LEARNING WHEN TO COMMIT: SCHEDULE DISTILLATION FOR FAST DIFFUSION LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Diffusion large language models (dLLMs) can generate multiple tokens in parallel, but often require many decoding rounds to complete a response. Efficient generation depends not only on which tokens are easy to predict, but also on how committing to them helps predict the remaining positions. A highly predictable token may provide little useful context, while revealing a less predictable token can help several other positions become predictable sooner. We propose ***dSpark***, a new schedule distillation-based framework for learning when to commit tokens for faster diffusion language generation. To achieve this, we construct unmasking schedules that prioritize tokens whose reveal makes other positions easier to predict, reducing uncertainty earlier in decoding. We then distill these schedules into the model to enable faster parallel sampling, with a KL-divergence regularizer to help preserve generation quality. We extensively evaluate our approach across mathematical reasoning and code generation tasks. On GSM8K, our method achieves remarkable **10.12**$\times$ and **5.12**$\times$ speedups over the various baselines on LLaDA-8B-Instruct and Dream-7B-Instruct, respectively, while maintaining superior task performance.

1 INTRODUCTION

Large language models (LLMs) have achieved remarkable capabilities across natural language understanding (Brown et al., 2020; Ouyang et al., 2022; Achiam et al., 2023; Touvron et al., 2023; Grattafiori et al., 2024; Team et al., 2023), and reasoning (Kojima et al., 2022; Jaech et al., 2024; Guo et al., 2025), yet efficient inference remains a fundamental challenge. Most LLMs adopt an autoregressive (AR) generation paradigm, in which each token is produced sequentially conditioned on all previously generated tokens. This strict left-to-right dependency fundamentally limits decoding parallelism: generating a sequence of L tokens typically requires L sequential decoding steps. Diffusion large language models (dLLMs) offer an appealing alternative by generating sequences through iterative denoising rather than strictly autoregressive factorization. Starting from a partially or fully masked sequence, a dLLM predicts multiple masked positions in parallel and progressively commits predicted tokens over a series of denoising rounds. Recent models such as LLaDA (Nie et al., 2026) and Dream (Ye et al., 2025) demonstrate that this paradigm can attain strong language modeling, reasoning, and coding capabilities while enabling flexible generation orders and substantially greater potential for parallel decoding (Ben-Hamu et al., 2026; Wu et al., 2026b; Chen et al., 2026a). However, realizing this potential in practice requires reducing the large number of iterative model evaluations needed to transform an initially masked sequence into a complete response.

A central factor governing dLLM inference efficiency is the *unmasking schedule*: which token positions should be committed at each decoding round and which should remain masked for subsequent refinement. Given the model's predictive distributions over all currently masked positions, a decoder may reveal one token, a fixed number of tokens, or a dynamically selected subset in each iteration. Committing more positions increases parallelism and reduces the total number of decoding rounds, but overly aggressive commitments may introduce errors that can no longer be corrected and subsequently contaminate the context used to predict other tokens. Conversely, conservative schedules preserve generation quality but sacrifice much of the computational advantage offered by parallel prediction. Existing decoding strategies therefore commonly rely on prediction confidence, for instance, committing positions whose maximum token probabilities are highest, under the intuition that highly confident predictions are safer to reveal. While effective as a simple heuristic, this strat-

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.

egy treats each masked position largely according to its current local predictability and does not explicitly model how revealing that position will change the difficulty of subsequent predictions.

We argue that this distinction between *predictability* and *utility* is crucial for efficient diffusion language generation. The easiest token to predict is not necessarily the most useful token to commit. Consider two masked positions i and j. Position i may already have an extremely peaked predictive distribution, yet revealing it may provide almost no additional information for predicting the rest of the sequence. In contrast, position j may be moderately less certain but encode a critical semantic, syntactic, or reasoning decision, once j is revealed, the uncertainty of several other positions may decrease substantially. This phenomenon is particularly important in structured generation tasks such as mathematical reasoning and code synthesis, where intermediate decisions induce strong dependencies among multiple downstream tokens. A schedule based solely on instantaneous confidence may therefore spend early commitments on tokens that are safe but informationally redundant, leaving influential positions unresolved for many additional rounds. Consequently, the effective decoding problem is not simply *which tokens can be predicted now*, but rather *which tokens should be committed now so that the remaining sequence becomes easier to predict*.

Motivated by this observation, we propose *dSpark*, a novel framework for learning when to commit tokens during diffusion language generation using the proposed *schedule distillation*. Our key idea is to construct informative unmasking schedules according to the downstream effect of each commitment rather than its prediction confidence alone. Specifically, we evaluate candidate commitments by how much revealing them helps reduce predictive uncertainty over the remaining masked positions. Tokens whose revelation makes multiple unresolved positions easier to determine are prioritized earlier, allowing useful context to propagate through the sequence more rapidly. From this perspective, decoding can be viewed as a sequential context-construction process: every committed token not only resolves one position but also modifies the conditional prediction problem faced by all remaining positions. By explicitly accounting for this effect, our schedules favor commitments with high *future predictive value*, producing trajectories that reduce sequence-level uncertainty more rapidly than conventional confidence-driven schedules. This enables substantially more useful progress to be made within each expensive model evaluation.

Constructing better schedules alone, however, can introduce additional computation if sophisticated token selection must be repeatedly performed during inference. We therefore further *distill* the behavior induced by these informative schedules into the language model itself. The constructed schedules provide training trajectories that expose the model to commitment orders in which informative tokens are revealed early and the remaining predictions are progressively simplified. After distillation, the model can exploit these learned dependencies under efficient parallel sampling without requiring expensive search over candidate schedules at every decoding step. Importantly, aggressively optimizing for fast commitment can alter the model's original predictive behavior and degrade generation quality. To mitigate this effect, we introduce a KL-divergence regularization objective that constrains the distilled model to remain close to the original model distribution while learning to operate effectively under accelerated schedules. Schedule distillation thus shifts computation from repeated inference-time schedule optimization into training, yielding a practical mechanism for jointly improving parallelism and preserving the generation capability of the underlying dLLM.

We extensively evaluate *dSpark* on two representative diffusion language models, LLaDA-8B-Instruct and Dream-7B-Instruct, across mathematical reasoning and code generation benchmarks. The results demonstrate that learning *when* to commit is a powerful yet previously underexplored dimension for accelerating diffusion language models. In particular, on GSM8K, our approach achieves $10.12\times$ and $5.12\times$ decoding speedups over the corresponding baselines for LLaDA-8B-Instruct and Dream-7B-Instruct, respectively, while maintaining superior task performance. These results show that the efficiency of diffusion generation is determined not merely by how many tokens are predicted in parallel, but by how effectively each commitment restructures the prediction problem for subsequent rounds.

Our main contributions are three-fold:

- We identify a key limitation of existing diffusion language model decoding strategies: token confidence alone does not capture the downstream utility of a commitment. We show that revealing a less predictable but more informative token can reduce uncertainty over multiple remaining positions and lead to faster overall decoding.

- We propose *schedule distillation*, a new strategy that constructs unmasking schedules by prioritizing tokens according to how much their commitment simplifies future predictions. We further distill these informative schedules into the model, enabling efficient parallel decoding without requiring expensive schedule optimization at inference time.
- We extensively demonstrate the effectiveness of our method across mathematical reasoning and code generation tasks on multiple diffusion language models. In particular, our approach achieves up to 10.12× speedup on LLaDA-8B-Instruct and 5.12× on Dream-7B-Instruct on GSM8K, while maintaining strong generation quality.

2 Related Work

Diffusion Language Models. Discrete diffusion models generate sequences by reversing a corruption process, and subsequent work developed stronger objectives for language modeling, including SEDD, MDLM, and RADD (Austin et al., 2021a; Lou et al., 2024; Sahoo et al., 2024; Ou et al., 2025). LLaDA (Nie et al., 2026) scaled masked diffusion to an instruction-following language model trained from scratch, while Dream (Ye et al., 2025) demonstrated strong reasoning and coding performance with a model initialized from an autoregressive LLM. Other work combines diffusion within blocks with generation across blocks or scales diffusion models further through conversion from autoregressive models (Arriola et al., 2025; Bie et al., 2025; Zhu et al., 2026). These advances establish capable diffusion backbones, but their iterative generation process leaves substantial room to improve inference efficiency.

Efficient Diffusion Language Decoding. Existing methods reduce either the cost or the number of decoding steps. Fast-dLLM (Wu et al., 2026b), dKV-Cache (Ma et al., 2026), Elastic-Cache (Nguyen-Tri et al., 2026), and dInfer (Ma et al., 2025) improve cache reuse and inference execution, while D2F (Wang et al., 2026) and Fast-dLLM v2 (Wu et al., 2026a) use block-based generation to support efficient parallel decoding. Other approaches distill generation into fewer steps or enable more aggressive token updates (Deschenaux & Gulcehre, 2025; Sahoo et al., 2025; Chen et al., 2026a). Closest to our work, dParallel (Chen et al., 2026b) uses certainty-forcing distillation to make tokens ready earlier, whereas d3LLM (Qian et al., 2026) distills pseudo-trajectories and applies multi-block decoding. Training-free approaches such as LEAP (Zhang et al., 2026) and Ripple-Pivot Search (Ye et al., 2026) use lookahead to identify tokens worth committing early.

3 Preliminaries

3.1 Masked Diffusion Language Models

Masked diffusion language models (MDLMs) generate text by learning to recover tokens from partially masked sequences (Austin et al., 2021a). For a clean sequence $x_0 \in \mathcal{V}^L$, the linear forward process independently replaces each token with the absorbing state `MASK` with probability t:

$$q_{t|0}(x_t \mid x_0) = \prod_{i=1}^{L} q_{t|0}(x_t^i \mid x_0^i) = \prod_{i=1}^{L} \mathrm{Cat}\Big(x_t^i;\ (1-t)\delta_{x_0^i} + t\delta_{\texttt{MASK}}\Big), \tag{1}$$

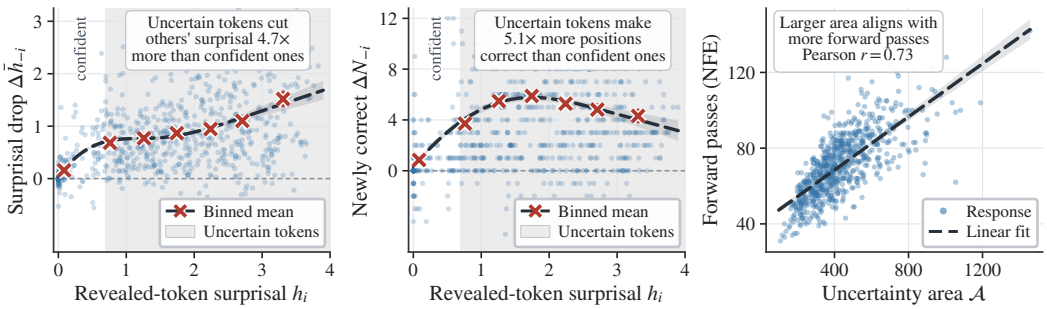
where $\mathcal{V}$ is the token vocabulary, $\texttt{MASK} \notin \mathcal{V}$, Cat denotes a categorical distribution, and δ_v is a point mass at v. The noise level $t \in [0,1]$ interpolates between the clean sequence and complete masking. A bidirectional denoiser $p_\theta(x_0^i \mid x_t)$ is trained to reconstruct masked tokens with the reweighted cross-entropy objective (Sahoo et al., 2024; Lou et al., 2024)

$$\mathcal{L}_{\mathrm{MDM}}(\theta; x_0) = \int_0^1 \frac{1}{t}\, \mathbb{E}_{q_{t|0}(x_t|x_0)} \left[\sum_{i:\, x_t^i = \texttt{MASK}} -\log p_\theta(x_0^i \mid x_t) \right] \mathrm{d}t. \tag{2}$$

This formulation underlies large-scale diffusion language models such as LLaDA (Nie et al., 2026) and Dream-7B (Ye et al., 2025), which predict masked tokens in parallel during iterative generation.

3.2 Parallel Decoding and Unmasking Schedules

Given a fixed prompt, decoding starts from a fully masked response. Let s_r denote the partially decoded sequence at round r and $\mathcal{M}_r$ the masked response positions eligible for commitment, such



(a) Surprisal drop at other positions (b) Newly ready positions (c) Unc. Area vs. forward passes

Figure 1: **The most predictable token is not the most useful one to reveal.** On correct GSM8K responses from LLaDA, we reveal one masked token y_i and measure its effect on the *other* masked positions. In (a) and (b), each dot is one reveal, and the gray region marks uncertain tokens ($p_{\theta_0} < 0.5$), which confidence-based decoders defer. **(a)** Mean surprisal drop at the other positions (nats). **(b)** Net number of other positions that become *ready*, i.e., whose top-1 prediction matches the reference. **(c)** Responses with a larger uncertainty area $\mathcal{A}$ (Eq. (4)) need more forward passes.

as those within an active block or window. A forward pass computes $p_\theta(\cdot \mid s_r, j)$ for each $j \in \mathcal{M}_r$. The decoder selects a nonempty set $S_r \subseteq \mathcal{M}_r$, for example using confidence thresholds (Nie et al., 2026) or predictive-entropy budgets (Ben-Hamu et al., 2026). Under greedy token assignment, the sequence is updated as

$$s_{r+1}^j = \begin{cases} \arg\max_{v \in \mathcal{V}} p_\theta(v \mid s_r, j), & j \in S_r, \\ s_r^j, & j \notin S_r. \end{cases} \tag{3}$$

An *unmasking schedule* $\pi = (S_0, \ldots, S_{T-1})$ specifies which positions are revealed in each round.

4 Methodology

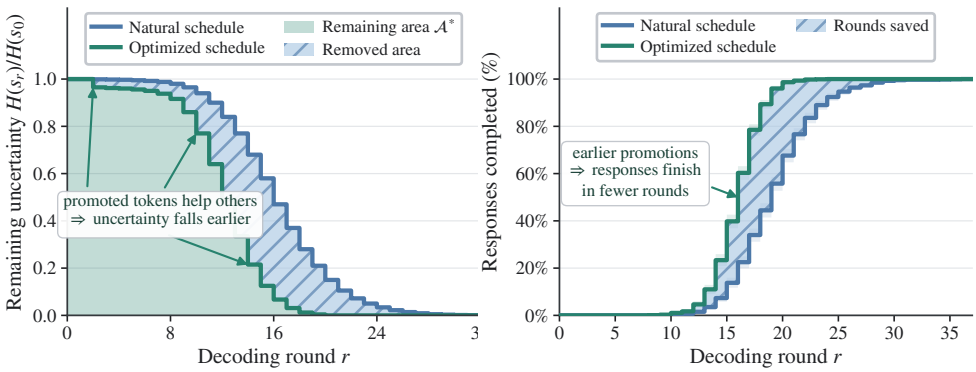
4.1 Motivation

The order in which tokens are revealed can affect how quickly a diffusion language model completes a response. Each reveal provides context for the remaining positions, but some tokens help more than others. We measure this with the frozen base model p_{θ_0} on a correct response y. In a partially decoded state s, the model's uncertainty about the target token at a masked position j is its surprisal $h_j(s) = -\log p_{\theta_0}(y_j \mid s, j)$. Fig. 1a and 1b show that revealing less predictable target tokens (larger h_i) can reduce uncertainty at other positions and help more predictions match the correct response. We call a position *ready* when the model's most likely token matches the ground truth. As shown in Figure 1b, revealing an uncertain token makes $5.1\times$ more other positions ready than revealing a confident one, and Fig. 1a shows that it also lowers the surprisal of the other masked positions $4.7\times$ more. Both averages count positions that become ready minus those that stop being ready. A useful schedule should therefore consider how each token helps predict the remaining positions, as well as how confident the model is about that token.

What matters for efficient decoding is not only how much uncertainty a reveal removes, but how early it removes it. Following the masked-token surprisal sum inside the brackets of Eq. (2), we measure remaining reference uncertainty and its accumulated area as

$$H(s_r; y) = \sum_{j \in \mathcal{M}_r} h_j(s_r), \qquad \mathcal{A}(\pi; y) = \sum_{r=0}^{T-1} H(s_r; y). \tag{4}$$

The area adds up the model's remaining uncertainty at every decoding round. If revealing a token early makes other tokens easier to predict, their uncertainty is lower in the following rounds and the total area is smaller. Two schedules can therefore produce the same response but have different areas. Figure 2a illustrates this: under the natural schedule, the remaining uncertainty stays high for many rounds before it falls, whereas promoting tokens that help the other positions makes it



(a) Uncertainty area (b) Completion of verified responses

Figure 2: **Promoting helpful tokens makes uncertainty fall earlier and responses finish sooner.** Both schedules reveal the same response, in a different order. **(a)** The hatched region is the part of the uncertainty area $\mathcal{A}$ (Eq. (4)) that the optimized schedule removes. **(b)** Share of responses completed by round r. The hatched area shows the mean number of rounds saved.

drop earlier and shifts the whole curve to the left, removing the hatched area from $\mathcal{A}$. Since the response is fixed, this gain comes only from the order of reveals. A smaller area also means faster completion. Figure 1c shows that responses with larger areas under the natural schedule also tend to require more forward passes during generation, and Figure 2b shows that optimized schedules finish verified responses in fewer rounds than the natural schedule, never take longer, and gain the most on the responses that need the most rounds. Our method starts from the natural schedule and searches for tokens whose earlier reveal helps other positions become ready sooner.

4.2 Uncertainty-Guided Schedule Construction

The uncertainty-area objective in Eq. (4) motivates revealing tokens that make the remaining positions easier to predict early in the trajectory. Figure 3 summarizes our framework. Given a correct response y, we construct its unmasking trajectory offline with the frozen model p_{θ_0}. The response remains fixed, and we change only the order in which its tokens are revealed. At each round, we compare the candidate reveals by how much they reduce uncertainty at the other masked positions, as illustrated on the right of Figure 3. This gives a local selection rule based on the context a token provides, rather than only on how predictable that token already is.

At round r, $\mathcal{M}_r$ contains the masked positions in state s_r. We consider plausible target tokens among these positions,

$$\mathcal{C}_r = \{i \in \mathcal{M}_r \mid p_{\theta_0}(y_i \mid s_r, i) \geq p_{\min}, \quad \mathrm{rank}_{\theta_0}(y_i \mid s_r, i) \leq k\}\,. \tag{5}$$

Here $p_{\min}$ is a minimum target token probability and k is the maximum allowed rank among vocabulary tokens. For each candidate i, let $s_r^{(i)}$ be s_r with y_i revealed and all other positions unchanged. Using the per-position surprisal $h_j(s) = -\log p_{\theta_0}(y_j \mid s, j)$, we select the candidate that most reduces uncertainty at other positions,

$$i_r^* \in \arg\max_{i \in \mathcal{C}_r} \sum_{j \in \mathcal{M}_r \setminus \{i\}} \left[h_j(s_r) - h_j(s_r^{(i)})\right]. \tag{6}$$

The score measures the reduction in uncertainty at the other masked positions, excluding the revealed position i itself. We promote the candidate with the largest positive score. If no candidate has a positive score, we skip promotion. Repeating these steps until the response is fully revealed produces $\tau^*(y) = (s_0^*, \ldots, s_T^*)$, where s_r^* is the state after r rounds. This optimized trajectory is illustrated on the left of Figure 3. We pair its intermediate states s_r^* with the correct response y to train the student p_θ to predict the remaining masked tokens.

4.3 Schedule Distillation

We train the student on intermediate states s_r^* from the optimized trajectory $\tau^*(y)$. The correct response provides the token targets, while the schedule determines which tokens are visible in each

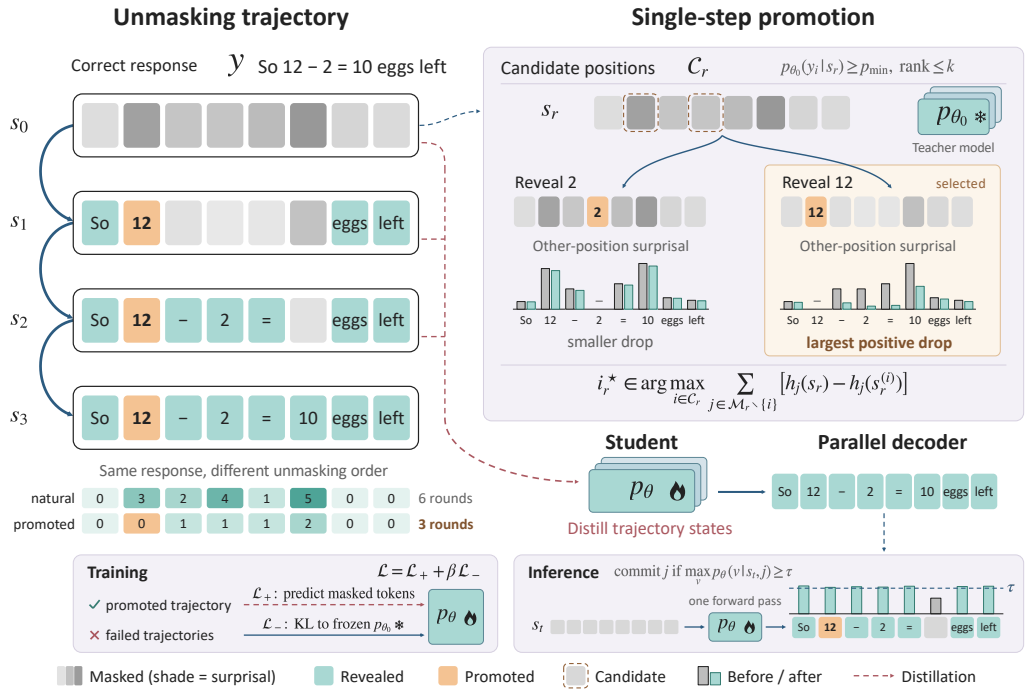


Figure 3: **Overview of dSpark.** *Left:* for a fixed correct response y, the frozen teacher p_{θ_0} builds an unmasking trajectory that reveals the same tokens in a new order. Masked cells are shaded by surprisal, with darker cells more uncertain. The grid below gives the round in which each token is revealed. Promoting "12" first makes the other tokens predictable sooner, so the response is complete after 3 rounds instead of 6. *Right: single-step promotion.* Each plausible candidate $i \in \mathcal{C}_r$ is revealed from the same state s_r. Gray and green bars show the surprisal at the other masked positions before and after the reveal. The candidate with the largest positive total drop (Eq. (6)) is promoted and committed with the ready positions. *Bottom:* the student is trained on promoted trajectories ($\mathcal{L}_+$) with KL to p_{θ_0} on failed ones ($\mathcal{L}_-$).

training example. The student, therefore, learns to predict the remaining tokens from contexts in which informative positions have been revealed earlier.

Let $\mathcal{D}_+$ denote the distribution of correct responses y and states s_r^* sampled from $\tau^*(y)$. We combine cross-entropy supervision with confidence sharpening. At each state s_r^*, $\mathcal{M}_r$ contains the eligible masked positions and $\mathcal{R}_r \subseteq \mathcal{M}_r$ those where the student's most likely token matches the target token y_j. The positive-trajectory objective is

$$\mathcal{L}_+(\theta) = \mathbb{E}_{(s_r^*, y)\sim\mathcal{D}_+}\left[\ell_{\mathrm{CE}}(s_r^*, y) + \ell_{\mathrm{sharp}}(s_r^*, y)\right], \tag{7}$$

where ℓ_{CE} trains the student to reconstruct the masked tokens of the correct response y, using cross-entropy averaged over $\mathcal{M}_r$. Training on intermediate states from our optimized trajectory encourages the student to follow its unmasking progression during decoding. The sharpening term ℓ_{sharp} averages the predictive entropy $\mathcal{H}(p_\theta(\cdot \mid s_r^*, j))$ over currently correct positions $j \in \mathcal{R}_r$, where $\mathcal{H}(q) = -\sum_{v\in\mathcal{V}} q(v)\log q(v)$. Minimizing this term strengthens confidence in correct predictions, supporting earlier commitments during decoding.

The sharpening term, however, cannot tell right from wrong on its own. It is applied only where the student is already correct on the constructed trajectories, but the parameters it updates are shared, so the added confidence carries over to states the student reaches on unsuccessful generation paths. On those states, more confidence can only hurt, so the student must stay where the base model was. We therefore retain failed trajectories for selective KL regularization. Response-level failure does not identify which individual tokens are incorrect, so we use the frozen base model's distribution as the target rather than the failed response's tokens. Let $\mathcal{D}_-$ denote the distribution of state–position pairs sampled for anchoring from these trajectories. The negative-trajectory loss is

$$\mathcal{L}_-(\theta) = \mathbb{E}_{(s^-, j)\sim\mathcal{D}_-}\left[D_{\mathrm{KL}}\left(p_{\theta_0}(\cdot \mid s^-, j) \,\|\, p_\theta(\cdot \mid s^-, j)\right)\right], \tag{8}$$

where both models receive the same state s^- and the base model remains frozen. This anchoring limits changes where reliable token-level supervision is unavailable, while leaving the constructed correct trajectories free of the KL penalty. The full training objective is

$$\mathcal{L}(\theta) = \mathcal{L}_+(\theta) + \beta\,\mathcal{L}_-(\theta), \tag{9}$$

where β controls the regularization strength. Together, these objectives encourage earlier confident commitments for faster parallel decoding while maintaining high task accuracy.

5 Experiments

5.1 Experimental Setup

Training Data. We build a separate distillation corpus for each backbone. For both models, we generate the trajectories from GSM8K (Cobbe et al., 2021), PRM12K (Lightman et al., 2024), NuminaMath-CoT (LI et al., 2024), and AceCode (Zeng et al., 2025) using a maximum length of 256 tokens, with a block size of 32. Overall, the training data contains 52k correct and 38k incorrect samples for LLaDA, and 80k correct and 57k incorrect for Dream. During training, we use all correct trajectories and subsample the incorrect ones to a 4:1 ratio, yielding 52k correct and 13k incorrect trajectories for LLaDA, and 80k correct and 20k incorrect for Dream.

Implementation Details. In our experiments we use LLaDA-Instruct (Nie et al., 2026) and Dream-Instruct (Ye et al., 2025) models. We fine-tune the models using LoRA (Hu et al., 2021). We use a multi-block decoding strategy similar to d3LLM (Qian et al., 2026). To create a set of target tokens in Eq. 5, we use $p_{min} = 0.02$ and $k = 10$. We set $\beta = 1$ in the training objective in Eq. 9. Training configurations and hardware details are provided in the Appendix.

Evaluation Details Our evaluation covers mathematical reasoning on GSM8K (0-shot) (Cobbe et al., 2021) and MATH (4-shot) (Hendrycks et al., 2021), and code generation on HumanEval (0-shot) (Chen et al., 2021) and MBPP (3-shot) (Austin et al., 2021b). We report accuracy for math and pass@1 for code, together with tokens per forward (TPF) and accuracy under parallelism (AUP) (Qian et al., 2026). We use `lm-eval-harness` (Gao et al., 2024).

Baselines. We evaluate our method against competitive dLLM baselines. These include the vanilla LLaDA (Nie et al., 2026) and Dream (Ye et al., 2025) models and the training-free Fast-dLLM (Wu et al., 2026b). We also compare with methods that fine-tune the model for faster decoding, namely dParallel (Chen et al., 2026b), d3LLM (Qian et al., 2026), and D2F (Wang et al., 2026).

5.2 Main Results

Table 1 shows that dSpark achieves the highest TPF and AUP across all four benchmarks on both backbones. The results demonstrate a consistent improvement in the accuracy–parallelism trade-off, with the relative contributions of accuracy retention and increased parallelism varying across tasks.

Results on LLaDA. On GSM8K, dSpark achieves $10.12\times$ the decoding parallelism of the base model while maintaining the base model's accuracy. Compared to d3LLM, it further increases parallelism while closing the accuracy gap relative to the base model, resulting in a 33.3% improvement in AUP. On MATH, dSpark similarly preserves base-model performance while achieving the highest TPF and AUP. The code benchmarks show that the benefits extend beyond mathematical reasoning. On HumanEval, dSpark matches d3LLM's pass@1 while decoding more tokens per forward pass. On MBPP, it improves both pass@1 and TPF over d3LLM. Across all four tasks, dSpark makes better use of parallel decoding than the competing methods, with the highest AUP in every case.

Results on Dream. On GSM8K, dSpark achieves $5.12\times$ the base model's TPF while improving accuracy by 1.5%. Relative to d3LLM, the larger parallelism gains occur on code generation: TPF increases by 30.4% on MBPP and 36.4% on HumanEval, accompanied by higher pass@1 on both benchmarks. These improvements give dSpark a substantial AUP lead on the two code tasks and show that its advantage extends across backbones. On MATH, the gain over d3LLM is smaller, but dSpark still achieves the highest TPF and AUP. On MBPP, dSpark achieves the highest TPF and AUP of all compared methods, while also improving pass@1 over d3LLM.

Accuracy–parallelism trade-off. Figure 4 shows why dSpark leads in AUP. Methods can decode more tokens per forward pass by accepting a sharp drop in accuracy. dSpark achieves higher TPF

Table 1: **Results by benchmark and backbone.** Accuracy, tokens per forward (TPF), and accuracy under parallelism (AUP) for LLaDA-8B-Instruct and Dream-7B-Instruct.

Benchmark	Method	LLaDA-8B-Instruct			Dream-7B-Instruct		
		Acc.↑	TPF↑	AUP↑	Acc.↑	TPF↑	AUP↑
GSM8K	Base model	78.8	1.00	78.80	82.1	1.00	82.10
	Fast-dLLM	75.5	2.91	199.24	80.0	1.52	122.01
	D2F	74.2	2.83	190.87	80.2	1.71	132.39
	dParallel	75.6	7.09	469.03	82.8	3.06	246.01
	d3LLM	74.2	9.05	571.28	80.7	4.97	379.09
	dSpark	**78.9**	**10.12**	**761.67**	**83.6**	**5.12**	**424.28**
MATH	Base model	31.7	1.00	31.70	**39.5**	1.00	39.50
	Fast-dLLM	30.8	1.91	46.24	38.4	1.71	53.87
	D2F	28.5	2.44	45.96	38.0	1.64	51.32
	dParallel	30.3	3.16	64.57	38.1	2.93	76.76
	d3LLM	30.3	5.75	107.59	38.6	4.07	101.47
	dSpark	**32.0**	**6.03**	**119.55**	38.9	**4.31**	**106.58**
MBPP	Base model	**41.7**	1.00	41.70	**57.0**	1.00	57.00
	Fast-dLLM	38.2	2.20	57.33	54.5	1.17	62.75
	D2F	38.0	1.98	50.70	53.8	1.31	67.52
	dParallel	40.2	2.27	59.46	56.2	2.35	113.76
	d3LLM	39.7	4.56	91.88	55.2	2.89	137.54
	dSpark	40.0	**5.52**	**104.94**	56.0	**3.77**	**171.43**
HumanEval	Base model	38.1	1.00	38.10	55.2	1.00	55.20
	Fast-dLLM	37.8	2.39	52.30	54.6	1.45	66.76
	D2F	38.2	2.71	60.59	54.0	1.39	66.78
	dParallel	39.0	4.95	83.88	54.3	2.30	91.46
	d3LLM	**39.6**	5.95	96.58	57.1	3.19	129.15
	dSpark	**39.6**	**6.73**	**106.19**	**57.3**	**4.35**	**167.17**

Table 2: Ablation study on different training recipes of dSpark on the LLaDA.

Schedule	KL Regularization	GSM8K			HumanEval		
		TPF ↑	Acc ↑	AUP ↑	TPF ↑	Acc ↑	AUP ↑
		7.6	74.9	463.4	5.2	37.9	82.2
✓		**11.4**	74.2	718.0	**8.3**	35.1	105.8
	✓	6.3	78.0	458.4	3.7	**40.2**	74.3
✓	✓	10.1	**78.9**	**761.7**	6.7	39.6	**106.2**

while maintaining strong accuracy. This is clearest on GSM8K, where it achieves both the highest TPF and the highest accuracy on LLaDA and Dream. On HumanEval, it exceeds d3LLM's TPF without losing pass@1 on LLaDA and improves both measures on Dream. The other plots show the same overall pattern: dSpark moves decoding toward greater parallelism while preserving enough task performance to make that gain useful. These results are consistent with the goal of dSpark, which encourages the model to reveal informative tokens earlier so that more of the remaining positions can be predicted together. The KL regularizer helps dSpark retain accuracy under aggressive parallel decoding, giving it an advantage over competing methods at higher TPF.

5.3 Ablation Study

Ablation Study on Training Recipe. To examine the contribution of each component, we ablate schedule distillation and KL regularization on LLaDA, with results shown in Table 2. Using ordinary trajectory training without optimized schedule and KL, it achieves the accuracy of 74.9 and TPF of 7.1 on GSM8K, and 37.9 accuracy and 5.2 TPF on Humaneval. Using the schedule alone substantially increases TPF on both GSM8K and HumanEval, but accuracy declines as the model commits more tokens in parallel. KL regularization alone has the opposite effect: it improves accuracy but reduces TPF, leading to lower AUP than the variant without either component. Combining the two gives the best balance. The full recipe retains 10.1 TPF on GSM8K and 6.7 TPF on HumanEval while raising accuracy to 78.9% and 39.6%, respectively. It achieves the highest AUP on both tasks. These results show that the schedule is responsible for the parallelism gain, while KL regularization helps prevent that gain from coming at the expense of accuracy.

Ablation study on KL regularization data.

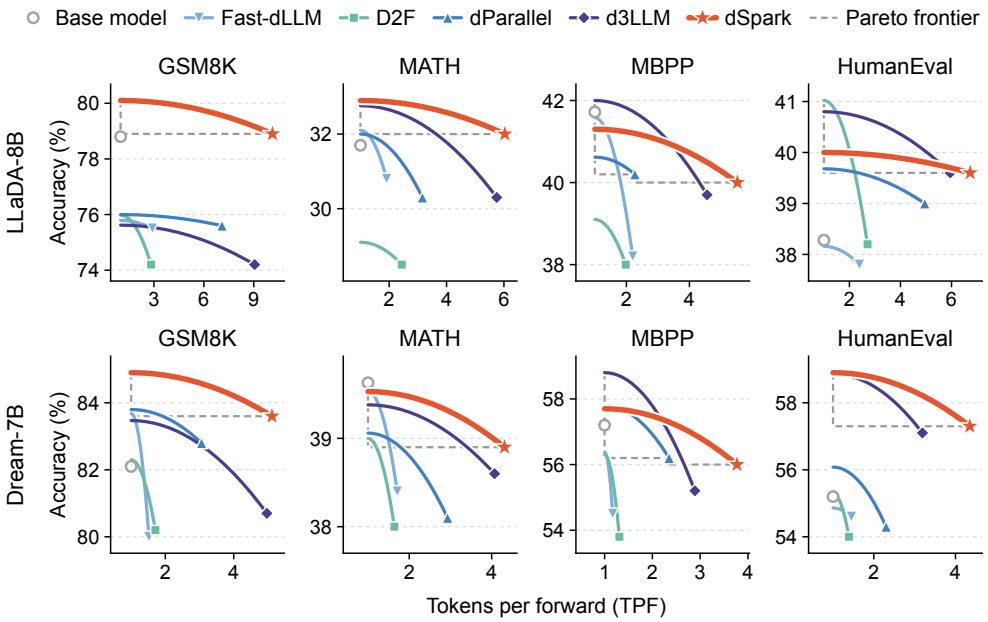


Figure 4: **dSpark extends the accuracy–parallelism frontier on every benchmark and backbone.** Accuracy versus TPF for LLaDA (top) and Dream (bottom). Markers are the operating points of Table 1. Each curve connects a method's accuracy at TPF $= 1$ to its operating point, the two points that make AUP. A flat curve means the method gains parallelism at little cost in accuracy, and a steep drop means it trades accuracy for speed. The dashed gray line is the Pareto frontier.

To validate our choice of training mixture, we ablate the ratio of correct to incorrect trajectories on LLaDA and evaluate the resulting models on GSM8K, as shown in Table 3. Without incorrect trajectories, the model reaches 11.4 TPF but only 74.19% accuracy. Adding incorrect samples improves accuracy, although a larger share generally reduces parallelism. We choose a 4:1 ratio because it retains 10.12 TPF while raising accuracy to 78.94%. Increasing the share further to 1:1 brings only a small gain in accuracy, while TPF falls to 7.90. At 2:1, both accuracy and TPF are lower than at 4:1, reaching 78.04% and 9.77, respectively. We also test using a separate, disjoint set of correct trajectories for KL regularization, keeping the same ratio of 4:1. Compared with using incorrect trajectories at the same ratio, accuracy falls from 78.94% to 77.10%, and TPF drops by 30.5%. Together, these results support using incorrect trajectories for KL regularization at a 4:1 ratio, which provides a strong balance between accuracy and parallel decoding.

Table 3: Accuracy and TPF on LLaDA GSM8K with different ratios of correct to incorrect training trajectories.

Data Ratio	Acc (%) ↑	TPF ↑
No inc. data	74.19	11.4
16 : 1	75.89	11.15
8 : 1	76.80	10.92
4 : 1	**78.94**	**10.12**
2 : 1	78.04	9.77
1 : 1	79.15	7.90

6 Conclusion

In this work, we studied efficient decoding for diffusion large language models from the perspective of *when to commit* predicted tokens. We showed that token confidence alone is insufficient for effective scheduling, since some less predictable tokens can provide more useful context for resolving the remaining sequence. We introduce **dSpark** enabled by *schedule distillation*, which constructs informative unmasking schedules based on the downstream utility of token commitments and distills these schedules into the model for efficient parallel inference. Experiments on mathematical reasoning and code generation show substantial acceleration, including 10.12× and 5.12× speedups on GSM8K for LLaDA-8B-Instruct and Dream-7B-Instruct, respectively, while maintaining strong task performance. Our results highlight unmasking schedules as an important learnable component of diffusion language modeling and suggest that better commitment strategies can substantially narrow the gap between the theoretical parallelism and practical efficiency of dLLMs.

Ethics Statement

This work focuses on improving the inference efficiency of diffusion large language models and does not involve human subjects or the collection of sensitive personal data. Our method does not introduce new model capabilities beyond those of the underlying pretrained models and therefore inherits their existing limitations and potential risks, including biased, harmful, or inaccurate generations. By reducing the number of decoding steps required during inference, the proposed approach can also help lower computational cost and energy consumption.

Reproducibility Statement

We provide detailed descriptions of the proposed method, training objectives, decoding procedures, experimental settings, and evaluation protocols in the main paper and supplementary material. We plan to release the implementation code and relevant configurations to facilitate reproducibility.

AI Use Statement

Generative AI tools were used only for language polishing and improving the clarity of the manuscript. All technical content, experimental design, analysis, and conclusions were developed and verified by the authors.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Marianne Arriola, Aaron Gokaslan, Justin Chiu, Zhihan Yang, Zhixuan Qi, Jiaqi Han, Subham Sahoo, and Volodymyr Kuleshov. Block diffusion: Interpolating between autoregressive and diffusion language models. In *International Conference on Learning Representations*, volume 2025, pp. 50726–50753, 2025.

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

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

Heli Ben-Hamu, Itai Gat, Daniel Severo, Niklas S Nolte, and Brian Karrer. Accelerated sampling from masked diffusion models via entropy bounded unmasking. *Advances in Neural Information Processing Systems*, 38:55981–56007, 2026.

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

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

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.

Zigeng Chen, Gongfan Fang, Xinyin Ma, Ruonan Yu, and Xinchao Wang. Dmax: Aggressive parallel decoding for dllms. *arXiv preprint arXiv:2604.08302*, 2026a.

Zigeng Chen, Gongfan Fang, Xinyin Ma, Ruonan Yu, and Xinchao Wang. dparallel: Learnable parallel decoding for dllms. In *International Conference on Learning Representations*, volume 2026, pp. 53193–53211, 2026b.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

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

Justin Deschenaux and Caglar Gulcehre. Beyond autoregression: Fast llms via self-distillation through time. In *International Conference on Learning Representations*, volume 2025, pp. 5007–5045, 2025.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. The language model evaluation harness, 07 2024. URL https://zenodo.org/records/12608602.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

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

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *NeurIPS*, 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. *arXiv preprint arXiv:2106.09685*, 2021.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.

Jia LI, Edward Beeching, Lewis Tunstall, Ben Lipkin, Roman Soletskyi, Shengyi Costa Huang, Kashif Rasul, Longhui Yu, Albert Jiang, Ziju Shen, Zihan Qin, Bin Dong, Li Zhou, Yann Fleureau, Guillaume Lample, and Stanislas Polu. Numinamath. [https://huggingface.co/AI-MO/NuminaMath-CoT](https://github.com/project-numina/aimo-progress-prize/blob/main/report/numina_dataset.pdf), 2024.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison 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*, volume 2024, pp. 39578–39601, 2024.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Aaron Lou, Chenlin Meng, and Stefano Ermon. Discrete diffusion modeling by estimating the ratios of the data distribution. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 32819–32848. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/lou24a.html.

Xinyin Ma, Runpeng Yu, Gongfan Fang, and Xinchao Wang. dkv-cache: The cache for diffusion language models. *Advances in Neural Information Processing Systems*, 38:149009–149033, 2026.

Yuxin Ma, Lun Du, Lanning Wei, Kun Chen, Qian Xu, Kangyu Wang, Guofeng Feng, Guoshan Lu, Lin Liu, Xiaojing Qi, et al. dinfer: An efficient inference framework for diffusion language models. *arXiv preprint arXiv:2510.08666*, 2025.

Quan Nguyen-Tri, Mukul Ranjan, and Zhiqiang Shen. Attention is all you need for kv cache in diffusion llms. In *International Conference on Learning Representations*, volume 2026, pp. 34915–34946, 2026.

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

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*, volume 2025, pp. 64972–65009, 2025.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35: 27730–27744, 2022.

Yu-Yang Qian, Junda Su, Lanxiang Hu, Peiyuan Zhang, Zhijie Deng, Peng Zhao, and Hao Zhang. d3llm: Ultra-fast diffusion llm using pseudo-trajectory distillation. *arXiv preprint arXiv:2601.07568*, 2026.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16, 2020. doi: 10.1109/SC41405.2020.00024.

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

Subham Sekhar Sahoo, Justin Deschenaux, Aaron Gokaslan, Guanghan Wang, Justin T Chiu, and Volodymyr Kuleshov. The diffusion duality. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 52584–52619. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/sahoo25a.html.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Xu Wang, Chenkai Xu, Yijie Jin, Jiachun Jin, Hao Zhang, Kai Yu, and Zhijie Deng. Diffusion llms can do faster-than-ar inference via discrete diffusion forcing. In *International Conference on Learning Representations*, volume 2026, pp. 2282–2299, 2026.

Chengyue Wu, Hao Zhang, Shuchen Xue, Shizhe Diao, Yonggan Fu, Zhijian Liu, Pavlo Molchanov, Ping Luo, Song Han, and Enze Xie. Fast-dllm v2: Efficient block-diffusion llm. In *International Conference on Learning Representations*, volume 2026, pp. 128353–128370, 2026a.

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*, volume 2026, pp. 57027–57051, 2026b.

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.

Yushi Ye, Xu Chen, Haoyun Jiang, Jinsong Lan, Haihong Tang, Bo Han, Ivor Tsang, Yanfeng Wang, Bo Zheng, and Jiangchao Yao. Ripple-pivot search: Active parallel decoding for diffusion large language models. *arXiv preprint arXiv:2608.11742*, 2026.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. ACE-CODER: Acing coder RL via automated test-case synthesis. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12023–12040, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.587. URL https://aclanthology.org/2025.acl-long.587/.

Haohui Zhang, Zhiye Wang, Xiaoying Gan, Xinbing Wang, and Bo Jiang. Leap: Unlocking dllm parallelism via lookahead early-convergence token detection. *arXiv preprint arXiv:2605.10980*, 2026.

Fengqi Zhu, Rongzhen Wang, Shen Nie, Xiaolu Zhang, Chunwei Wu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Llada 1.5: Variance-reduced preference optimization for large language diffusion models. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 11425–11460, 2026.

APPENDIX

A ADDITIONAL RESULTS

A.1 WALL-CLOCK THROUGHPUT AND KV CACHING

TPF counts forward passes, but a forward pass with a longer context costs more time. We measure wall-clock throughput on NVIDIA A100-SXM4 GPUs with batch size 1. Table 4 therefore reports the wall-clock throughput of dSpark on LLaDA, without a KV cache and with the prefix and dual caches of Fast-dLLM (Wu et al., 2026b). dSpark runs with both caches. With the short 0-shot prompt, the caches barely change throughput (253 and 213 tokens/s, against 249 without a cache), since they also increase the number of forward passes (from 24.8 to 29.4 with the prefix cache). With a 5-shot prompt, the prefix cache raises throughput from 92 to 155 tokens/s ($1.7\times$) at similar accuracy. At 512 tokens, the caches lower accuracy by 1.1–2.7 points without raising throughput. Caching is therefore most useful for long prompts, and dSpark reaches 249 tokens/s without it.

We also implement a static KV cache that runs every decoding step as a captured CUDA graph. It keeps the keys and values of the prompt and of committed tokens and, at each step, recomputes only the active window, the block behind it and a rotating slice of 64 cached positions, so that every cached position is refreshed periodically. With this cache, dSpark generates 313 tokens/s at 79.2% accuracy with the 0-shot prompt, 260 tokens/s with the 5-shot prompt and 300 tokens/s at 512 tokens, 1.3–$2.8\times$ its throughput without a cache, with accuracy within 1.3 points of it (Table 4).

Table 4: **Wall-clock throughput of dSpark with KV caching** (LLaDA-8B-Instruct, GSM8K, batch size 1). *Prefix* and *Dual* are the caches of Fast-dLLM (Wu et al., 2026b), and *Static* is our CUDA-graph cache. TPS: tokens per second.

Setting	KV cache	Acc.↑	TPF↑	TPS↑
0-shot, 256 tokens	None	78.94	10.12	249
	Prefix	77.86	8.52	253
	Dual	78.17	7.94	213
	Static (ours)	79.23	9.51	**313**
5-shot, 256 tokens	None	79.68	9.60	92
	Prefix	80.29	8.07	155
	Dual	80.97	7.45	142
	Static (ours)	79.08	8.53	**260**
0-shot, 512 tokens	None	84.00	11.69	168
	Prefix	81.35	7.94	153
	Dual	82.94	7.28	160
	Static (ours)	82.71	9.86	**300**

Table 5 shows that dSpark's gains in decoding parallelism translate into faster inference at batch size 1. On GSM8K, it achieves 249 tokens per second (TPS), a $16.6\times$ improvement over the base LLaDA model and 13.2% higher throughput than d3LLM. It is also the most accurate method in the table, scoring 78.94% compared with 78.82% for LLaDA and 74.21% for d3LLM. Thus, dSpark's higher TPF corresponds to a wall-clock speedup while preserving the backbone's task performance.

Table 5: Accuracy and wall-clock throughput against the baselines on GSM8K for LLaDA models with batch size 1.

Model	Acc.↑	TPF↑	TPS↑
LLaDA	78.82	1.00	15
Fast-dLLM	75.51	2.91	94
D2F	74.24	2.83	89
dParallel	75.62	7.12	165
d3LLM	74.21	9.05	220
dSpark	**78.94**	**10.12**	**249**

A.2 Generation Length

Table 6: **The gains of dSpark hold across generation lengths.** Accuracy (%) and TPF on LLaDA for maximum generation length L. Bold: best in each column.

		$L = 128$		$L = 256$		$L = 512$	
Benchmark	**Method**	Acc.↑	TPF↑	Acc.↑	TPF↑	Acc.↑	TPF↑
GSM8K	LLaDA	72.86	1.00	78.82	1.00	82.26	1.00
	dParallel	69.53	6.28	75.62	7.09	80.89	8.66
	d3LLM	68.06	8.31	74.21	9.05	78.92	10.73
	dSpark	**73.31**	**8.85**	**78.94**	**10.12**	**84.00**	**11.69**
MATH	LLaDA	**30.86**	1.00	31.74	1.00	34.26	1.00
	dParallel	27.48	2.78	30.31	3.16	30.36	3.39
	d3LLM	28.62	4.93	30.34	5.75	30.46	6.28
	dSpark	30.50	**4.96**	**32.03**	**6.03**	**35.64**	**6.79**

Effect of Generation Length. Table 6 compares performance at maximum generation lengths of 128, 256, and 512 tokens. dSpark achieves the highest TPF in every setting and the highest accuracy in five of six settings. On GSM8K, it matches or exceeds the base LLaDA model's accuracy at every length while increasing TPF from 8.85 at $L = 128$ to 11.69 at $L = 512$. On MATH, dSpark is more accurate than both accelerated baselines at all three lengths and reaches the highest accuracy overall at $L = 256$ and $L = 512$. Its advantage is especially clear at $L = 512$, where it achieves 35.64% accuracy and 6.79 TPF, compared with 30.46% and 6.28 TPF for d3LLM. These results show that dSpark's gains in parallelism persist as the generation length increases, while accuracy remains strong across both tasks.

A.3 General Knowledge

General Knowledge. We evaluate dSpark on MMLU (Hendrycks et al., 2020) and ARC-C (Clark et al., 2018) to check whether training for faster decoding affects performance beyond math and code. As shown in Table 7, dSpark matches the base LLaDA model's 63.1% accuracy on MMLU and scores 47.2% on ARC-C, which is only 0.4 drop from base model. This suggests that dSpark largely preserves the backbone's general knowledge performance.

Table 7: **General knowledge is preserved.** Accuracy (%) of LLaDA-8B-Instruct before and after dSpark training.

	LLaDA	dSpark
MMLU	63.0	63.1
ARC-C	47.6	47.2

A.4 Confidence and False Positives

Distillation for parallel decoding makes a model confident enough to commit many tokens per forward pass. The risk is that it also becomes confident about wrong tokens, which a threshold decoder then commits. Table 8 measures this on the natural schedules of 250 test GSM8K samples, recording at every round the prediction at every masked position. The three distilled models are confident about a similar share of correct predictions (74–76% at $p \geq 0.9$, against 52% for the base model). Among wrong predictions, however, d3LLM and dParallel are confident for 22.9% and 21.7%, while dSpark is confident for 15.5%, so its confident predictions are correct 91% of the time, against 82% for the two baselines.

Figure 5 breaks these predictions into confidence bins. The base model is well calibrated: in every bin, its accuracy is close to its confidence. dParallel and d3LLM are overconfident at every confidence level; for example, their predictions made with a confidence between 0.8 and 0.9 are correct only 51% and 52% of the time. dSpark stays much closer to the diagonal, at 65% in the same bin, although it places slightly more predictions in the top bin than the two baselines (56%, against

Table 8: **dSpark is less often confident when it is wrong.** Top-1 predictions at every masked position and decoding round along the natural schedules of 250 GSM8K samples.

	Base model	dParallel	d3LLM	dSpark
Correct predictions with $p \geq 0.9$ (%)↑	51.6	74.4	74.3	76.1
Wrong predictions with $p \geq 0.9$ (%)↓	2.5	21.7	22.9	15.5
Predictions with $p \geq 0.9$ that are correct (%)↑	97.6	81.7	82.2	90.8
Expected calibration error (%)↓	4.6	19.2	20.4	8.2

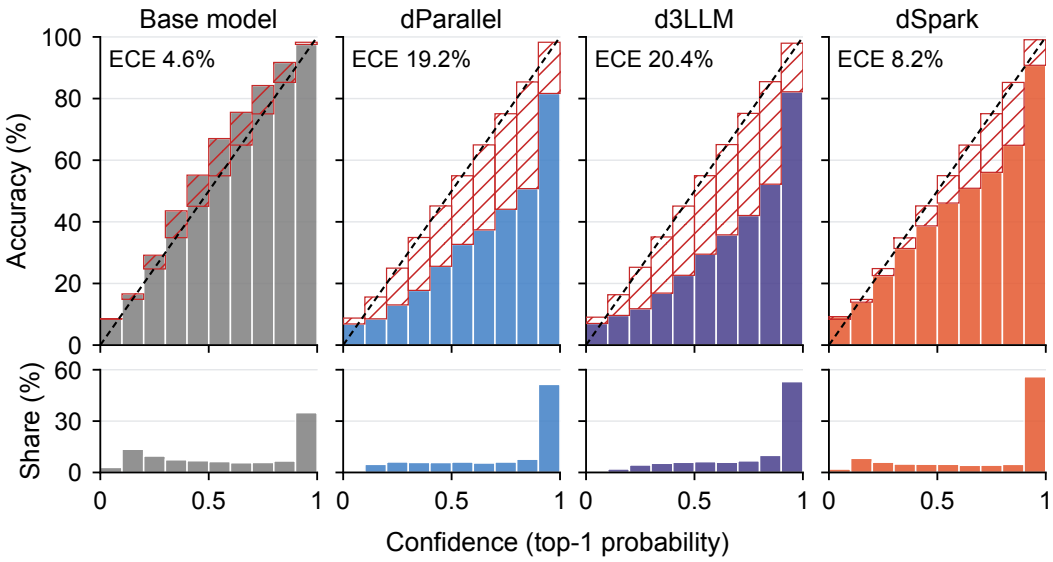


Figure 5: **dSpark stays much closer to calibration than the baselines it is compared with.** Reliability diagrams for the predictions of Table 8 in 10 equal-width confidence bins. *Top:* accuracy per bin. The dashed diagonal marks perfect calibration, and the red hatched area is the gap between confidence and accuracy. *Bottom:* share of predictions in each bin.

52% and 53%). Its expected calibration error is also substantially lower than theirs (8.2% versus 19.2% and 20.4%). This more reliable confidence helps explain why dSpark can commit tokens aggressively without the same loss of accuracy.

B Training Details

We fine-tune both backbones with LoRA (Hu et al., 2021) on all attention and feed-forward projections, keeping the pretrained weights frozen, and merge the adapters into the backbone before evaluation. Table 9 lists the configuration. The two backbones share every optimization setting and differ only in their training corpora. We train each model on four NVIDIA A100-SXM4 GPUs with data parallelism under DeepSpeed ZeRO stage 2 (Rajbhandari et al., 2020), without CPU offloading.

C Evaluation Details

Decoding for dSpark. We use a multi-block decoding strategy with a confidence threshold of 0.90 to 0.95, which enables decoding both the current and next blocks in parallel. All models decode greedily with at most 256 new tokens and a block size of 32. A new block enters the window once the last active block is 50% decoded and becomes fully active once its predecessor is 50% decoded. We use no KV cache in the main results.

Metrics. Tokens per forward (TPF) is the number of generated tokens, excluding end-of-sequence padding, divided by the number of forward passes, both summed over the benchmark. Accuracy under parallelism (AUP) (Qian et al., 2026) integrates accuracy over parallelism. For points

Table 9: Training configuration of dSpark.

	LLaDA-8B-Instruct	**Dream-7B-Instruct**
LoRA rank r / α	256 / 256	
LoRA target modules	`q_proj, k_proj, v_proj, attn_out, ff_proj, up_proj, ff_out`	`q_proj, k_proj, v_proj, o_proj, gate_proj, up_proj, down_proj`
Optimizer	AdamW (Loshchilov & Hutter, 2017)	
Learning rate (schedule)	2×10^{-5} (constant)	
Weight decay / gradient clipping	0.01 / 1.0	
Epochs	6	
Batch size	64	
Max. sequence length	512	
KL weight β (Eq. 9)	1	
Precision	bfloat16	

$(\rho_1, y_1), \ldots, (\rho_n, y_n)$ sorted by TPF ρ, with accuracy y, and keeping only points with $y_i \geq y_1 - 5$,

$$\mathrm{AUP} = \rho_1 y_1 + \sum_{i=2}^{n} \frac{\rho_i - \rho_{i-1}}{2}\big(y_i W(y_i) + y_{i-1} W(y_{i-1})\big), \qquad W(y) = \min\Big(e^{-\alpha(1-y/y_{\max})}, 1\Big), \tag{10}$$

where $y_{\max}$ is the highest accuracy of any method on the benchmark and $\alpha = 3$. Each method contributes its one-token-per-forward point ($\rho = 1$) and its reported operating point.

D Additional Ablations

D.1 Candidate Filter

A promoted token is committed before the frozen model would commit it on its own, so it is, by construction, a token the model is not yet confident about. The filter in Eq. (5) therefore does not ask for confidence. It only asks that the reference token be *plausible* to the model: among its $k = 10$ most likely tokens at that position, with a probability of at least $p_{\min} = 0.02$, i.e., a surprisal of at most about 4 nats, the range shown in Figure 1. The two bounds complement each other. When the model's distribution is flat, many tokens exceed 2% and the rank bound applies; when it is peaked, the tenth-ranked token can have a negligible probability and the probability bound applies.

Why not a stricter filter. The tokens that help the other positions most are rarely the confident ones. As shown in Figure 1, revealing an uncertain token ($p_{\theta_0} < 0.5$) makes $5.1\times$ more other positions ready than revealing a confident one and lowers their surprisal $4.7\times$ more. A filter that asks for confidence would therefore exclude the most useful promotions. Promoting these low-probability tokens also does not cost accuracy (Table 10).

Why not a looser filter. Below these bounds, the model has effectively ruled the reference token out given the current context. Promoting such a token would make the schedule depend on knowing the verified answer rather than on the model's own uncertainty, and would train the student to commit a token that its context does not support. The bounds also keep the search small, since every candidate is scored with its own forward pass in Eq. (6). We set $p_{\min}$ and k from these considerations and did not tune them in end-to-end training.

To test whether low-probability promotions cost accuracy, we take 240 states at which the schedule promotes a token, 80 per probability range, and finish decoding either by committing the promoted token or by committing nothing. Table 10 shows that committing the promoted token leaves accuracy unchanged within sampling noise while reducing the forward passes needed to finish, by 6.5 for the least likely tokens.

D.2 Token Promotion Strategy.

We compare the proposed surprisal-reduction criterion 6 with the natural schedule, random promotion of a plausible token, and promotion of the highest-surprisal token on GSM8K as shown in

Table 10: **Committing a low-probability promoted token keeps accuracy and saves forward passes.** From 240 states where our schedule promotes a token (80 per probability range), we finish decoding with (*Promoted*) or without (*Not promoted*) committing it.

	Accuracy (%)		Forward passes to finish	
Probability of promoted token	Promoted	Not promoted	Promoted	Not promoted
$[0.02, 0.1)$	80.0	80.0	31.6	38.1
$[0.1, 0.3)$	85.0	86.3	24.8	28.6
$[0.3, 1]$	86.3	83.8	25.1	27.7

Table 11. Without KL regularization, our criterion reaches 11.4 TPF, compared with 7.6 for the natural schedule and 8.3 for random promotion. Promoting the highest-surprisal token reaches 10.1 TPF but lowers accuracy to 73.3%. With KL regularization, our criterion achieves the highest TPF and accuracy, at 10.1 and 78.9%, respectively. It improves both measures over highest-surprisal promotion (8.9 TPF and 77.8% accuracy) and raises TPF substantially over random promotion (7.5) while maintaining similar accuracy. These results support selecting tokens for how much they simplify the remaining predictions, rather than for their own surprisal alone.

Table 11: Effect of the token promotion strategy on LLaDA GSM8K, with and without KL regularization. Accuracy is reported as a percentage.

	Without KL		With KL	
Promotion strategy	TPF↑	Acc.↑	TPF↑	Acc.↑
Natural schedule	7.6	74.9	6.3	78.0
Random plausible token	8.3	**75.1**	7.5	78.7
Highest-surprisal token	10.1	73.3	8.9	77.8
Surprisal reduction (Eq. 6)	**11.4**	74.2	**10.1**	**78.9**

D.3 KL Weight

Sensitivity to the KL Weight. We vary the KL regularization weight β on LLaDA and report the results in Table 12. Without KL regularization, the model reaches the highest TPF but has the lowest accuracy on both GSM8K and MATH-500. Increasing β to 1 raises accuracy from 74.19% to 78.94% on GSM8K and from 32.81% to 34.40% on MATH-500, while retaining 9.8 and 4.96 TPF, respectively. Raising β further to 2 adds only 0.16 accuracy on GSM8K while reducing TPF from 10.12 to 8.1. On MATH-500, accuracy remains unchanged and TPF also falls. We therefore use $\beta = 1$ for its stronger balance between accuracy and parallel decoding.

Table 12: Effect of the KL weight β on LLaDA accuracy and TPF for GSM8K and MATH-500. **Bold** indicates the selected setting.

	GSM8K		MATH-500	
$\boldsymbol{\beta}$	**Acc (%)↑**	**TPF↑**	**Acc (%)↑**	**TPF↑**
0	74.19	11.4	32.81	6.03
0.5	75.74	10.8	33.22	5.36
1	**78.94**	**10.12**	**34.40**	**4.96**
2	79.10	8.1	34.40	4.61

E Qualitative Example

In the following, we present the example of generation from our model and different baselines for LLaDA.

Prompt: *A raspberry bush has 6 clusters of 20 fruit each and 67 individual fruit scattered across the bush. How many raspberries are there total?* Reference answer: 187

Base model (LLaDA-8B-Instruct)

To determine the total number of raspberries on the bush, we need to add the number of raspberries in the clusters to the number of individual raspberries scattered across the bush. Let's break this down step by step: 1. Calculate the number of raspberries in the clusters: – There are 6 clusters. – Each cluster contains 20 raspberries. – Therefore, the total number of raspberries in the clusters is: $6 \times 20 = 120$. 2. Add the number of individual raspberries scattered across the bush: – There are 67 individual raspberries scattered across the bush. – Therefore, the total number of raspberries is: $120 + 67 = 187$. So, the total number of raspberries on the bush is $\boxed{187}$.

Forward passes: 199 Tokens: 199 Correct

dParallel

To determine the total number of raspberries on the bush, we need to add the number of raspberries in the clusters and the number of individual raspberries scattered across the bush. 1. First, calculate the number of raspberries in the clusters: – There are 6 clusters, and each cluster contains 20 raspberries. – Therefore, the number of raspberries in the clusters is: $6 \times 20 = 120$. 2. Next, add the number of individual raspberries scattered across the bush: – There are 67 individual raspberries scattered across the bush. – Therefore, the total number of raspberries is: $120 + 67 = 187$. Thus, the total number of raspberries on the bush is $\boxed{187}$.

Forward passes: 25 Tokens: 188 Correct

d3LLM

To determine the total number of raspberries on the bush, we need to calculate the number of raspberries in the clusters and the number of individual raspberries separately, and then sum these amounts. First, let's calculate the number of raspberries in the clusters: – There are 6 clusters. – Each cluster contains 20 raspberries. – Therefore, the total number of raspberries in the clusters is: 6 clusters $\times$ 20 raspberries per cluster $=$ 120 raspberries. Next, we add the number of individual raspberries: – There are 67 individual raspberries scattered across the bush. – Therefore, the total number of raspberries is: 120 raspberries $+$ 67 raspberries $=$ 187 raspberries. Thus, the total number of raspberries on the bush is $\boxed{187}$.

Forward passes: 23 Tokens: 216 Correct

dSpark (ours)

To determine the total number of raspberries on the bush, we need to follow these steps: 1. Calculate the number of raspberries in the clusters. 2. Add the number of individual raspberries to the number of raspberries in the clusters. First, let's calculate the number of raspberries in the clusters: – There are 6 clusters. – Each cluster contains 20 raspberries. – Therefore, the total number of raspberries in the clusters is: $6 \times 20 = 120$. Next, we add the number of individual raspberries to the number of raspberries in the clusters: – There are 67 individual raspberries. – Therefore, the total number of raspberries is: $120 + 67 = 187$. Thus, the total number of raspberries on the bush is $\boxed{187}$.

Forward passes: 17 Tokens: 202 Correct

Prompt: *John wins an award at work. The award has a 1 time monetary reward of $5000 and a 5% raise in salary. If he makes 2000 a week, every week of the year and he got the award on the first day of the year how much money did he make that year?* Reference answer: 114,200

Base model (LLaDA-8B-Instruct)

To determine how much money John made that year, we need to consider both his one-time monetary reward and his annual salary after the 5% raise. 1. One-Time Monetary Reward: John receives a one-time monetary reward of $5000. 2. Annual Salary After Raise: John's weekly salary is $2000. A 5% raise in his salary means he will earn: $2000 \times 0.05 = 1000$ dollars per week. Therefore, his annual salary is: $2000 \times 52 = 104000$ dollars. 3. Total Earnings for the Year: Adding the one-time monetary reward to his annual salary, we get: $5000+104000 = 109000$ dollars. Thus, the total amount of money John made that year is $\boxed{109000}$.

Forward passes: 256 Tokens: 255 Wrong

dParallel

To determine how much money John made in the year, we need to consider both the one-time monetary reward and the percentage increase in his salary. 1. One-time Monetary Reward: John receives a one-time monetary reward of $5000. 2. Percentage Increase in Salary: John receives a 5% raise in his salary. His original salary is $2000 per week. 3. Total Annual Salary: There are 52 weeks in a year. Therefore, his annual salary is: $2000 \times 52 = 104000$. 4. Total Earnings for the Year: Adding the one-time monetary reward to his total annual salary: $5000 + 104000 = 109000$. Thus, the total amount of money John made that year is $\boxed{109000}$.

Forward passes: 45 Tokens: 225 Wrong

d3LLM

To determine how much money John made that year, we need to break down his earnings into two parts: the one-time monetary reward and his increased salary. 1. One-Time Monetary Reward: John receives a one-time monetary reward of $5000. 2. Increased Salary: John receives a 5% raise in his salary. His original salary is $2000 per week. – Calculate the raise amount: $2000 \times 0.05 = 1000$ – Add the raise to his original salary: $2000 + 1000 = 3000$ – Calculate the total salary for the year: $3000 \times 52 = 156000$ 3. Total Earnings for the Year: – Add the one-time monetary reward to the total salary for the year: $5000 + 16000 = 161000$. Therefore, the total amount of money John made that year is `\boxed{161000` [cut off at 256 tokens]

Forward passes: 46 Tokens: 256 Wrong

dSpark (ours)

To determine how much money John made that year, we need to break down the problem into two parts: the one-time monetary reward and the 5% raise in his salary. 1. One-Time Monetary Reward: John receives a one-time monetary reward of $5000. 2. 5% Raise in Salary: John's original salary is $2000 per week. A 5% raise in his salary is: $0.05 \times 2000 = 100$. Therefore, his new salary is: $2000 + 100 = 2100$. There are 52 weeks in a year, so his annual salary is: $2100 \times 52 = 109200$. Now, we add the one-time monetary reward to his annual salary to find the total amount he made: $5000 + 109200 = \boxed{114200}$.

Forward passes: 28 Tokens: 255 Correct