at $L = 32$, close to the block length, confirming that most drafted tokens are accepted even at large L. The practical recommendation that follows is simple. Use a small block, L between 4 and 8, when wall-clock is the objective. That is where the speedup is positive, and the tokens-per-forward gain is already most of what larger blocks provide. Use a large block only when the forward-pass count itself is the objective, for instance when each forward pass carries a fixed overhead that dominates. In that case committing ten tokens per two forward passes is worthwhile despite the higher per-call compute.

E.3 SEQUENCE-LENGTH SCALING

Table 18 and Figure 13 sweep the generation length at $L = 8$, $\tau = 0.6$. Acceptance rises with generation length, from 0.846 at 128 tokens to 0.943 at 512, because longer contexts constrain each

Table 17: Block-length sweep (LLaDA-8B, GSM8K).

L	2	4	6	8	12	16	24	32
accept	0.929	0.931	0.911	0.915	0.903	0.896	0.899	0.900
tok/fwd	1.46	2.29	2.89	3.49	4.10	4.38	4.82	5.09
speedup	1.15	1.17	1.10	1.04	0.88	0.72	0.56	0.45
commit	2.92	4.57	5.82	7.01	8.29	9.01	9.96	10.53

Table 18: Sequence-length sweep (LLaDA-8B, GSM8K, $L = 8$).

generation length	128	256	512
accept	0.846	0.915	0.943
tok/fwd	2.81	3.49	3.76
speedup	1.20	1.04	0.93

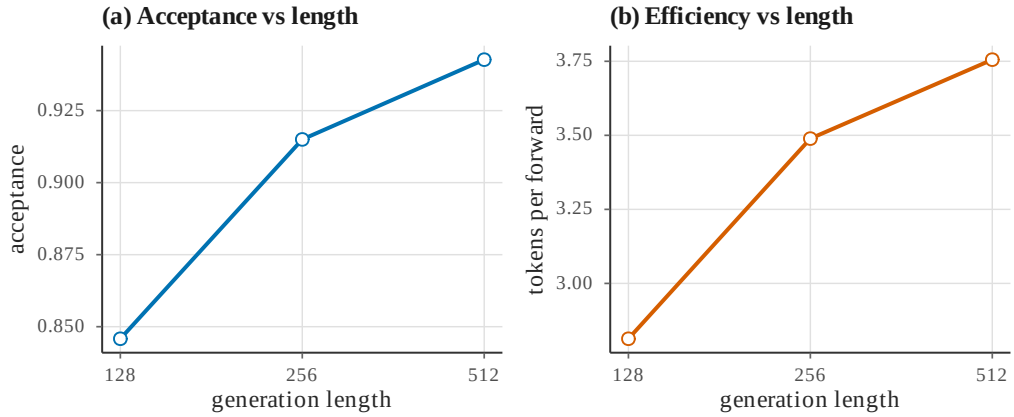


Figure 13: Sequence-length scaling of (a) acceptance and (b) tokens per forward pass.

continuation more tightly and reduce the number of high-entropy forking positions per token. Tokens per forward pass rise accordingly. Wall-clock speedup falls from 1.20 at 128 tokens to 0.93 at 512, because a longer canvas makes each forward pass more expensive and the L-row verification batch is correspondingly larger. Self-consistency therefore improves with length while the wall-clock cost of exploiting it grows, which is consistent with the compute-bound analysis of Section E.5.

E.4 What Determines Acceptance

The efficiency of SPEAR is governed by how close the one-shot proposal marginals are to the exact AR-order conditionals. The right way to state that relationship is an identity rather than an empirical law. For the speculative rule the per-position acceptance probability is

$$\alpha = \sum_{x} \min\{\pi_{\mathrm{AR}}(x), q(x)\} = 1 - \mathrm{TV}(\pi_{\mathrm{AR}}, q), \tag{7}$$

so mean acceptance and mean total variation are the same measurement, and tokens per forward pass follow directly from α. Total variation, not any other divergence, is the quantity that determines acceptance.

An earlier version of this paper claimed a decreasing empirical relationship between mean $\mathrm{KL}(q \,\|\, \pi_{\mathrm{AR}})$ and acceptance, and read it as a predictive law. That claim was wrong, and we correct it here. Across our sweep configurations the two are in fact weakly *positively* correlated (Pearson $r = +0.16$ over the 38 temperature, block-length, sequence-length and model configurations), so KL does not predict acceptance, and the fitted line in the figure we previously showed sloped the opposite way to the text. The reason is visible in the temperature sweep: at $\tau = 0.2$ the one-shot

marginal and the AR-order conditional both concentrate on the same token, so acceptance is high (0.926) even though their KL is large (2.35 nats), because KL is dominated by the tails while acceptance depends only on the overlap. By Eq. 7 the correct predictor is TV, and the correct diagnostic is therefore to measure per-position overlap, which is what the acceptance rate already reports.

What survives, and what we do claim, is the descriptive part: acceptance is high on both models and all three cells, it falls with temperature, and rejections concentrate at high-entropy positions (Table 7). Those statements rest on direct measurement rather than on a divergence-to-acceptance relationship.

E.5 Why Bidirectional Attention Forbids Single-Pass Verification

In autoregressive speculative decoding, a draft of L tokens is verified in one forward pass because the causal attention mask makes the hidden state at position i depend only on positions $\leq i$. Placing the draft tokens $\tilde{o}_1, \ldots, \tilde{o}_L$ into a single causal forward pass therefore yields, at position i, exactly the distribution $p(\cdot \mid \tilde{o}_{<i})$, which is the target conditional for verifying $\tilde{o}_i$. One pass scores every prefix simultaneously.

A masked diffusion model uses full bidirectional attention: the hidden state at every position depends on every other position. Consider verifying the drafted token at block position j, which requires the conditional $\pi_{\text{AR}}(\cdot \mid o_{<p}, \tilde{o}_{p:p+j})$, that is, the distribution when exactly the first j block tokens are revealed and the rest are masked. The input that produces this conditional has $\tilde{o}_{p:p+j}$ filled in and $\tilde{o}_{p+j:p+L}$ masked. For a different j', the input differs in which positions are masked, and because attention is bidirectional, the hidden state at position $p + j$ differs between the two inputs. There is no single input whose one forward pass yields the correct conditional at every block position simultaneously: the required inputs are different tensors, one per revealed prefix.

SPEAR therefore issues L inputs, the j-th revealing $\tilde{o}_{p:p+j}$, and runs them as one batched forward pass. This is L times the compute of a single sequence, in contrast to the single causal pass of AR speculative decoding. The saving relative to running the L inputs sequentially is that the batch executes in one kernel launch and is memory-bound at moderate L. Wall-clock therefore grows slower than L until the batch becomes compute-bound. This is the regime visible in Table 15, where tokens per forward pass keep rising with L while wall-clock speedup falls below one. The forward-pass count (NFE) and the wall-clock time are therefore different metrics for SPEAR, and we report both. A verification scheme that reused a shared key-value cache across the revealed prefixes would close this gap. The bidirectional dependence of the prefix representation on the masked suffix makes such a cache non-trivial. We leave it to future work.

F Reinforcement Learning

F.1 Reinforcement Learning Details

Setup. We train the AR-order policy with GRPO (Shao et al., 2024) using LoRA (Hu et al., 2022) of rank 32. LoRA is applied to the attention projections (`q_proj`, `k_proj`, `v_proj`, `att_proj`, `attn_out`) and feed-forward projections (`ff_proj`, `up_proj`). We use group size 6, learning rate 10^{-5}, clip 0.2, and generation length 256. The reward is the verified GSM8K answer match. The vocabulary head is frozen and detached (the sparse head), so gradients flow through the LoRA adapters into the shared hidden states. Rollouts are produced by the sampler under test. The policy-gradient log-probabilities are computed by a batched masked-prefix forward, with the position subset selected by the ρ-estimator mode. Training draws from the GSM8K train split. The held-out evaluation set is 128 GSM8K test problems, so there is no leakage.

Rollout study. An earlier ablation over rollout mode and ρ-estimator, at 120 steps, was run before the canvas-conditioning fix described below, so its learner scored EOS-trimmed canvases while its rollouts used the full generation canvas. We do not report its numbers, because they cannot be compared with the corrected 600-step run and re-running five conditions would spend compute we would rather put into independent benchmark items. The one conclusion that does not depend on the fix is that all rollout modes train to completion and that confidence rollouts are the cheapest per step, which is a statement about cost, not about learning.

Table 19: Held-out Pass@1 every 50 GRPO steps (LLaDA-8B, 128 GSM8K problems).

step	0	50	100	150	200	250	300	350	400	450	500	550	600
SPEAR	.750	.773	.766	.750	.797	.750	.703	.766	.758	.789	.781	.742	.727
conf	.750	.734	.773	.734	.695	.727	.695	.742	.742	.734	.742	.773	.734

Scaled comparison. SPEAR returns the AR-order log-probabilities of accepted tokens as a by-product of verification, so a GRPO rollout and the log-probability in its ratio come from one pass. We ran a focused 600-step comparison of SPEAR rollouts against confidence rollouts with a held-out evaluation every 50 steps (Figure 14). Post-training held-out Pass@1 is 0.727 for SPEAR and 0.734 for confidence, a difference of -0.008. We report this descriptively and run no significance test on it. An earlier version quoted a Welch p of 0.30 and called the two conditions statistically indistinguishable, but that test treats the checkpoints as independent samples, which is the same assumption we reject below when declining to state a detectable effect size. Neither condition shows a clear learning trend. The 13 evaluations per condition are checkpoints along a *single* training trajectory, scored on the same 128 held-out problems. They are correlated repeated measurements, so treating them as independent samples, as an earlier version of this appendix did, understates the variance and produces a detectable-effect figure that the design does not support. A defensible power statement needs several independent seeds per condition, and preferably a setup in which at least one condition demonstrably learns, so that the comparison has something to detect. We ran one seed per condition at 600 steps with one problem per step, which our compute allowed, and neither condition shows a clear trend. For scale, a single evaluation on 128 problems carries a binomial standard error of 0.038 at an accuracy of 0.75, which is larger than the 0.011 difference between the conditions, so the individual points in Table 19 move by several points between checkpoints for reasons unrelated to the rollout sampler. The honest reading is that this experiment demonstrates that exact rollouts are usable in GRPO and detects no quality difference at the one scale we ran. Acceptance during SPEAR rollouts stays flat at 0.718 across training, so the self-consistification hypothesis is not supported at this scale. Table 19 gives the full held-out curves.

A conditioning mismatch between the rollout and the learner, since fixed. An earlier version of this experiment generated rollouts on a fixed canvas of 256 masked positions, trimmed them at the first end-of-sequence token, and then rebuilt the canvas at the *trimmed* length for the policy-gradient pass. Under bidirectional attention the conditional at a position depends on how many masked positions follow it, so the learner was differentiating log-probabilities conditioned on a shorter canvas than the sampler drew from, and the "exact AR log-probabilities for free" claim held only up to that mismatch. We fixed it: the learner now builds its canvas at the sampler's generation length, in both the loss and the entropy-selection pass, a gradient test passes, and both 600-step runs were repeated from scratch under the corrected conditioning. Every RL number in this paper comes from those repeated runs. The remaining caveat is the one the design cannot fix, a single seed per condition.

The ρ-estimators. The GRPO importance ratio requires the AR-order log-probability at each scored position. Evaluating all positions is exact but expensive. The top-entropy estimator scores only the top fraction of positions by entropy, following the JustGRPO-Fast heuristic of Ni et al. (2026). It is biased because position selection depends on the value being estimated. The random estimator scores a random fraction with Horvitz-Thompson inverse-probability weights, which is unbiased. In the pre-fix ablation the unbiased estimator tracked the full-position estimator while the biased top-entropy estimator is the weakest of the three, though the differences are not significant at this scale.

G Implementation and Protocol

G.1 Implementation Details

Decoder definitions. We compare five decoders at $\tau = 0.6$. SPEAR samples π_{AR} at block length $L = 8$ unless stated; STEPWISE samples π_{AR} at one forward pass per token as the exact reference. CONFS is confidence-ordered decoding with S denoising steps, so CONF64 commits 4 tokens per step over a 256-token canvas and CONF256 commits one. FASTDLLM is the threshold-based parallel decoder of Wu et al. (2026), committing every masked position above confidence 0.9; we implement its

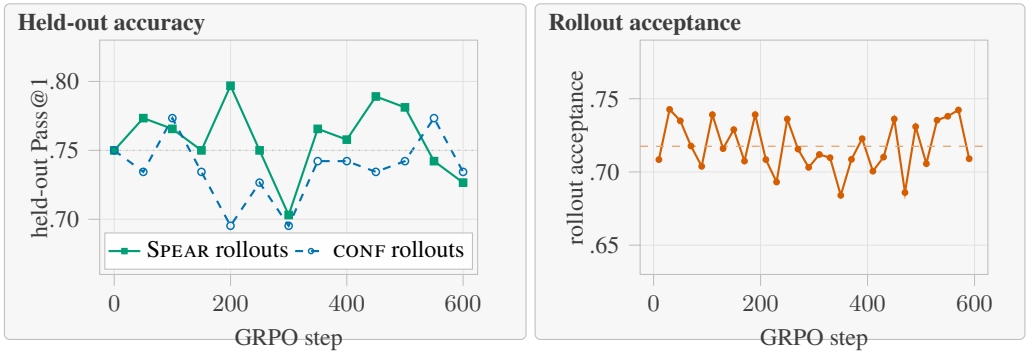


Figure 14: **GRPO with exact and with lossy rollouts,** 600 **steps each**, after the conditioning fix of Appendix F.1 so that the learner scores the same generation canvas the sampler produced. Left: held-out Pass@1 on 128 GSM8K test problems, evaluated every 50 steps. Both conditions wander in a band of 0.70 to 0.80 around the shared 0.75 starting point (dotted) with no trend, and the post-training difference of -0.008 is a fifth of the 0.039 binomial standard error of a single evaluation. The two curves come from one run each, so the 13 points per condition are correlated measurements of a single trajectory and we run no test on them. Right: acceptance of drafted tokens during SPEAR rollouts, in 20-step block means. It is flat at 0.718 (dashed), so training does not make the policy more self-consistent.

Table 20: Full experimental configuration (prob: problems; samp: samples per problem).

experiment	configuration	value
Pass@k (main, GSM8K)	model / τ / gen	LLaDA-8B / 0.6 / 256
	problems / samples	96 / 64 (48 stepwise)
	block length L	8
Pass@k (generalization)	LLaDA-1.5 GSM8K	32 prob / $\geq$ 32 samp
	LLaDA-8B MATH-500	500 prob / 24–40 samp
Pass@k (FASTDLLM)	threshold / problems / samples	0.9 / 96 / 64
Dream-7B	self-consistency / Pass@k	24 prompts / 32 prob $\times$ 32 samp
	verification	one row per forward (Appendix G.4)
Matched-budget control	STEPWISE vs CONF256	64 shared problems / 48–64 samp
Timing sweep	prompts / τ / L	16 / $\{0.6, 1.0\}$ / $\{4, 8, 16\}$
Adaptive truncation	threshold (nats)	$\{0.35, 0.80\}$, nominal L=16
Self-consistency	prompts / cells	24 / three model-dataset
Fidelity	metric	per-position KL$(q_j \Vert \pi_{\text{AR}})$
GRPO (main)	steps / group / lr	600 / 6 / 10^{-5}
	LoRA rank / clip	32 / 0.2
	held-out eval	128 GSM8K test, every 50
GRPO (ρ-ablation)	steps / conditions	120 / 5
Sublinear variant	clip B / oracle	$\{1, \ldots, 6\}$ / T=3, V=4
Bootstrap	resamples	2000 to 5000

threshold rule with exact attention and omit its approximate key-value cache, which is itself a source of distributional change, so our baseline is the more accurate version of that method (Appendix J.2). SSD is greedy self-speculative decoding (Gao et al., 2025), equal to SPEAR at $\tau = 0$. The confidence decoders are not handicapped: they get the full range of step budgets, and CONF256 spends more calls per sample than SPEAR. Hyperparameters, prompts and grading are in Appendix G.1.

Table 20 lists the full configuration for every experiment. All experiments run on a single H100 GPU in bfloat16 with generation length 256 unless noted.

All experiments run on a single H100 GPU in bfloat16 with generation length 256. Models are loaded with their released remote code. The mask token id is 126336 for LLaDA and 151666 for Dream. The sparse head is implemented by replacing the model output head with an identity map and applying the head weight only at queried positions, which is mathematically identical to the model head. Verification builds the L revealed-prefix rows as a single batched tensor and runs one forward pass. Wall-clock is measured with CUDA synchronization around the sampling loop, excluding

model load and tokenization. Pass@k uses the unbiased estimator of Chen et al. (2021). Bootstrap confidence intervals use 2000 to 5000 resamples over problems, and paired differences bootstrap the per-problem difference on shared problems. Sampling temperature is 0.6 for Pass@k following Ni et al. (2026) and 1.0 for the reinforcement learning rollouts.

G.2 Evaluation Cohorts

Methods differ in cost by almost a factor of four per sample, so they were not all run on the same number of problems, and additional runs were added during revision to support the matched-budget and threshold-decoder comparisons. Cohorts therefore differ between methods, and a comparison is only meaningful when it is paired on the problems both methods cover. Table 21 records what was actually run on GSM8K with LLaDA-8B, and Table 22 records the cohort behind every paired comparison we report. Earlier versions of this paper quoted the same sampler pair at two different cohort sizes, because comparisons computed before and after the additional runs were reported side by side. Every number in the current version is computed from the inventory below.

Table 21: Samples drawn on GSM8K with LLaDA-8B, by method.

method	problems	indices	samples per problem	total samples
SPEAR	96	0–95	64 on 64, 48 on 32	5632
CONF64	96	0–95	64 on 64, 48 on 32	5632
CONF256	96	0–95	64 on 64, 48 on 32	5632
CONF128	64	0–63	64	4096
FASTDLLM	96	0–95	64	6144
STEPWISE	64	0–87	48 on 56, 16 on 8	2816

Table 22: Shared problems behind each paired comparison on GSM8K.

comparison	shared problems	where reported	role
SPEAR VS STEPWISE	64	Table 13	exactness check
SPEAR VS CONF64	96	Table 25	matched-call quality
SPEAR VS FASTDLLM	96	Section 4.2	matched-call quality
SPEAR VS CONF256	96	Table 27	full-budget reference
STEPWISE VS CONF256	64	Table 26	order axis
CONF256 VS CONF64	96	Table 27	step-count axis

G.3 Evaluation Protocol

Prompting. We use the reasoning template of Zhao et al. (2025) (Figure 16): the problem statement followed by an instruction to reason step by step and to place the final answer within a boxed expression, wrapped in the model's chat template. Generation length is 256 tokens.

Answer extraction. We extract the predicted answer as the content of the last boxed expression in the generation. If no boxed expression is present, we fall back to the last number in the text. The extracted string is normalized by removing commas, currency symbols, and surrounding whitespace, and reduced to its leading signed decimal.

Correctness. GSM8K reference answers are integers, so we extract the boxed span if present and otherwise the last number, and compare numerically with tolerance 10^{-6}.

MATH-500 reference answers are LaTeX expressions such as $\left(3, \frac{\pi}{2}\right)$ and $3\sqrt{13}$, and grading them needs more care than we first gave it. Our initial grader matched the boxed span with the regular expression `\boxed\{([^{}]*)\}` and then reduced the result to its leading signed decimal. That is wrong twice over. The expression cannot match nested braces, so an answer like `\boxed{\frac{14}{3}}` never matched at all and the code silently fell back to the last number in the text; and reducing to a leading decimal collapses distinct answers, sending both $12 + 9i$ and $12 + 8i$ to 12, and destroys every symbolic answer. Using one broken grader for all methods does not make the comparison safe either, because decoders differ in how they format answers.

We therefore regenerated every MATH-500 sample and regraded it, in three steps, each of which we record because each changed numbers. First we replaced the extractor with a brace matcher, so that \boxed{\frac{14}{3}} is read rather than discarded. Second we removed `math_verify` from the comparison: its parser reads 5π as 5, $70\sqrt{2}$ as 70, $2 + 7i$ as 2 and the list $3, 5, 7$ as 3.5, so each compared equal to a different answer, and a guard that stripped LaTeX commands before looking for symbols could not detect it. Third we fixed the replacement: sympy's grammar needs braced arguments, so it read $11\sqrt{2}$ written without braces as 11, accepting 11 and rejecting the correct answer. The current grader braces bare arguments, parses both sides with sympy, compares comma-separated answers element-wise, and refuses any parse that drops a marker present in the source, falling back to string equality. On the original 96 problems, regenerating and regrading moves SPEAR Pass@1 from 0.296 under the first grader to 0.381 and CONF64 from 0.252 to 0.325. Those problems are easier than the benchmark average, so the full-benchmark values (0.333 and 0.286) are lower. Regrading every saved MATH-500 file from its stored predictions changes 586 of 26,096 labels (2.25%) against the labels each file carried before the regrade. All MATH-500 numbers in this paper come from the current grader, and `scripts/regrade_math.py` reproduces it.

Sampling. Pass@k uses temperature 0.6 following Ni et al. (2026). The reinforcement learning rollouts use temperature 1.0. The greedy self-speculative baseline uses temperature 0. Confidence decoding uses low-confidence remasking with a semi-autoregressive block of 32 and the stated number of steps.

The MATH-500 grader, and a defect we found late. Grading MATH-500 needs symbolic equivalence, and we delegated it to `math_verify`. That was a mistake. Its parser silently discards mathematical content: it reads 5π as 5, $70\sqrt{2}$ as 70, $2 + 7i$ as 2 and the list $3, 5, 7$ as the decimal 3.5. Every one of those then compares equal to a different answer, and the guard we had added against the complex-number case did not catch the others, because it stripped LaTeX commands before looking for symbolic content and so removed the very symbols it was meant to detect.

The repaired grader parses both sides with SYMPY's own LaTeX parser, compares comma-separated answers element-wise, and falls back to normalised string equality; a pair that neither matches as a string nor parses on both sides is graded incorrect, which is the conservative direction. It rejects all four cases above and still accepts the equivalences a grader should, such as $\frac{1}{2}$ against 0.5 and $\frac{1}{3}$ against its decimal. Regrading every saved MATH-500 sample changes 586 of 26,096 labels (2.25%), and the effect is not confined to one decoder. In the version of this paper that first applied the repair, when MATH-500 was represented by 96 problems, it moved one reported conclusion: the voting advantage over the threshold decoder, until then significant at $k = 16$, no longer excluded zero at any k. That statement is historical. The current pooled comparison includes all 500 MATH-500 problems, and against the threshold decoder it excludes zero at both ends: in that decoder's favour at $k = 1$ ($[-0.027, -0.003]$) and in SPEAR's favour at $k = 8$ ($[+0.001, +0.036]$) and $k = 16$ ($[+0.005, +0.046]$; Table 31). Every MATH-500 number in this paper, and every pooled result that includes the MATH-500 cell, is regenerated from the repaired grader; `scripts/regrade_math.py` reproduces the regrade and keeps a pre-regrade copy of each file.

G.4 DREAM-7B: TWO PRECONDITIONS FOR EXACT SELF-SPECULATIVE SAMPLING

Dream-7B is a masked diffusion model built on a Qwen backbone, so it is the architecture test that LLaDA-1.5 cannot provide. An earlier version of this paper reported that Dream produced incoherent output and failed the reveal-zero check with a maximum absolute log-probability difference of 0.18, and attributed this to the model's forward path. That attribution was wrong, and the diagnosis matters beyond Dream: it identifies two preconditions that any backbone must meet before the correction of Section 3 can be applied to it.

Precondition 1: know where the model puts its logits. A masked diffusion model may place the distribution for position k at hidden position k, as LLaDA does, or at position $k - 1$, inheriting the next-token convention of the autoregressive backbone it was adapted from. Dream is the second kind: its own generation code applies

$$\texttt{logits} \leftarrow \texttt{cat}(\texttt{logits}[:, :1],\ \texttt{logits}[:, :-1])$$

before reading masked positions. We had been reading position k. The consequence is instructive, because it is not a crash: every distribution is off by one position, which yields fluent-looking but wrong text and a reveal-zero difference that is large but finite. With the alignment corrected, Dream answers correctly. This is the whole of the incoherence we previously reported, and none of it was a property of Dream.

Precondition 2: the backbone must be batch-invariant. The remaining failure is real and belongs to the model. SPEAR verifies a block by running L revealed-prefix rows as one batched forward, which presumes that a row returns the same logits whether it is evaluated alone or alongside others. On LLaDA that holds bitwise. On Dream it does not: the row that reveals nothing, whose input is identical to the draft pass, differs from it by a total variation of 0.018 at batch size eight. The effect is deterministic, since the same input evaluated twice agrees to 0.000000, and it is not an artefact of kernel selection, since it persists on every scaled-dot-product backend we tested (0.0096 on the math backend, 0.012 on flash, 0.013 on memory-efficient). Batched verification on such a backbone would therefore target a distribution about 0.018 away from π_{AR} while appearing to work.

The fix is to verify one row per forward pass, which reproduces the single-row draft pass exactly (maximum absolute logit difference 0.00000000, total variation 0.00000000) and restores exactness. It costs L calls per block instead of one, so SPEAR on Dream commits 0.72 tokens per forward call at $L = 8$ against 3.45 on LLaDA. We accept that cost here because the point of this experiment is exactness across architectures, not speed, and we report the call count rather than hiding it.

What this says about the method. The reveal-zero check earns its place here. We described it in Section 4.1 as a weak test, because it cannot detect an error in the rows that reveal drafted tokens, and that remains true. But it is exactly the test that catches a non-batch-invariant backbone, which is a failure mode that produces plausible output and would otherwise go unnoticed. Any application of exact self-speculative sampling to a new dLLM should run it before trusting the sampler. With both preconditions met, Dream satisfies the bidirectionality the method assumes: revealing a token five positions to the right changes the distribution at the earlier position by a total variation of 0.83, so the conditionals SPEAR verifies are genuinely order-dependent rather than a causal model in disguise.

H ADDITIONAL PASS@k ANALYSIS

H.1 COMPLETE GSM8K PASS@k AND PAIRED TESTS

Table 23: Pass@k on GSM8K with LLaDA-8B.

method	n	fwd/sample	Pass@1	Pass@8	Pass@16	Pass@64
SPEAR (exact AR)	96	73	0.792	0.941	0.957	0.969
STEPWISE (AR)	32	256	0.788	0.944	0.968	n/a
FASTDLLM	96	71	0.800	0.908	0.928	0.969
CONF256	64	256	0.787	0.907	0.927	0.953
CONF128	64	128	0.751	0.939	0.961	0.984
CONF64	96	64	0.731	0.942	0.965	0.984

The rows of Table 23 cover different problem sets, given by n, because STEPWISE costs 256 forward passes per sample and was run on a subset. Its Pass@64 column uses the 64 problems that carry 64 samples in every method. Every comparison we draw is paired on shared problems: SPEAR against STEPWISE in Table 13, SPEAR against CONF64 in Table 25, and STEPWISE against CONF256 in Table 26.

Table 24 gives the complete Pass@k curves for the main GSM8K, LLaDA-8B experiment (96 problems, 64 samples) at every reported k. Table 25 gives the paired SPEAR minus CONF64 differences with bootstrap 95% CIs at every k. The single-shot advantage is significant at Pass@1 and Pass@2 ($+0.060$ and $+0.026$, both CIs excluding zero). The two methods are statistically tied from Pass@4 through Pass@16. The sign reverses without significance at the largest k, where ultra-parallel diversity helps the lossy baseline. This is the quantitative form of the two-axis reading in Section 4: exact AR-order sampling wins single-shot quality per forward pass and does not lose coverage.

Table 24: Complete Pass@k on GSM8K with LLaDA-8B, each row on its own n problems.

method	n	P@1	P@2	P@4	P@8	P@16	P@32	P@64
SPEAR	96	0.792	0.867	0.913	0.941	0.957	0.966	0.969
FASTDLLM	96	0.800	0.854	0.886	0.908	0.928	0.947	0.969
STEPWISE†	56	0.779	0.852	0.900	0.932	0.944	0.946	n/a
CONF256	64	0.787	0.843	0.880	0.907	0.927	0.942	0.953
CONF128	64	0.751	0.840	0.900	0.939	0.961	0.974	0.984
CONF64	96	0.731	0.841	0.903	0.942	0.965	0.976	0.984

Table 25: Paired SPEAR minus CONF64 Pass@k on GSM8K (star: interval excludes zero).

	P@1	P@2	P@4	P@8	P@16	P@32	P@64
Δ	$+0.060^{\star}$	$+0.026^{\star}$	$+0.010$	-0.000	-0.007	-0.011	-0.016
CI lo	$+0.035$	$+0.005$	-0.007	-0.017	-0.025	-0.032	-0.047
CI hi	$+0.086$	$+0.047$	$+0.028$	$+0.017$	$+0.009$	$+0.002$	$+0.000$

In Table 24, the † STEPWISE row carries 48 samples on 56 problems and 16 on a further 8, and is computed on the fixed 56-problem cohort that supports every k up to 32. Reporting each k on all problems eligible at that k, as an earlier version did, let the cohort change with k and produced a non-monotone curve (Pass@16 0.968 above Pass@32 0.958), which the estimator cannot produce on a fixed cohort.

H.2 MATCHED-BUDGET CONTROL: ORDER VERSUS STEP COUNT

The main Pass@k comparison sets SPEAR against CONF64, which differs from it both in sampling order and in how many tokens it commits per forward pass. To attribute the gain we hold the budget fixed and vary only the order. STEPWISE and CONF256 both spend 256 forward passes per sample, the first in AR order and the second in confidence order, so their difference isolates the order axis. We ran additional CONF256 and STEPWISE samples so that the two overlap on 64 problems, since the original runs shared only 8.

Table 26 reports the result, and Table 27 places it beside the step-count axis. Order does not significantly change single-shot accuracy: the paired Pass@1 difference is $+0.017$ with a CI that includes zero. It does change coverage, by about $+0.055$ at $k = 4$ to 16, with intervals excluding zero. Reducing the step count has the opposite profile: CONF256 leads CONF64 by $+0.061$ Pass@1 and trails it by about 0.04 at $k \geq 8$. The two axes are therefore close to orthogonal, and each governs the metric the other does not.

This is what makes SPEAR useful, and it is a narrower claim than the one the raw comparison invites. Against CONF256 we detect no Pass@1 difference (-0.001, CI $[-0.035, +0.033]$) at 73 calls against 256, and SPEAR leads at $k \geq 4$ ($+0.037$ to $+0.040$). The Pass@1 interval is wide, so this is an absence of detected difference rather than a demonstration of parity. Its $+0.060$ Pass@1 over CONF64 is the step-count effect, recovered at a low call count, rather than evidence that AR order is intrinsically better single-shot. The order axis is not thereby unimportant: it is significant on coverage here, and the answer distributions of the two orders differ decisively (excess TV $= +0.106$, $p < 0.001$, Table 14).

Table 26: Matched-budget control: STEPWISE against CONF256 at 256 passes (star: interval excludes zero).

	P@1	P@2	P@4	P@8	P@16	P@32
STEPWISE (AR order)	0.779	0.852	0.900	0.931	0.951	0.946
CONF256 (confidence order)	0.762	0.809	0.846	0.875	0.898	0.926
Δ (order effect)	$+0.017$	$+0.042$	$+0.053^{\star}$	$+0.056^{\star}$	$+0.053^{\star}$	$+0.020^{\star}$
CI lo	-0.025	-0.004	$+0.005$	$+0.012$	$+0.011$	$+0.000$
CI hi	$+0.061$	$+0.093$	$+0.107$	$+0.107$	$+0.107$	$+0.058$

In Table 26, Pass@32 uses the 56 problems with enough samples.

Table 27: Order and step count, each varied with the other fixed.

axis	contrast	n	Δ Pass@1	Δ Pass@8	Δ Pass@16
order (budget fixed)	STEPWISE − CONF256	64	$+0.017$	$+0.056^{\star}$	$+0.053^{\star}$
step count (order fixed)	CONF256 − CONF64	96	$+0.061^{\star}$	$-0.040^{\star}$	$-0.043^{\star}$
SPEAR against the full-budget decoder	SPEAR − CONF256	96	-0.001	$+0.040^{\star}$	$+0.036^{\star}$

H.3 PASS@k AT A MATCHED TIME BUDGET

Forward-pass counts are hardware independent, which is why the acceleration literature reports them, but they flatter SPEAR: one SPEAR call verifies an L-row batch while one confidence call is a single row. The budget a practitioner actually holds is time. We therefore give every method the same wall-clock and let cheaper decoders draw proportionally more samples, which is the allocation a user facing a fixed deadline would choose.

Table 28 gives the per-sample cost, measured for SPEAR and the confidence decoders inside a single job on one H100 at batch size one, so those four numbers are directly comparable; FASTDLLM was timed in a separate job on the same hardware. SPEAR costs 5.39s per 256-token sample, at parity with CONF256 and STEPWISE and about $3.9\times$ the cost of CONF64. Table 29 converts this into Pass@k at a common budget.

The result does not favour SPEAR. At every budget we can estimate, CONF64 reaches a higher Pass@k, because it buys roughly four samples for each one SPEAR draws. The paired difference is -0.144 (CI $[-0.186, -0.106]$) at 10s, -0.065 (CI $[-0.100, -0.036]$) at 20s, -0.038 (CI $[-0.069, -0.014]$) at 40s and -0.036 (CI $[-0.076, -0.005]$) at 80s, every interval excluding zero. Against FASTDLLM the difference is within noise throughout (-0.026, CI $[-0.055, +0.002]$ at 20s and -0.001, CI $[-0.021, +0.026]$ at 80s). We state the consequence plainly: for test-time scaling, where the objective is Pass@k per second and any sampler will do, a lossy parallel decoder is the better choice, and our own quality-per-forward-pass result does not transfer to that setting.

Two caveats bound this in the other direction. All timings are batch size one. Outer batching reduces the per-sample cost of single-row decoders more than it reduces SPEAR's: from one to four samples per pass, STEPWISE throughput rises by $1.40\times$ and SPEAR throughput by $1.12\times$ (Appendix H.7), because SPEAR's verification rows already use the capacity that batching would fill. We did not time the lossy decoders under outer batching, but the same argument says batching widens the gap rather than narrowing it. And a key-value-cached verification, which we do not have, would cut SPEAR's per-sample cost directly. The comparison here is of the systems as we built them.

Table 28: Per-sample wall-clock on GSM8K at batch size one.

	SPEAR	STEPWISE	CONF256	FASTDLLM	CONF64
seconds per sample	5.39	5.53	5.56	1.51	1.39
calls per sample	73	256	256	71	64
row-forwards per sample	≈ 330	256	256	71	64

Table 29: Pass@k at a matched wall-clock budget on GSM8K.

budget	10s	20s	40s	80s
SPEAR (k)	1	3	7	14
SPEAR	0.792	0.896	0.937	0.955
FASTDLLM (k)	6	13	26	52
FASTDLLM	0.899	0.922	0.941	0.956
CONF64 (k)	7	14	28	$57^{\dagger}$
CONF64	**0.935**	**0.961**	**0.975**	**0.984**

In Table 29, each method draws $k = \lfloor T/t \rfloor$ samples at budget T and its own per-sample cost t, so cheaper decoders receive more samples. Pass@k needs at least k drawn samples per problem, so every entry uses all 96 problems except the one marked †, which uses the 64 problems with 64 samples, and budgets above 80 s, which would need 105 to 228 samples per problem, are omitted.

H.4 Self-Consistency Voting and Calibration

Section 4.3 argues that voting and Pass@k read different aspects of the sampling distribution. Here we give the measurements. For each problem we draw k of the available samples without replacement, take the modal extracted answer, and score it; repeating this 400 times per problem gives the majority-vote accuracy, and we pair over problems as elsewhere. Vote share is the fraction of the k samples that agree with the modal answer, which is the natural confidence signal a practitioner would read off. The analysis script is `scripts/vote_analysis.py` and it regenerates every number below.

Table 30: Majority-vote accuracy on GSM8K with LLaDA-8B.

k	1	4	8	16	32
SPEAR	0.791	**0.845**	**0.860**	**0.871**	**0.877**
FASTDLLM	**0.802**	0.827	0.840	0.848	0.854
CONF64	0.732	0.819	0.849	0.861	0.864
CONF256	0.786	0.811	0.819	0.825	0.828

Table 30 uses 96 problems, and 64 for CONF256. CONF256 is included to show that the effect is not simply a step-count effect: it spends four times the calls of CONF64 and votes worse, because its samples are less diverse.

Table 31: Paired majority-vote differences pooled over the four math cells (star: interval excludes zero).

comparison	$k=1$	$k=2$	$k=4$	$k=8$	$k=16$
SPEAR − CONF64	$+0.060^\star$	$+0.059^\star$	$+0.043^\star$	$+0.034^\star$	$+0.032^\star$
	$[+0.049, +0.071]$	$[+0.048, +0.070]$	$[+0.031, +0.056]$	$[+0.021, +0.048]$	$[+0.016, +0.049]$
SPEAR − FASTDLLM	-0.015	-0.016	$+0.005$	$+0.018^\star$	$+0.025^\star$
	$[-0.027, -0.003]$	$[-0.027, -0.004]$	$[-0.009, +0.020]$	$[+0.001, +0.036]$	$[+0.005, +0.046]$

The four cells of Table 31 supply 660 model-item combinations from 596 distinct items: 96 GSM8K items, of which 32 are scored under all three models, and all 500 MATH-500 items. Intervals are 95% cluster bootstraps that resample items, not rows.

Inputs. The MATH-500 cell uses only the regraded runs. The earlier MATH-500 sample files were produced with the superseded extractor of Appendix G.3, which could not match nested braces and reduced answers to a leading decimal; re-applying the current comparison to their stored predictions flips 3.1%, 0.1% and 6.1% of labels, and the extraction itself cannot be repaired from what those files saved. They also carry no threshold-decoder samples, so pooling them would have given SPEAR and CONF64 extra and differently graded draws that FASTDLLM never received. GSM8K grading was never changed, and re-applying the current comparison to every GSM8K file flips no labels, so the three GSM8K cells pool their July and September runs directly.

Dependence between comparisons. Two dependencies matter here. The pooled unit is a model-item combination, and the three GSM8K cells share their benchmark items: the 32 GSM8K items used for LLaDA-1.5 and Dream-7B are a subset of the 96 used for LLaDA-8B, so the 660 pooled rows come from 596 distinct items. Resampling rows would treat one item scored under three models as three independent observations, so every pooled interval we report is a cluster bootstrap over items. On the current cohort this widens the intervals by at most 8% and changes the sign of no interval bound, so no conclusion depends on the choice: most of the pool is MATH-500, whose 500 items are each scored once, and only the 32 shared GSM8K items appear under more than one model. On the smaller cohort of an earlier version, where those shared items were a larger fraction, clustering did cost one comparison its significance, which is why we keep it. Separately, the five k values are computed from the same draws, so the twenty cell-by-k comparisons are not twenty independent tests. We therefore report the consistency of the sign descriptively and attach no sign-test p-value to it. At a fixed k the cells are the independent unit, and a sign test over four cells, all positive, gives $p = 0.125$, which is the smallest value four cells can produce. An earlier version of this appendix reported $p = 2 \times 10^{-6}$ from treating the twenty comparisons as independent; that test was not valid. The effect is largest where the baseline is weakest, reaching $+0.250$ at $k = 1$ on Dream-7B,

Table 32: Diversity and quality on GSM8K with LLaDA-8B.

method	forward passes	Pass@1	Pass@64
SPEAR (exact)	73	**0.792**	0.969
CONF256	256	0.787	0.953
CONF128	128	0.751	0.984
CONF64	64	0.731	0.984

whose arbitrary-order decoding at 64 steps is much worse than LLaDA's. Per-cell numbers are in `paper/figdata/vote_percell.csv`.

Calibration. The vote share should predict correctness, and it can only be compared on a cohort every decoder covers. Over the 660 model-item combinations that SPEAR, CONF64 and FASTDLLM all cover, the expected calibration error is 0.049, 0.051 and 0.143 respectively. SPEAR and CONF64 are indistinguishable at that precision, and we attach no interval that would separate them; both are well calibrated relative to FASTDLLM, whose vote share overstates its accuracy. That ranking does not extend to STEPWISE, which was run on a subset: the 80 combinations all four share are GSM8K only, and on them the order changes to 0.077 for FASTDLLM, 0.093 for SPEAR, 0.108 for STEPWISE and 0.163 for CONF64. We therefore do not claim SPEAR is the best-calibrated of the four decoders, and an earlier version of this appendix did so by comparing a 256-problem estimate against an 80-problem one. The two exact samplers land far apart on both cohorts, which says calibration is not a property of exactness. By the competing measure of ranking ability, AUROC does not favour SPEAR on either cohort (0.873 against 0.876 for CONF64 on the larger one). Both cohorts are saved, in `vote_calibration.csv` and `vote_calibration_common4.csv`.

At a matched time budget. Section H.3 showed exact sampling losing Pass@k once cheaper decoders are given proportionally more samples. Voting is not an exception. We fix budgets in GPU-seconds per problem before running the analysis, and convert each budget into a sample count using each cell's own measured per-sample cost, because the cost ratio is not common to the cells: a SPEAR sample costs 5.41 to 6.65s across the four cells against 1.26 to 1.40s for CONF64, giving per-cell ratios of 3.87, 4.00, 4.42 and 5.28. Earlier versions of this comparison matched every cell at one pooled ratio and then capped two cells that ran out of drawn baseline samples, which understated the baselines. We have since drawn those samples, so nothing below is censored.

Pooled and clustered over benchmark items, SPEAR is behind CONF64 by -0.025 ($[-0.037, -0.014]$) at 10 GPU-seconds, by -0.020 ($[-0.034, -0.007]$) at 20, and by $+0.003$ ($[-0.013, +0.019]$) at 40; against FASTDLLM the same budgets give -0.040 ($[-0.055, -0.026]$), -0.024 ($[-0.040, -0.007]$) and $+0.001$ ($[-0.018, +0.021]$). All four comparisons at 10 and 20 seconds exclude zero, and all four favour the baseline. At 40 seconds the intervals span zero, and we do not read that as parity: they are wide enough to contain differences that would change the practical conclusion in either direction. The defensible summary is that the per-sample advantage of Table 3 does not convert into an advantage per unit of compute at any budget we can measure, and that the gap narrows as the budget grows.

H.5 DIVERSITY VERSUS QUALITY

The confidence baselines expose a second axis, orthogonal to sampling order: the number of denoising steps. Table 32 contrasts single-shot quality (Pass@1) with high-k coverage (Pass@64) as the step count varies. Fewer steps commit more tokens per forward pass, which raises sample diversity and hence Pass@64 ($0.953 \to 0.984$ as steps fall from 256 to 64) but lowers Pass@1 ($0.787 \to 0.731$). SPEAR attains the best Pass@1 of any method at 73 forward passes while remaining competitive on Pass@64. The matched-budget control of Section 4.2 shows which axis does what, and it is the opposite of what this table alone suggests. At an equal budget of 256 forward passes, order has no significant effect on Pass@1 ($+0.017$, CI $[-0.025, +0.061]$) and a significant effect on coverage ($+0.053$ to $+0.056$ at $k = 4$ to 16). The step count is what moves single-shot quality ($+0.061$ Pass@1 from CONF64 to CONF256) and it moves high-k coverage the other way. A natural extension is to combine them, applying a diversity-inducing schedule within exact AR-order sampling, which we leave to future work.

H.6 The Threshold Baseline Against Its Official Implementation

Our threshold decoder reimplements the Fast-dLLM confidence rule and runs without a KV cache, which invites the objection that the baseline is handicapped. We therefore ran the authors' own code, vendored unmodified at `third_party/fastdllm_v1`, on the same 96 GSM8K problems with the settings their evaluation script uses for the dual-cache parallel configuration (256 steps, block length 32, confidence 0.9), and also with caching disabled as an ablation.

Table 33: Our threshold decoder against the official implementation.

implementation	Pass@1	Pass@8	s / sample
official, dual cache + parallel	0.790	0.916	2.31
official, no cache	0.796	0.906	1.93
ours (uncached), used throughout	**0.800**	0.908	**1.51**
SPEAR (exact), for reference	0.792	**0.941**	5.41

The baseline is not handicapped: our version is the fastest of the three and its Pass@1 is the highest, so every comparison in the paper is against the strongest threshold decoder we could produce. Caching does not help here, and is in fact slower. That is not a contradiction of the Fast-dLLM results, which are reported on five-shot prompts: a prefix cache amortizes over the prompt, and our prompts are zero-shot and short, so there is little prefix to reuse while the per-block cache bookkeeping still costs. The honest reading is that the cached decoder's advantage depends on the prompt regime, and in ours it does not appear.

Against the official cached implementation SPEAR is level at Pass@1 (0.792 against 0.790) and ahead at Pass@8 (0.941 against 0.916), at 2.3× the wall-clock per sample. That is the same trade the rest of the paper reports, now measured against the baseline's own code rather than ours.

H.7 Scaling in Generation Length, and What Sets the Speedup

Both SPEAR and STEPWISE target π_{AR}, so comparing their cost is a like-for-like question about two exact decoders. We run both on the same eight GSM8K prompts at generation lengths 256, 512 and 1024, discard the first sample as warm-up and time the remaining 7, giving both the sparse output head. SPEAR uses entropy-adaptive truncation at 0.35 nats with nominal $L = 16$, the configuration of Table 16. All timings are on one H100.

Table 34: Cost of two exact decoders against generation length.

	256 tokens		512 tokens		1024 tokens	
	SPEAR	STEPWISE	SPEAR	STEPWISE	SPEAR	STEPWISE
wall-clock (s)	4.71	5.54	15.26	16.36	55.92	56.17
s.d. (s)	0.36	0.14	0.94	0.50	1.54	1.29
tokens/s	54.6	46.3	33.7	31.3	18.3	18.2
peak memory (GiB)	15.8	15.0	16.4	15.1	17.4	15.1
speedup	1.18×		1.07×		1.00×	

A cost model, measured rather than assumed. Per committed token, STEPWISE costs one forward pass c. SPEAR costs one draft pass plus one verification batch of $\bar{L}$ rows per block, $c + v$, and a block commits t tokens, so its speedup is $t\,c/(c + v)$. Writing $v = \alpha\bar{L}\,c$, the factor α is the batching efficiency: the cost of an $\bar{L}$-row batch relative to $\bar{L}$ separate single-row passes, below 1 while the forward is partly memory-bound and approaching 1 as it becomes compute-bound. Two of these quantities the scaling table cannot separate, so we measured them directly: c and v by timing forward passes at batch sizes 1 to 16 on each canvas (12 repetitions after warm-up, median), and t and $\bar{L}$ by running SPEAR on six prompts per length.

The model reproduces the measured decay to within 0.05, erring slightly high because it leaves out sampling and host overhead. It also corrects the explanation we gave in an earlier version, which attributed the decay to the growing per-pass cost swamping the saving of a shared draft pass. The

Table 35: The speedup decomposed on LLaDA-8B.

length	c (ms)	$\bar{L}$	t	v/c at $\bar{L}$	α	predicted	measured	bound
256	20.4	4.85	5.73	3.67	0.76	1.23	1.18	1.25
512	30.6	6.79	7.57	5.95	0.88	1.09	1.07	1.12
1024	53.6	9.63	10.52	9.35	0.97	1.02	1.00	1.03

draft pass is not a saving; it is the overhead. The entire speedup is the batching discount, $\alpha < 1$. Two things change with length and they pull in opposite directions. Blocks get longer, $\bar{L}$ rising from 4.85 to 9.63 as later text holds fewer high-entropy positions, which by itself would help. But α rises from 0.76 to 0.97 as the canvas makes the forward compute-bound, and that dominates. Because a block commits at most $\bar{L}+1$ tokens, the speedup can never exceed $(\bar{L}+1)/(1+\alpha\bar{L})$, which tends to 1 as $\alpha \to 1$ whatever the acceptance rate; at 1024 tokens even perfect acceptance would give only 1.03.

Outer batching. The discount comes from spare capacity in a single-row forward pass, and serving several samples of one problem together uses the same capacity. We therefore decoded b samples of one prompt together with both samplers. Stepwise decoding batches trivially. SPEAR runs one draft pass over the b canvases and one verification pass over the revealed-prefix rows of all b sequences, and a finished sequence leaves the batch. The accept/reject rule of each sequence is unchanged, so on a batch-invariant backbone every output is still an exact sample. On the toy oracle of Appendix B.2 the batched sampler is within total variation 0.015 of the exact joint without entropy truncation and 0.018 with it, inside the Monte Carlo noise scale of 0.030, while the same sampler with verification removed is at 0.33 (`tests/test_batched.py`). Table 36 reports LLaDA-8B with 3 prompts per cell (2 for $b \geq 16$). The advantage over STEPWISE exists only at $b = 1$: 1.21× at 256 tokens and 1.09× at 512, against 0.97× and 0.97× at $b = 4$. The forward-cost curve says why. At 256 tokens a call over 4 rows costs 2.97 single-row calls and a call over 64 rows costs 48.45, so the cost per row stops falling after about four rows. Once b sequences fill the forward pass, a verification call over $b\bar{L}$ rows costs close to $\bar{L}$ draft passes, and the effective α, measured from the draft and verification phase times, rises from 0.72 at $b = 1$ to 0.92 at $b = 4$ and 0.99 at $b = 32$. Batching also helps the baseline more. From $b = 1$ to $b = 4$, STEPWISE throughput rises by 1.40× and SPEAR throughput by 1.12×. Any single-row decoder should gain in the same way, so we expect outer batching to widen the equal-time gap to the lossy decoders of Section 4.3, not close it. We did not time those decoders under outer batching.

Table 36: Outer batching: b samples of one prompt decoded together.

	256 tokens				512 tokens			
b	speedup	α	tok/s STEPWISE	tok/s SPEAR	speedup	α	tok/s STEPWISE	tok/s SPEAR
1	1.21	0.72	48	58	1.09	0.80	33	36
4	0.97	0.92	68	65	0.97	0.96	38	36
16	0.99	0.96	66	65	0.99	0.98	37	37
32	0.97	0.99	67	65	n/a	n/a	n/a	n/a

A second architecture. We repeated the decomposition on Dream-7B, whose Qwen-based forward pass has the same shape as LLaDA's (Table 37). Its batching efficiency at the realized verification batch rises from 0.65 at 256 tokens to 0.98 at 1024, against 0.76 to 0.97 on LLaDA-8B. Dream cannot use the discount, because its backbone is not batch-invariant and exactness forces one row per call (Appendix G.4). With $v = \bar{L}c$ the model predicts no speedup at any length, and the measured speedups over STEPWISE are 1.00, 1.01 and 1.01 (5 timed samples per length), within 0.03 of the prediction. Had the rows been batchable, the model predicts 1.36, 1.16 and 1.02, the same decay as on LLaDA-8B. The decomposition therefore predicts the measured speedup on a second architecture and in a second regime.

What this does and does not establish. The bound is general for a class of decoders: any exact sampler that verifies each drafted position with its own row and commits at most one token beyond the verified block. On a backbone without cheap prefix-conditional scoring that is forced (Section 3.2), which is why we think the result is the most transferable one in the paper. Where the bound bites is not general. α is a property of the hardware, the kernels and the canvas, and we measured it on

Table 37: The speedup decomposed on Dream-7B.

length	c (ms)	$\bar{L}$	t	α	one row per call	measured	if batched
256	20.0	4.22	5.09	0.65	0.97	1.00	1.36
512	27.0	6.56	7.51	0.83	0.99	1.01	1.16
1024	43.7	9.53	10.47	0.98	0.99	1.01	1.02

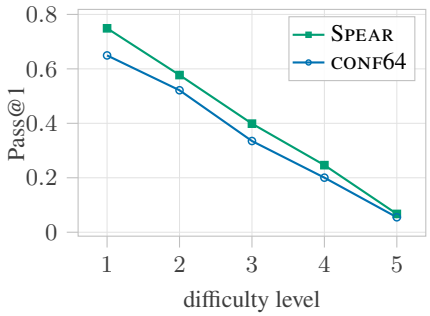


Figure 15: MATH-500 Pass@1 by difficulty level.

Table 38: MATH-500 Pass@1 by difficulty level, with paired differences.

level	problems	SPEAR	CONF64	Δ	95% CI
1	43	0.749	0.649	$+0.100$	$[+0.061, +0.143]$
2	90	0.577	0.521	$+0.056$	$[+0.029, +0.083]$
3	105	0.399	0.335	$+0.064$	$[+0.034, +0.095]$
4	128	0.246	0.200	$+0.046$	$[+0.026, +0.067]$
5	134	0.067	0.055	$+0.012$	$[+0.002, +0.023]$

one H100, for LLaDA-8B and Dream-7B; a device or kernel that stays memory-bound over longer canvases would keep α lower and the speedup alive for longer. Two further limits are ours: the timings rest on 7 prompts per cell, six per length for the decomposition and two or three per outer-batch cell, and our outer batching does not refill a finished sequence's slot, as a continuous-batching server would. The bound does not apply to verification schemes whose cost is sublinear in the number of verified positions, such as any-subset models (Guo & Ermon, 2026) or the Bernoulli-factory variant of Section 3.3, which attempts exactly that and fails for a different reason. Peak memory moves the other way and modestly, because the verification batch holds $\bar{L}$ rows of activations.

H.8 Robustness Across Difficulty

MATH-500 is annotated with difficulty levels one (easiest) to five (hardest). Table 38 and Figure 15 break the SPEAR versus CONF64 Pass@1 comparison down by level. The advantage of exact AR-order sampling is present at every level, from $+0.100$ on level one to $+0.012$ on level five, and is larger where absolute accuracy is higher. The effect is therefore not concentrated in one difficulty band. It is a consistent shift toward the exact distribution across the difficulty range. This is what the self-consistency mechanism predicts, since the exact conditionals differ most from the one-shot marginals at forking tokens, which occur throughout.

H.9 Solution-Space Overlap

The Pass@k advantage of SPEAR could arise either because it solves problems the lossy baseline never solves, or because it solves the same problems at a higher single-shot rate. Table 39 distinguishes these by reporting, on shared problems, how many are solved (at least one correct sample) by each method. Each pair is compared at the same number of samples per problem, the smaller of the two, since a larger sample count can only enlarge a solved set. On GSM8K the solved sets coincide: the symmetric difference is 1 problem against CONF256 and 1 against CONF64. On the harder MATH-500, at 24 samples per problem on all 500 problems, the two cases differ. Against CONF64

Table 39: Solved-set overlap on shared problems at equal samples per problem.

dataset	baseline	shared	both	SPEAR-only	base-only	neither
GSM8K	CONF256	64	61	1	0	2
GSM8K	CONF64	96	93	0	1	2
MATH-500	CONF64	500	310	29	27	134
MATH-500	FASTDLLM	500	289	50	21	140

the sets differ on 56 problems, split almost evenly (29 solved only by SPEAR, 27 only by CONF64), so the advantage over CONF64 is single-shot rate, not reach. Against the threshold decoder the split is uneven: 50 problems are solved only by SPEAR and 21 only by FASTDLLM. This is the coverage counterpart of the Pass@k pattern in Table 12, where FASTDLLM leads at Pass@1 and falls behind as k grows. It is consistent with a decoder that commits only confident tokens producing less varied samples, though we have not measured sample diversity directly. We therefore do not claim that exact sampling reaches new problems in general. Against a diverse lossy decoder it wins on quality per sample, and against a low-diversity one it also wins on reach.

I CODE GENERATION

Protocol. We run LLaDA-8B-Instruct on all 164 HumanEval problems (Chen et al., 2021) and all 257 problems of the sanitized MBPP test split (Austin et al., 2021), with the math settings: $\tau = 0.6$, $L = 8$ and generation length 256. The HumanEval prompt (Figure 17) gives the function stub and asks for the complete function in one Python code block. The MBPP prompt gives the task and its first test assertion, which fixes the function name. We extract the first fenced block that defines a function, falling back to the first block and then to the raw text, and grade by running every test of the problem in a fresh Python process with a 10-second and 2 GiB limit. Under this grader the reference solutions pass all 164 and all 257 problems, and the same extraction applies to every decoder. SPEAR draws 10 samples per problem; CONF64 and FASTDLLM draw 36, enough for the 40-second budget at their cost; STEPWISE draws 12 on the first 24 problems of each dataset. All decoders of a shard run in one job, so their per-sample costs are measured on the same node.

Table 40: Code generation with LLaDA-8B (parses: share of programs Python can parse).

dataset	method	problems	s/sample	fwd/sample	parses (%)	P@1	P@2	P@4	P@8
HumanEval	SPEAR	164	6.42	68	95.2	0.345	0.432	0.520	0.612
	CONF64	164	1.64	64	47.5	0.157	0.220	0.285	0.347
	FASTDLLM	164	1.12	44	98.1	0.393	0.440	0.484	0.523
	STEPWISE	24	5.85	256	98.6	0.562	0.645	0.711	0.745
MBPP	SPEAR	257	5.29	73	98.9	0.459	0.534	0.595	0.654
	CONF64	257	1.40	64	42.1	0.164	0.251	0.347	0.439
	FASTDLLM	257	1.62	74	99.3	0.482	0.519	0.550	0.580
	STEPWISE	24	5.47	256	98.3	0.483	0.544	0.583	0.610

Exactness and self-consistency carry over. Paired on shared problems, SPEAR minus STEPWISE Pass@1 is $+0.004$ ($[-0.060, +0.074]$) on HumanEval and $+0.013$ ($[-0.038, +0.062]$) on MBPP. Mean acceptance is 0.875 and 0.823 ($\pm\, 0.004$ over problems), a little below the math cells.

Where the lossy decoders fail. CONF64 commits four tokens per pass, and only 47.5% and 42.1% of its programs parse. The failures are the token-level artifacts of Appendix K.1, now fatal: duplicated or dropped tokens at syntax positions, such as `inside = = False` or `paren_groups = [] = []`. The threshold decoder commits only positions whose confidence exceeds 0.9, and 98.1% and 99.3% of its programs parse. At a matched forward-pass budget SPEAR leads CONF64 by $+0.187$ Pass@1 on HumanEval ($[+0.146, +0.230]$) and $+0.295$ on MBPP ($[+0.257, +0.335]$). Against the threshold decoder the math crossover reappears: behind at $k = 1$ (-0.049, $[-0.083, -0.016]$; -0.022, $[-0.045, +0.000]$) and ahead at $k = 8$ ($+0.090$, $[+0.039, +0.140]$; $+0.074$, $[+0.040, +0.110]$).

Matched GPU time. Table 41 applies the budgets of Section 4.3, converting each at the decoder's measured cost on that dataset; no budget needed more samples than were drawn. Against CONF64, SPEAR leads at every budget on MBPP and at 20 and 40 seconds on HumanEval, and each of those intervals excludes zero. At 10 seconds on HumanEval, one SPEAR sample against six, the interval spans zero. Against the threshold decoder, SPEAR is behind at 10 seconds on both datasets and at 20 on HumanEval, and level at 20 seconds on MBPP and at 40 on both. This is the only setting in the paper where exact sampling is ahead at equal compute, and the reason is specific. A fixed-step decoder that breaks the output wastes the extra samples its lower cost buys; a decoder that commits only confident tokens does not.

Table 41: Pass@k at matched GPU time on code (k: samples per budget, SPEAR/baseline).

	HumanEval				MBPP			
budget	k	vs CONF64	k	vs FASTDLLM	k	vs CONF64	k	vs FASTDLLM
10 s	1/6	$+0.023$ $[-0.015, +0.060]$	1/8	-0.178 $[-0.226, -0.130]$	1/7	$+0.037$ $[+0.007, +0.069]$	1/6	-0.109 $[-0.138, -0.080]$
20 s	3/12	$+0.102$ $[+0.058, +0.146]$	3/17	-0.074 $[-0.121, -0.029]$	3/14	$+0.067$ $[+0.035, +0.101]$	3/12	-0.026 $[-0.053, +0.001]$
40 s	6/24	$+0.142$ $[+0.089, +0.196]$	6/35	-0.010 $[-0.060, +0.040]$	7/28	$+0.077$ $[+0.038, +0.119]$	7/24	$+0.018$ $[-0.013, +0.051]$

J Comparison and Extensions

J.1 The Confidence Decoding Baseline in Detail

The confidence baseline is the standard lossy parallel decoder for masked diffusion language models, following Nie et al. (2025). The canvas is split into semi-autoregressive blocks of size 32. Within the current block, the model runs S/n_{blocks} steps. At each step it reads the whole canvas and predicts a distribution at every masked position. It samples a token at each masked position at temperature τ, and scores each by its probability under the temperature-one distribution. It then commits the highest-confidence positions and remasks the rest. The number of steps S controls parallelism: $S = N$ commits one token per step, the most sequential setting, and smaller S commits several tokens per step. This is a valid sampler of some distribution over sequences, but that distribution is not π_{AR}. Committing several tokens per step from the per-position marginals, without conditioning each on the others, is the uncorrected parallel commitment whose distributional error we measure in Section C.4. SPEAR uses the same per-position marginals as its proposal but corrects them by accept and reject, which is the difference between an approximate and an exact sampler.

J.2 The Threshold Parallel Decoder

The second lossy baseline is the parallel decoding rule of Wu et al. (2026). At each step it reads the canvas, samples a token at every masked position of the current semi-autoregressive block at temperature τ, scores each by its probability under the temperature-one distribution, and commits every position whose score exceeds a threshold, instead of a fixed number per step. At least one position is committed per step so the loop terminates. At a threshold of 0.9 this spends 71 forward passes per 256-token sample on GSM8K, between CONF64 at 64 and SPEAR at 73, which is what makes it the right budget-matched comparison. Fast-dLLM also caches keys and values across steps. That cache is approximate: it reuses attention state computed under a different mask, so it perturbs the distribution as well as saving time, and Wu et al. (2026) present it as an approximation rather than an identity. We therefore do not reimplement it. Our baseline runs the threshold rule with exact attention, which makes it a stronger and more accurate opponent than the published system, and it means our forward-pass counts are not comparable to the wall-clock numbers reported for cached Fast-dLLM.

The decoder is lossy in the same sense as CONFS: it commits several tokens per forward pass from the per-position marginals without conditioning each on the others. Thresholding makes the lossiness adaptive rather than removing it, since positions are committed in parallel exactly when the model is confident, which is where the one-shot marginal and the exact conditional agree most closely. This is why it recovers most of the single-shot accuracy of a full-budget decoder, reaching Pass@1 0.800 against 0.793 for CONF256, and why it still loses coverage relative to SPEAR at $k \geq 8$: the positions it commits in parallel are the safe ones, so the sampled trajectories stay close together.

Table 42: Decoding methods for masked diffusion language models.

method	exact	sampling	calls for N tokens	rows per call	AR log-prob
stepwise AR	yes	yes	N	1	yes
confidence / Fast-dLLM	no	yes	$\ll N$	1	no
greedy self-speculative (SSD)	at τ=0 only	no	$< N$	1 and L	no
SPEAR (this work)	yes	yes	$< N$	1 and L	yes

J.3 Comparison to Prior Decoding Methods

Table 42 places SPEAR among decoding methods for masked diffusion language models. It compares them on the properties that matter for the AR-order distribution: whether the output distribution is exact, whether it supports temperature sampling (and hence Pass@k diversity), the forward-pass cost, and whether it supplies exact log-probabilities for reinforcement learning. Stepwise decoding is exact and temperature-aware but pays one forward pass per token. Confidence and entropy-threshold parallel decoding (the Fast-dLLM family) are fast but lossy. Greedy self-speculative decoding is exact only at temperature zero and deterministic. SPEAR is the method that is exact at temperature, sub-linear in forward passes, and returns exact AR log-probabilities.

In Table 42, "exact" means the output distribution equals the AR-order policy, "sampling" means temperature above zero with Pass@k diversity, and "AR log-prob" means exact AR-order log-probabilities are available for reinforcement learning. The calls and rows-per-call columns must be read together: a SPEAR verification call is an L-row batch, so its lower call count is not a proportional reduction in compute (Appendix B.4). Batching also presumes a batch-invariant backbone; where that fails the rows run singly, at L calls per block (Appendix G.4).

J.4 General Path-Integral Estimator

The single-position estimator of Section B.2 is the simplest unbiased estimator of the block log-ratio. More generally, following Chen et al. (2026), one can estimate $W = \sum_k w_k$ by any importance distribution over positions. For a distribution P over $\{1, \dots, L-1\}$, the estimator $\hat{W} = w_m / P(m)$ with $m \sim P$ is unbiased, since $\mathbb{E}[\hat{W}] = \sum_m P(m)\, w_m / P(m) = \sum_m w_m = W$. The uniform choice $P(m) = 1/(L-1)$ recovers $(L-1)w_m$.

The discrete path integral. The continuous method of Chen et al. (2026) estimates a log-density difference by integrating gradient queries along a path from a reference point to the target. Our discrete analog replaces the path by the left-to-right unmasking sequence. Writing the block log-ratio as a telescoping sum along this path,

$$W = \log \frac{\pi_{\text{AR}}(\tilde{o}_{p:p+L})}{q(\tilde{o}_{p:p+L})} = \sum_{j=0}^{L-1} \Big[\log \pi_{\text{AR}}(\tilde{o}_{p+j} \mid \tilde{o}_{p:p+j}) - \log q_{p+j}(\tilde{o}_{p+j}) \Big] = \sum_{j=1}^{L-1} w_j, \qquad (8)$$

where the $j = 0$ term vanishes because $q_p = \pi_{\text{AR}}(\cdot \mid o_{<p})$ at the block start. Each term w_j is a single position query: the conditional $\pi_{\text{AR}}(\cdot \mid \tilde{o}_{p:p+j})$ is read from the verification row that reveals $\tilde{o}_{p:p+j}$. Subsampling the path at a random position, rather than evaluating all $L-1$ terms, is what makes verification sublinear. This is the discrete counterpart of subsampling the continuous path integral.

Variance and the optimal importance distribution. The second moment of the estimator is $\mathbb{E}[\hat{W}^2] = \sum_m w_m^2 / P(m)$. Minimizing this over P subject to $\sum_m P(m) = 1$ gives, by Cauchy-Schwarz, the optimal choice $P^\star(m) \propto |w_m|$, with minimized second moment $(\sum_m |w_m|)^2$. Compared to the uniform choice, whose second moment is $(L-1)\sum_m w_m^2$, the variance reduction is the ratio $(L-1)\sum_m w_m^2 / (\sum_m |w_m|)^2$, which is largest when the $|w_m|$ are unequal. Since the per-position log-ratio grows with block depth (Table 9, Figure 9), the $|w_m|$ are indeed unequal, so a depth-weighted P reduces variance. Lower estimator variance reduces the Poisson intensity needed for a target clip error and hence the expected number of position evaluations per accepted block. We did not need this refinement because batched verification is faster than the sublinear variant on current hardware, but it would matter in the memory-bound regime where the sublinear variant is preferred.

K Reproducibility, Limitations, and Impact

K.1 Prompts

In this section, we provide the prompts used for math and code generation. Each prompt is shown inside the chat template of the model that receives it. Placeholders in braces denote the problem-specific fields, and long lines are wrapped for display. LLaDA-1.5 uses the template of LLaDA-8B. Generation length is 256 in every experiment.

LLaDA-8B and LLaDA-1.5

```
<|startoftext|><|start_header_id|>user<|end_header_id|>

{question}
Please reason step by step, and put your final answer within
\boxed{}.<|eot_id|><|start_header_id|>assistant<|end_header_id|>
```

Dream-7B

```
<|im_start|>system
You are a helpful assistant.<|im_end|>
<|im_start|>user
{question}
Please reason step by step, and put your final answer within \boxed{}.<|im_end|>
<|im_start|>assistant
```

Figure 16: Math prompt for GSM8K and MATH-500, in the chat template of each model.

HumanEval

```
<|startoftext|><|start_header_id|>user<|end_header_id|>

Complete the following Python function. Return the complete function, including its signature, in a
single ```python code block.

```python
{function stub}
```<|eot_id|><|start_header_id|>assistant<|end_header_id|>
```

MBPP

```
<|startoftext|><|start_header_id|>user<|end_header_id|>

{task description}
Your code should pass this test:
{first test assertion}
Return the complete Python function in a single ```python code
block.<|eot_id|><|start_header_id|>assistant<|end_header_id|>
```

Figure 17: Code prompts for HumanEval (top) and MBPP (bottom), in the LLaDA-8B chat template.

Extraction. Answers are extracted from the last boxed expression, with a fallback to the last number in the generation. Correctness is exact numeric match to the reference after normalization (Section G.3).

K.2 Qualitative Examples

In this section, we provide complete generations of SPEAR, STEPWISE and CONF64 on eight math problems (Figures 19 to 26) and of SPEAR and CONF64 on one HumanEval problem (Figure 18). All come from LLaDA-8B at $\tau = 0.6$ with generation length 256 and are shown unedited, except that the end-of-sequence token is dropped.

Coherence of exact sampling. Exact AR-order sampling produces fluent step-by-step reasoning, not merely a distributional match in aggregate. On the GSM8K sprint question (Figure 19), SPEAR generates "To determine the total number of meters James runs in a week, we need to break down the

problem into smaller, manageable steps" and proceeds, step by step, to the correct product. Fluency is not correctness, and we flag one case where the two come apart. On the MATH-500 double-sum question (Figure 23), which asks for $\sum_{j,k\geq 1}(j+k)^{-3}$ in terms of $p = \sum_k k^{-2}$ and $q = \sum_k k^{-3}$, the sampler produces a well-formed derivation that reaches $\boxed{q}$. The correct value is $p - q$, since $\sum_{j,k\geq 1}(j+k)^{-3} = \sum_{n\geq 2}(n-1)n^{-3}$, and the step that replaces $\sum_k (j+k)^{-3}$ by $\sum_k k^{-3}$ is invalid. An earlier version of this appendix described this generation as a correct expansion. That was our error. It illustrates the point of Section G.3 rather than undermining it: exact sampling reproduces the AR-order distribution of the model, including the reasoning errors the model makes, and we make no claim about the mathematical validity of any individual generation.

Artifacts of lossy parallel decoding. Confidence decoding at a low step count, which commits several tokens per forward pass from uncorrected marginals, produces visible token-level artifacts that the exact samplers do not. On the file-download question (Figure 20), the confidence decoder writes "`the 2 we0 GB file`", "`the first00 GB`" and "`takes minutes20`", with corrupted numbers and misplaced tokens. On the double-sum question (Figure 23) it writes "`(j + ) ^33`" and "`TheTherefore`". On code the same artifacts are fatal. On HumanEval/55 (Figure 18), CONF64 writes a correct loop but garbles the function signature into three lines, so the program does not parse and scores zero, while SPEAR returns a program that passes the tests. These are the local inconsistencies expected when tokens are committed in parallel without the accept-and-reject correction, and they illustrate at the token level the drift quantified in Section C.4.

Prompt (HumanEval/55)

````
Complete the following Python function. Return the complete function, including its signature, in a
single ```python code block.

```python


def fib(n: int):
    """Return n-th Fibonacci number.
    >>> fib(10)
    55
    >>> fib(1)
    1
    >>> fib(8)
    21
    """
```
````

SPEAR (passes the tests)

````
```python
def fib(n: int) -> int:
    """Return n-th Fibonacci number.
    >>> fib(10)
    55
    >>> fib(1)
    1
    >>> fib(8)
    21
    """
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        a, b = 0, 1
        for _ in range(2, n + 1):
            a, b = b, a + b
        return b
```
````

conf64 (does not parse)

````
```python Below is the complete Python function:

def fib(n: int
    int
    :
    """Return n-th Fibonacci number.
    >>> fib(10)
    55
    >>> fib(1)
    1
    >>> fib(8)
    21
    """
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        a, b = 0, 1
        for _ in range(2, n + 1):
            a, b = b, a + b
        return b
```
````

Figure 18: Generations on HumanEval/55: CONF64 garbles the function signature.

Question

```
James decides to run 3 sprints 3 times a week. He runs 60 meters each sprint. How many
total meters does he run a week?
```

SPEAR (exact AR order)

```
To determine the total number of meters James runs in a week, we need to break down
the problem into smaller, manageable steps. 1. **Calculate the number of sprints James
runs in a week:** - James runs 3 sprints each time. - He runs 3 times a week. -
Therefore, the total number of sprints in a week is: \[ 3 \text{ sprints/time} \times
3 \text{ times/week} = 9 \text{ sprints/week} \] 2. **Calculate the total distance
James runs in a week:** - Each sprint is 60 meters long. - There are 9 sprints in a
week. - Therefore, the total distance in a week is: \[ 9 \text{ sprints/week} \times
60 \text{ meters/sprint} = 540 \text{ meters/week} \] Thus, the total number of meters
James runs in a week is \(\boxed{540}\).
```

Stepwise (exact reference)

```
To determine how many total meters James runs in a week, we need to break down the
problem into smaller, manageable steps. Step 1: Calculate the distance James runs in
one sprint. James runs 60 meters each sprint. Step 2: Calculate how many sprints James
runs in one week. James runs 3 sprints 3 times a week. Therefore, the total number of
sprints in one week is: \[ 3 \text{ sprints/session} \times 3 \text{ sessions/week} =
9 \text{ sprints/week} \] Step 3: Calculate the total distance James runs in one week.
Since he runs 60 meters each sprint and he runs 9 sprints in one week, we can find the
total distance by multiplying the number of sprints by the distance of each sprint: \[
9 \text{ sprints/week} \times 60 \text{ meters/sprint} = 540 \text{ meters/week} \]
Step 4: Write the final answer within \boxed{}. The total distance James runs in a
week is \boxed{540} meters.
```

conf64 (64 steps)

```
To determine the total distance James runs in a week, we need to break down the
problem into smaller, manageable steps. 1. **Calculate the distance James runs in one
day:** - James runs 3 sprints each day. - Each sprint is 60 meters long. - Distance,
the distance he runs in one day: \[ 3 \text{ sprints/day} \times 60 \text{
meters/sprint} = 180 \text{ meters/day} \] 2. **Calculate the distance James runs in
one week:** - James runs 3 times a week. - Therefore, the distance he runs in one
week: \[ 180 \text{ meters/day} \times 3 \text{ days/week} = 540 \text{ meters/week}
\] Thus, the total distance James runs in a week is \(\boxed{540}\) meters.
```

Figure 19: Generations on the GSM8K sprint question.

Question

```
Carla is downloading a 200 GB file. Normally she can download 2 GB/minute, but 40% of
the way through the download, Windows forces a restart to install updates, which takes
20 minutes. Then Carla has to restart the download from the beginning. How load does
it take to download the file?
```

SPEAR (exact AR order)

```
1. Calculate the time it takes to download 40% of the file: - 40% of 200 GB = 0.4 *
200 GB = 80 GB - Download speed = 2 GB/minute - Time to download 80 GB = 80 GB / 2
GB/minute = 40 minutes 2. Calculate the time it takes to download the remaining 60% of
the file: - 60% of 200 GB = 0.6 * 200 GB = 120 GB - Download speed = 2 GB/minute -
Time to download 120 GB = 120 GB / 2 GB/minute = 60 minutes 3. Add the time for the
restart: - Time for the restart = 20 minutes 4. Sum up all the times: - Total time =
40 minutes + 60 minutes + 20 minutes = 120 minutes Therefore, the total time taken to
download the file is \boxed{120} minutes.
```

Stepwise (exact reference)

```
1. Carla starts downloading a 200 GB file, and she downloads at a rate of 2 GB per
minute. 2. After downloading 40% of the file, she has downloaded 0.4 * 200 GB = 80 GB.
3. During this period, she downloads 80 GB / 2 GB/minute = 40 minutes. 4. After the
restart, she needs to download the remaining 200 GB - 80 GB = 120 GB. 5. The time to
download the remaining 120 GB is 120 GB / 2 GB/minute = 60 minutes. 6. The total time
for the download from the beginning to the restart is 40 minutes + 60 minutes = 100
minutes. 7. Additionally, the restart takes 20 minutes. 8. Therefore, the total time
to download the file, including the restart, is 100 minutes + 20 minutes = 120
minutes. So, the final answer is \boxed{120} minutes.
```

conf64 (64 steps)

```
To determine the long time it takes for Carla to download the 2 we0 GB file, we need
to break down the process into two steps. First. **First Part Download:** - Carla
downloads the first00 GB of the file at a rate of 2 GB per minute. - Time taken for
100 GB = 2 GB/minute = 50 minutes. 2. **Restart and Download:**:** - the first 00 GB
of the file, Carla forces a restart, install updates, which takes minutes20. - the
Carla has to restart the the download from the beginning, - Carla then downloads the
the remaining 10 GB of the file at a rate of 2 GB per minute. - Time taken for the
remaining00 GB = 2 GB/minute = 10 minutes. 3. **Total Time:** - - Total time = 50
minutes (first part) + 20 minutes (restart) + 10 minutes (second part) = 80 minutes.
Therefore, the total it taken to download the file is \boxed{80} minutes.
```

Figure 20: Generations on the GSM8K file-download question.

Question

```
Toula went to the bakery and bought various types of pastries. She bought 3 dozen
donuts which cost $68 per dozen, 2 dozen mini cupcakes which cost $80 per dozen, and 6
dozen mini cheesecakes for $55 per dozen. How much was the total cost?
```

SPEAR (exact AR order)

```
To determine the total cost of the pastries that Toula bought, we need to calculate
the cost of each type of pastry separately and then sum these costs. Step 1: Calculate
the cost of the donuts. Toula bought 3 dozen donuts at $68 per dozen. \[ 3 \times 68 =
204 \] Step 2: Calculate the cost of the mini cupcakes. Toula bought 2 dozen mini
cupcakes at $80 per dozen. \[ 2 \times 80 = 160 \] Step 3: Calculate the cost of the
mini cheesecakes. Toula bought 6 dozen mini cheesecakes at $55 per dozen. \[ 6 \times
55 = 330 \] Step 4: Calculate the total cost by summing the costs of the donuts, mini
cupcakes, and mini cheesecakes. \[ 204 + 160 + 330 = 694 \] Therefore, the total cost
of the pastries is \(\boxed{694}\).
```

Stepwise (exact reference)

```
To determine the total cost of the pastries, we need to calculate the cost of each
type of pastry separately and then sum these costs. 1. **Cost of the donuts:** - Toula
bought 3 dozen donuts. - Cost per dozen donuts = $68 - Total cost for donuts = \( 3
\times 68 = 204 \) 2. **Cost of the mini cupcakes:** - Toula bought 2 dozen mini
cupcakes. - Cost per dozen mini cupcakes = $80 - Total cost for mini cupcakes = \( 2
\times 80 = 160 \) 3. **Cost of the mini cheesecakes:** - Toula bought 6 dozen mini
cheesecakes. - Cost per dozen mini cheesecakes = $55 - Total cost for mini cheesecakes
= \( 6 \times 55 = 330 \) Now, we sum the costs of all the pastries to find the total
cost: \[ \text{Total cost} = 204 + 160 + 330 = 694 \]
```

conf64 (64 steps)

```
To determine the total cost of the pastries Toula bought, we need to calculate the
cost of each type of pastry separately and then sum these costs. 1. **Cost of the
donuts:** - Toula bought 3 dozen of donuts. - Cost per dozen: $68 - Total cost for
donuts: \(3 \times 68 = 204\) 2. **Cost of the mini cupcakes:** - Toula bought 2 dozen
of mini cupcakes. - Cost per dozen: $80 - Total cost for mini cupcakes: \(2 \times 80
= 160\) 3. **Cost of the mini cheesecakes:** - Toula bought 6 dozen of mini
cheesecakes. - Cost per dozen: $55 - Total cost for mini cheesecakes: \(6 \times 55 =
330\) Now, we sum the costs of all the pastries: \[204 + 160 + 330 = 694\] Thus, the
total cost of the pastries is \(\boxed{694}\).
```

Figure 21: Generations on the GSM8K bakery question.

Question

```
A merchant wants to make a choice of purchase between 2 purchase plans: jewelry worth
$5,000 or electronic gadgets worth $8,000. His financial advisor speculates that the
jewelry market will go up 2.5% while the electronic gadgets market will rise 1.2%
within the same month. If the merchant is looking to maximize profit at the end of
this month by making a choice, how much profit would this be?
```

SPEAR (exact AR order)

```
To determine the profit the merchant would make from each purchase plan, we need to
calculate the potential increase in value for each item based on the given percentage
increase. 1. **Jewelry Purchase Plan:** - Current value of jewelry: $5,000 - Potential
increase: 2.5% of $5,000 = \( \frac{2.5}{100} \times 5000 = 125 \) - Profit from
jewelry: 125 2. **Electronic Gadgets Purchase Plan:** - Current value of electronic
gadgets: $8,000 - Potential increase: 1.2% of $8,000 = \( \frac{1.2}{100} \times 8000
= 96 \) - Profit from electronic gadgets: 96 To maximize profit, the merchant should
choose the purchase plan that yields the higher potential increase in value. In this
case, the jewelry purchase plan yields a higher profit of $125. Therefore, the maximum
profit the merchant would make by choosing the jewelry is \(\boxed{125}\).
```

Stepwise (exact reference)

```
To determine the profit, we need to calculate the future value of each purchase after
the market increase. Since the problem states that the merchant is looking to maximize
profit, we will assume that the merchant will sell the items immediately after the
purchase. First, let's calculate the future value of the jewelry: - Initial price of
jewelry: $5,000 - Increase in price: 2.5% - Future price of jewelry: \( 5000 \times
1.025 = 5125 \) Next, let's calculate the future value of the electronic gadgets: -
Initial price of electronic gadgets: $8,000 - Increase in price: 1.2% - Future price
of electronic gadgets: \( 8000 \times 1.012 = 8096 \) Now, we compare the future
prices: - Future price of jewelry: $5,125 - Future price of electronic gadgets: $8,096
The merchant should choose the electronic gadgets to maximize profit. Therefore, the
profit would be: \[ \boxed{8096} \]
```

conf64 (64 steps)

```
To determine the profit from merchant's purchase of jewelry or electronic gadgets, we
first need to calculate the future value of each item after the given market increase.
1. **Jewelry:** - Current value: $5000 - Market increase: 2.5% - Future value: \(50000
\0 0.025 = 5000 \times 1.025 = 5125\) 2. **Electronic Gadgets:** - Current value:
$8000 - Market increase: 1.2% - Future value: \(8000 \times 1.012 = 8000 \times 1.012
= 8096\) Next, we calculate the profit from each purchase: - Future profit from
jewelry: \(5125 - 5000 = 125\) - Future profit from electronic gadgets: \(8096 - 8000
= 96\) Therefore, the profit from the merchant to purchasing the jewelry is
\(\boxed{125}\).
```

Figure 22: Generations on the GSM8K purchase-plan question.

Question

```
Define \[p = \sum_{k = 1}^\infty \frac{1}{k^2} \quad \text{and} \quad q = \sum_{k =
1}^\infty \frac{1}{k^3}.\]Find a way to write \[\sum_{j = 1}^\infty \sum_{k =
1}^\infty \frac{1}{(j + k)^3}\]in terms of $p$ and $q.$
```

SPEAR (exact AR order)

```
We can write \[\sum_{j = 1}^\infty \sum_{k = 1}^\infty \frac{1}{(j + k)^3} = \sum_{j =
1}^\infty \left( \sum_{k = 1}^\infty \frac{1}{(j + k)^3} \right) = \sum_{j = 1}^\infty
\left( \sum_{k = 1}^\infty \frac{1}{k^3} \right) = \boxed{q}.\]
```

Stepwise (exact reference)

```
First, we compute the double sum \[\sum_{j = 1}^\infty \sum_{k = 1}^\infty \frac{1}{(j
+ k)^3}.\] We have \begin{align*} \sum_{j = 1}^\infty \sum_{k = 1}^\infty \frac{1}{(j
+ k)^3} &= \sum_{j = 1}^\infty \sum_{k = 1}^\infty \frac{1}{k^3}. \end{align*} Then,
\begin{align*} \sum_{j = 1}^\infty \sum_{k = 1}^\infty \frac{1}{k^3} &= \sum_{k =
1}^\infty \sum_{j = 1}^\infty \frac{1}{k^3}. \end{align*} Now, let's consider the sum
\[\sum_{k = 1}^\infty \sum_{j = 1}^\infty \frac{1}{k^3}.\] This is essentially
\[\sum_{k = 1}^\infty \frac{1}{k^3} = q.\] Therefore, the double sum we want to
compute is equal to \boxed{q}.
```

conf64 (64 steps)

```
We can express the double sum as \[\sum_{j = 1}^\infty \sum_{k = 1 - 1}^\infty
\frac{1}{(j + k)^3} = \sum_{j = 1}^\infty \sum_{k = 1}^\infty \frac{1}{(j + k)^3} =
\sum_{j = 1}^\infty \sum{1 = 1}^\infty \frac{1}{(j + )^33} = \sum_{j = 1}^\infty
\frac{1}{j^ \cdot \frac{1}{j^3} = \sum_{j = 1}^\infty \frac{1}{j^3} = qsum_{j =
1}^\infty \frac{1}{j^3} = qsum_{j = 1}^\infty \frac{1}{j^3} = qsum_{j = 1}^\infty
\frac{1}{j^3} = qsum_{j = 1}^\infty \frac{1}{j^3} = qsum_{j = 1}^\infty \frac{1}{j^3}
= q.\] TheTherefore, \boxed{q}.
```

Figure 23: Generations on the MATH-500 double-sum question.

Question

```
A regular hexagon can be divided into six equilateral triangles. If the perimeter of
one of the triangles is 21 inches, what is the perimeter, in inches, of the regular
hexagon?
```

SPEAR (exact AR order)

```
To determine the perimeter of the regular hexagon, we need to follow these steps: 1.
**Understand the relationship between the hexagon and the equilateral triangles:** A
regular hexagon can be divided into 6 equilateral triangles. This means that the
perimeter of one equilateral triangle is the same as the perimeter of the hexagon
divided by 6. 2. **Determine the perimeter of one equilateral triangle:** We are given
that the perimeter of one equilateral triangle is 21 inches. 3. **Calculate the
perimeter of the hexagon:** Since the hexagon is made up of 6 equilateral triangles,
we multiply the perimeter of one equilateral triangle by 6. \[ \text{Perimeter of the
hexagon} = 6 \times \text{Perimeter of one equilateral triangle} \] Substituting the
given perimeter of one equilateral triangle: \[ \text{Perimeter of the hexagon} = 6
\times 21 = 126 \text{ inches} \] Therefore, the perimeter of the hexagon is
\boxed{126}.
```

Stepwise (exact reference)

```
To find the perimeter of the regular hexagon, we need to follow these steps: 1.
**Understand the relationship between the hexagon and the equilateral triangles:** - A
regular hexagon can be divided into six equilateral triangles. - Each equilateral
triangle has a perimeter of 21 inches. 2. **Determine the side length of one
equilateral triangle:** - The perimeter of an equilateral triangle is given as 21
inches. - Since an equilateral triangle has three sides, we can find the side length
by dividing the perimeter by 3. - Side length of one equilateral triangle =
\(\frac{21}{3} = 7\) inches. 3. **Calculate the perimeter of the hexagon:** - The
hexagon is composed of six equilateral triangles. - Each side of the hexagon is the
same as the side length of one equilateral triangle. - Therefore, the perimeter of the
hexagon is \(6 \times 7 = 42\) inches. 4. **Present the final answer:** - The
perimeter of the hexagon is \(\boxed{42}\) inches.
```

conf64 (64 steps)

```
To determine the perimeter of the regular hexagon, we need to follow these steps: 1.
**Understand the relationship between the hexagon and the equilateral triangles:** - A
regular hexagon can be divided into six equilateral triangles. - Each equilateral
triangle has three equal sides. 2. **Calculate the side length of one equilateral
triangle:** - Given perimeter of one equilateral triangle = 21 inches. - - Since an
equilateral triangle has three sides,, the side length one equ is triangle = \(
\frac{1}{3} \times 21 = 7 \) inches. 3. **Determine the side length of the hexagon:**
- Since each side of the equilateral triangle is also a side of the regular hexagon,
the side length of the hexagon = 7 inches. 4. **Calculate the perimeter of the
hexagon:** - A regular hexagon has six equal sides. - Therefore, the perimeter of the
hexagon = \( 6 \times 7 = 42 \) inches. Thus, the perimeter of the regular hexagon is
\boxed{42} inches.
```

Figure 24: Generations on the MATH-500 hexagon question.

Question

```
The proper divisors of 12 are 1, 2, 3, 4 and 6. A proper divisor of an integer $N$ is
a positive divisor of $N$ that is less than $N$. What is the sum of the proper
divisors of the sum of the proper divisors of 284?
```

SPEAR (exact AR order)

```
To solve the problem, we need to follow a step-by-step approach: **Step 1: Find the
proper divisors of 284.** A proper divisor of an integer \( N \) is a positive divisor
of \( N \) that is less than \( N \). Therefore, the proper divisors of 284 are: \[ 1,
2, 4, 7, 14, 46 \] **Step 2: Calculate the sum of the proper divisors of 284.** We sum
the proper divisors of 284: \[ 1 + 2 + 4 + 7 + 14 + 46 = 74 \] **Step 3: Find the
proper divisors of 74.** The proper divisors of 74 are: \[ 1, 2, 37 \] **Step 4:
Calculate the sum of the proper divisors of 74.** We sum the proper divisors of 74: \[
1 + 2 + 37 = 40 \] **Final Answer:** \boxed{40}
```

Stepwise (exact reference)

```
To solve the problem, we need to follow a step-by-step approach. Let's break it down:
### Step 1: Find the proper divisors of 284 A proper divisor of an integer \( N \) is
a positive divisor of \( N \) that is less than \( N \). The proper divisors of 284
are: \[ 1, 2, 4, 7, 14, 46 \] ### Step 2: Calculate the sum of the proper divisors of
284 Add the proper divisors together: \[ 1 + 2 + 4 + 7 + 14 + 46 = 74 \] ### Step 3:
Find the proper divisors of 74 The proper divisors of 74 are: \[ 1, 2, 7, 11 \] ###
Step 4: Calculate the sum of the proper divisors of 74 Add the proper divisors
together: \[ 1 + 2 + 7 + 11 = 21 \] ### Final Answer The final answer is
\(\boxed{21}\).
```

conf64 (64 steps)

```
To find the sum of the proper divisors of the sum of the proper divisors of 284, we
will follow these steps: 1Step 1: Find the proper divisors of 284. The proper divisors
of 284 are 1, 2, 4, 8, 14, 28, 56, and 112. 2Step 2: Calculate the sum of 284's proper
divisors. The sum of 284's proper divisors is \(1 + 2 + 4 + 8 + 14 + 28 + 56 + 112 =
225\). 3Step 3: Find the proper divisors of 225. The proper divisors of 225 are 1 and
5.. StepStep 4: Calculate the sum of 225's proper divisors. The sum of 225's proper
divisors is \(1 + 5 = 6\). Therefore, the sum of the proper divisors of the sum of the
proper divisors of 284 is \(\boxed{6}\).
```

Figure 25: Generations on the MATH-500 proper-divisor question.

Question

```
Evaluate $(1+2i)6-3i$.
```

SPEAR (exact AR order)

```
We need to evaluate \((1+2i)(6-3i)\). First, we distribute the \(1 + 2i\) to each term
inside the parentheses: \[ (1+2i)(6-3i) = 1 \cdot 6 + 1 \cdot (-3i) + 2i \cdot 6 + 2i
\cdot (-3i) \] Simplify each term: \[ = 6 - 3i + 12i - 6i^2 \] Next, we simplify the
expression by combining like terms: \[ = 6 + (12i - 3i) - 6i^2 \] Since \(i^2 = -1\),
we can substitute this in: \[ = 6 + 9i - 6(-1) \] Simplify further: \[ = 6 + 9i + 6 \]
\[ = 12 + 9i \] Therefore, the final answer is: \[ \boxed{12 + 9i} \]
```

Stepwise (exact reference)

```
To evaluate \((1+2i)(6-3i)\), we will use the distributive property (also known as the
FOIL method for binomials), but extended to complex numbers. \[ (1+2i)(6-3i) = 1 \cdot
6 + 1 \cdot (-3i) + 2i \cdot 6 + 2i \cdot (-3i) \] Now, we perform the
multiplications: \[ = 6 - 3i + 12i - 6i^2 \] Recall that \(i^2 = -1\), so we
substitute \(-1\) for \(i^2\): \[ = 6 - 3i + 12i + 6 \] Next, we combine the real
parts and the imaginary parts separately: \[ = (6 + 6) + (-3i + 12i) \] \[ = 12 + 9i
\] Thus, the final answer is: \[ \boxed{12 + 9i} \]
```

conf64 (64 steps)

```
To evaluate \((1+2i)(6-3i)\), we will use the distributive property (also known as the
FOIL method for binomials). 1, Multiply the first terms: \(1 \cdot 6 = 6\). 2,
Multiply the outer terms: \(1 \cdot (-3i) = -3i\). 3, Multiply the inner terms: \(2i
\cdot 6 = 12i\). 4, Multiply the last terms: \(2i \cdot (-3i) = -6i^2\). Now, combine
all these results: \[6 - 3i + 12i - 6i^2.\] Recall that \(i^2 = -1\), so: \[6 - 3i +
12i - 6(-1) = 6 - 3i + 12i + 6.\] Combine the like parts: \[6 + 6 + (-3i + 12i) = 12 +
9i.\] Therefore, the answer is \(\boxed{\{12+9i\}}\).
```

Figure 26: Generations on the MATH-500 complex-arithmetic question.

K.3 Discussion

The experiments together support a single picture. A masked diffusion language model is highly self-consistent. The token it would draft from a one-shot marginal usually matches the token it would commit under the exact autoregressive order. The two disagree mainly at a minority of high-entropy forking positions (Sections C.1, C.2). This self-consistency is stable across models, datasets, sequence positions, and generation lengths, and it degrades smoothly with temperature (Sections C.3, E.3). It is the property that makes self-speculative sampling efficient. Its strength is captured by a single quantity: the divergence between the one-shot marginals and the exact conditionals. That divergence governs acceptance, and hence tokens per forward pass (Section E.4).

The consequence for decoding is that the exact autoregressive-order distribution, previously reachable only by stepwise decoding at one forward pass per token, is reachable at a fraction of the forward-pass count. What this buys is quality per forward pass, and raw wall-clock only where lossy decoding breaks the output, as a fixed-step decoder does on code (Appendix I). The exact distribution has higher single-shot accuracy than matched-budget confidence decoding, significantly on two models and four datasets (Sections D.1, H.8, I). It attains this while committing several tokens per forward pass. The distributional error that lossy decoders incur, and that SPEAR removes, is measurable and grows with the aggressiveness of parallel commitment (Section C.4). It manifests qualitatively as token-level artifacts in the lossy output (Section K.1).

The picture also bounds the method. Verification is a batched multi-row forward pass rather than a single causal pass (Section E.5). The forward-pass reduction is therefore not a proportional wall-clock reduction, and the sweet spot in block length is modest (Section E.2). Against CONF64 the advantage is a shift toward the exact distribution rather than a coverage gain, so it is concentrated in single-shot quality. Against the threshold decoder on MATH-500 it also shows up as reach. At the scale we tested, it does not translate into a reinforcement learning improvement (Sections F.1, H.9).

K.4 Negative Results and Non-Findings

We list the results that did not go the way an optimistic reading of the method would predict, so that they are not lost.

- **Wall-clock is not a large speedup.** The forward-pass reduction is $3.5\times$ at matched quality, but wall-clock is at parity with the strongest baseline, because verification is a batched multi-row forward pass rather than a single causal pass (Section E.5).
- **Reinforcement learning shows no quality gain.** Exact SPEAR rollouts and biased confidence rollouts differ by -0.008 held-out Pass@1 after 600 steps, well inside the evaluation noise, and acceptance does not rise during training (Figure 14). We report no significance test, because the design gives one trajectory per condition. The self-consistification hypothesis is not supported at LoRA scale.
- **Coverage depends on the baseline.** On GSM8K SPEAR solves essentially the same problems as CONF64 and CONF256, the solved sets differing by at most one. On MATH-500 the sets differ almost symmetrically against CONF64 (29 against 27), so there the advantage is single-shot rate. Against FASTDLLM SPEAR also reaches more problems (50 against 21; Table 39).
- **The sublinear variant is not usable on real models.** Its error is governed by the expected clipping excess, and the per-position log-ratio on LLaDA-8B is heavy-tailed enough that a clip level avoiding bias at $L = 8$ would cost about e^{508} position evaluations per block. It is validated on a toy oracle only (Appendix B.2).
- **At a matched time budget SPEAR loses to a cheap lossy decoder.** SPEAR costs about $3.9\times$ a CONF64 sample, so at equal wall-clock CONF64 draws about four times as many samples and reaches a higher Pass@k at every budget we can estimate (-0.065 at 20s, -0.036 at 80s, all CIs excluding zero; Appendix H.3). The quality-per-forward-pass advantage does not transfer to quality per second on math. On code it does against CONF64, whose programs often fail to parse, but not against the threshold decoder (Appendix I).
- **A strong lossy baseline matches SPEAR on single-shot accuracy.** The threshold decoder of Wu et al. (2026) reaches Pass@1 0.800 at 71 forward passes against SPEAR's 0.792 at 73,

a difference of -0.008 whose CI includes zero (Table 23). SPEAR is ahead of it on coverage ($+0.033$ at Pass@8), so the claim we support is that SPEAR is not on the quality-coverage trade-off, not that it wins on single-shot accuracy outright.

- **The advantage is single-shot, not high-k.** The paired Pass@k advantage is significant only at small k (Pass@1 and Pass@2) and reverses without significance at the largest k, where ultra-parallel diversity helps the lossy baseline (Table 25).
- **Our MATH-500 grader was broken three times.** The first version could not match nested braces; the second delegated equivalence to a library whose parser drops symbolic content; the third failed on unbraced arguments. The regrade changed 586 of 26,096 stored labels, and on the original 96 problems SPEAR moved from 0.296 to 0.381 and CONF64 from 0.252 to 0.325 (Appendix G.3). The paired conclusion survived every repair and a MATH-500 exactness interval that had excluded zero under the first grader now includes it, but the absolute numbers in earlier versions did not survive, and we would not have found the second or third defect without an external check.
- **Batched verification silently biases a non-batch-invariant backbone.** Dream-7B returns different logits for the same row depending on the batch it shares, by TV ≈ 0.018 at batch eight, on every attention backend we tested. Verifying one row per forward pass restores exactness bitwise, at 353 calls per sample against 64 for CONF64, so on that backbone SPEAR is exact but not cheap (Section G.4).

Reporting these is part of the contribution: the method delivers an exact distribution at reduced forward-pass cost, and we are explicit about what that does and does not buy.

K.5 REPRODUCIBILITY CHECKLIST

- **Code.** `https://anonymous.4open.science/r/spear-code-iclr27-5C0B/`: the sampler, the baselines, the job script of every run, the analysis scripts and every logged sample. Its `verify.sh` recomputes every number in the paper from the logged samples.
- **Models.** LLaDA-8B-Instruct, LLaDA-1.5 and Dream-7B, loaded with their released remote code in bfloat16. Mask token id 126336 for LLaDA and 151666 for Dream.
- **Data.** GSM8K test and train splits, the MATH-500 test split, HumanEval, and the sanitized MBPP test split. Training and evaluation splits are disjoint.
- **Hardware.** A single H100 GPU per experiment. All samplers are training-free. The reinforcement learning study uses LoRA.
- **Determinism.** Seeds are fixed. The exactness of SPEAR is independent of the seed. Reported differences use bootstrap confidence intervals over problems.
- **Metrics.** Pass@k by the unbiased estimator. Acceptance and KL from logged per-position records. Wall-clock with CUDA synchronization.
- **Full configuration.** Table 20.
- **Every reported number** is computed from logged samples by the analysis scripts. No numbers are transcribed by hand from training logs.
- **Compute.** Each sampler experiment runs on one H100 GPU. The Pass@k, timing, sweep, self-consistency and code experiments together are on the order of a hundred and twenty GPU-hours, dominated by the confidence baselines at high step counts and the stepwise references. The two 600-step and five 120-step LoRA-GRPO runs add roughly thirty GPU-hours. The toy-oracle validations run on CPU.

K.6 FUTURE DIRECTIONS

Several extensions follow directly from the analysis.

Key-value-cached verification. The wall-clock cost of SPEAR is the L-row verification batch (Section E.5). The L revealed-prefix inputs share the committed prefix $o_{<p}$ and differ only in how many block tokens are revealed. A verification scheme that cached the contribution of the shared prefix

and recomputed only the block region would reduce the per-row cost. The obstacle is that bidirectional attention makes the prefix representation depend on the masked suffix, so the cache is not just the prefix keys and values. A block-diagonal or windowed attention approximation might recover most of the saving while keeping the error small. That would turn the forward-pass reduction into a wall-clock reduction.

Diversity within exact sampling. The two axes of Section H.5 are separable. One could apply a diversity-inducing mechanism within exact AR-order sampling, such as a higher sampling temperature restricted to forking tokens or a determinantal reweighting across samples. This would aim to combine the single-shot quality of the exact distribution with the high-k coverage of ultra-parallel decoding, which neither method achieves alone.

Variance-reduced sublinear verification. The depth-weighted importance distribution of Section J.4 reduces the estimator variance and hence the number of position evaluations. Combined with a key-value-cached single-position query, this would make the sublinear variant competitive with batched verification even on compute-bound hardware, extending SPEAR to the long-sequence regime where the batch does not fit in memory.

Broader model coverage. The Dream-7B integration failed under our batched verification (Section G.4). Reconciling the exactness check with the Dream forward path, and testing on further masked-diffusion families as they are released, would establish how architecture-independent the self-consistency phenomenon is. The temperature and block-length sweeps suggest the phenomenon is a property of masked diffusion training rather than of a specific model, which this would confirm.

Reinforcement learning at scale. The null reinforcement learning result is at LoRA scale and 600 steps (Section F.1). A full fine-tune over more steps could use the exact log-probabilities that SPEAR supplies for free. It would test whether the sampler-learner mismatch that the exactness removes matters when the policy is pushed harder. Our scale cannot resolve this.

K.7 Limitations, Reproducibility, and Broader Impact

Limitations. The wall-clock benefit is modest (1.15 to $1.25\times$) and exists only when one sample occupies the GPU: it is gone at 1024 tokens and at four samples per forward pass (Appendix H.7), because bidirectional attention forbids the single-pass verification that makes AR speculative decoding fast. The forward-pass reduction becomes a wall-clock reduction only in memory-bound or key-value-cacheable regimes. Two of our three models, LLaDA-8B and the preference-optimized LLaDA-1.5, share an architecture and most weights, so on their own they would test post-training rather than model families. Dream-7B supplies the second architecture, on a Qwen backbone, and it requires verifying one row per forward pass because its backbone is not batch-invariant (Appendix G.4). Three models across two families is still narrow, and we have not tested a masked diffusion model outside these two lineages. The MATH-500, HumanEval and MBPP measurements use the full benchmarks, but the GSM8K measurements use subsets (96 problems for LLaDA-8B and 32 for LLaDA-1.5 and Dream-7B) rather than the full test set, and the acceptance and timing sweeps use 16 to 32 prompts, so the intervals we report are correspondingly wide. The reinforcement learning study is a feasibility demonstration with a null quality result at LoRA scale. The theory transplants known results (speculative sampling, the Bernoulli factory) to the discrete AR-order setting rather than proving new bounds.

Reproducibility. Every reported number is computed from logged samples by the analysis scripts. We report model identifiers, mask token ids, temperatures, block lengths, sample counts, problem counts, LoRA configuration, learning rate, and bootstrap settings above and in the main text. All experiments run on a single H100 GPU, and the samplers are training-free.

Broader impact. SPEAR is an inference-time sampler for existing diffusion language models. It does not change what a model can express. It draws exact samples from a distribution the model already defines. The main effect is to make the exact AR-order distribution accessible at reduced forward-pass cost, which is useful for evaluation, calibration, and reinforcement learning. We see no impact specific to this method beyond those of the underlying language models.