FLEXDLLM: DIFFUSE WHEN POSSIBLE, AUTOREGRESS WHEN NECESSARY

Anonymous authors
Paper under double-blind review

ABSTRACT

Modern Large Language Model (LLMs) inference paradigms face a fundamental quality-efficiency trade-off: autoregressive (AR) decoding achieves strong generation quality but is inherently sequential, while diffusion-based LLMs (dLLMs) enable parallel decoding but often suffer from the degenerated performance. To overcome this dilemma, in this paper, we propose **FlexDLLM, a flexible hybrid inference framework that adaptively switches between diffusion decoding and AR refinement.** FlexDLLM is built on our key observation that inference errors in dLLMs typically stem from a few high-coupling tokens that are poorly suited for simultaneous updates. By applying AR refinement exclusively to these critical tokens, we can significantly boost performance while preserving the efficiency of highly parallel generation. Based on this insight, during inference, FlexDLLM dynamically alternates between two modes: a diffusion mode that efficiently decodes in parallel while autonomously identifying high-risk positions, and an AR mode that revises these difficult tokens conditioned on the global denoising state. Through this flexible switching mechanism, FlexDLLM strategically allocates sequential computation strictly to difficult, failure-prone spans. Experimental results demonstrate that FlexDLLM successfully marries the performance of AR with the speed of diffusion. It improves accuracy over the original LLaDA-8B-Instruct by 30.4% on GSM8K and 20.2% on HumanEval, while optimizing the accuracy-latency trade-off with a $4.1\times$ speedup on an H200 at comparable accuracy.

1 INTRODUCTION

While the autoregressive (AR) bottleneck of Large Language Models (LLMs) makes accelerating lengthy reasoning trajectories an urgent priority (Brown et al., 2020; Chen et al., 2021; Kasneci et al., 2023; Romera-Paredes et al., 2024; Leviathan et al., 2023; Wei et al., 2022; Snell et al., 2024), diffusion language models (dLLMs) address this via parallel decoding to generate multiple tokens simultaneously (Austin et al., 2021a; Li et al., 2022; Sahoo et al., 2024; Nie et al., 2025; Ye et al., 2025). However, a fundamental speed-quality trade-off arises because dLLMs rely on a conditional independence assumption, which restricts their ability to capture complex inter-token dependencies or correct premature denoising errors (Wu et al., 2025a; Kang et al., 2025; Nie et al., 2025). Consequently, high parallelism severely degrades performance, as demonstrated when LLaDA-8B-Instruct generates 4 tokens per step on GSM8K and achieves only 49.2

Recognizing this fundamental trade-off, recent approaches have attempted to hybridize AR and diffusion inference to capture the best of both worlds. For instance, Block Diffusion explicitly models AR dependencies between text blocks while utilizing diffusion to decode within each block (Arriola et al., 2025). Similarly, Planned Diffusion partitions the output into independent chunks via AR planning before applying diffusion denoising (Israel et al., 2025). However, as these methods merely alternate between the two paradigms at the chunk level, the tokens generated within each diffusion block still lack strict AR guidance. Consequently, while these methods are effective at small block sizes, their performance degrades significantly as the block size increases (Kang et al., 2025). For example, on LLaDA-8B-Base with a block size of 8, Block Diffusion scores only 27.4% on HumanEval, underperforming even pure diffusion (Chen et al., 2021). The core challenge thus remains: ***How can we optimally integrate AR and diffusion paradigms to maximize both AR-level accuracy and diffusion-level parallel efficiency?***

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

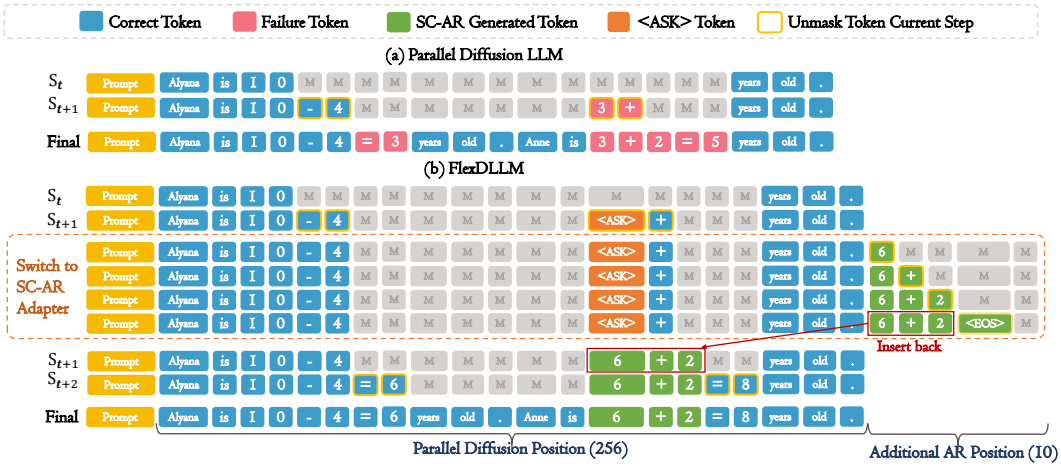

Figure 1: **FlexDLLM vs. Parallel dLLM.** Top. Parallel dLLM: mutually dependent tokens violate independence assumptions, causing cascading errors. Bottom. FlexDLLM switches modes: RG-Diff decodes in parallel and emits <ASK> at high-risk positions; SC-AR appends a short workspace, autoregressively generates an <EOS>-terminated refinement span, and overwrites the failure span.

To tackle this challenge, we first analyze failure tokens that lead dLLM decoding toward an incorrect final answer. By quantifying token dependencies during inference, we find that these failure tokens often occur at high-coupling positions: that are both shaped by and actively shape the other tokens generated in the same step. In such highly correlated positions, the same-step independence approximation assumed by parallel diffusion does not hold, making them poorly suited for simultaneous diffusion updates. To better handle these failure tokens, we propose *State-Conditioned Autoregressive Refinement* (SC-AR), which autoregressively revises difficult tokens conditioned on the global denoising decoding state rather than just the left prefix. By comparing the performance benefits of refining different tokens, we find that modifying merely a small fraction of failure tokens yields massive improvements, which cannot be achieved by refining other tokens. This insight implies a highly efficient strategy: **by delegating only a minimal set of critical tokens to the AR process, we can rapidly decode the vast majority of tokens in parallel without sacrificing performance.**

Motivated by these findings, we propose **FlexDLLM**, a flexible hybrid inference framework that adaptively switches between diffusion decoding and autoregressive refinement. As shown in Figure 1, FlexDLLM contains two switchable modes, implemented by two LoRA adapters on the same dLLM backbone. The first mode, Refinement-Gated Diffusion (*RG-Diff*) mode, performs parallel diffusion decoding for most tokens, but is trained to emit a special token <ASK> when it detects a high-risk position that may lead to a failure. The second mode, *SC-AR* mode: Once <ASK> is detected, the model conditions on the current global denoising state and enters refinement process, where a short, targeted replacement span is autoregressively generated for the <ASK> position until the process autonomously halts. The refined span is then written back into the diffusion state, and decoding switches back to RG-Diff mode. In this way, FlexDLLM spends autoregressive computation only on difficult failure-prone spans, while preserving fast parallel diffusion for the majority of tokens.

Extensive experiments demonstrate that FlexDLLM achieves superior performance across six benchmarks (e.g., +30.4% on GSM8K, +20.2% on HumanEval over LLaDA-8B-Instruct). Through targeted AR refinement, it also maintains high precision even at 8-token parallel decoding, establishing an exceptional accuracy-latency trade-off. Our contributions are summarized as follows:

- We propose State-Conditioned Autoregressive Refinement (**SC-AR**) to resolve parallel diffusion failure tokens by revising difficult positions conditioned on global denoising state.
- We design Refinement-Gated Diffusion (**RG-Diff**), an adaptive parallel inference mechanism autonomously identifies high-risk, failure-prone positions to trigger targeted AR intervention.
- We introduce **FlexDLLM**, an hybrid inference framework dynamically switching between RG-Diff and SC-AR. Experiments show that FlexDLLM improves both accuracy and efficiency, achieving a strong accuracy-latency Pareto frontier over existing baselines.

2 MOTIVATION: NOT ALL TOKENS ARE EQUALLY DIFFUSIBLE

2.1 NOTATION AND BACKGROUND

Notation. Let x denote the input prompt and $y = (y_1, \ldots, y_n)$ denote the target sequence. At diffusion step t, a dLLM maintains a partially denoised sequence state $s_t = (s_{t,1}, \ldots, s_{t,n})$, where each position contains either a visible token or a mask token. For a given position i, dLLM parameterized by θ_D, predicts a token-wise conditional distribution $p_{\theta_D}(Y_i \mid x, s_t)$. We use U_t to denote the set of positions to be updated at step t. For any two positions i and j, we denote the remaining context by $C_{ij,t} = (x, s_t \setminus \{i, j\})$, where $s_t \setminus \{i, j\}$ represents the state s_t with the tokens at positions i and j forced to be mask tokens while keeping all other existing visible and mask tokens in s_t unchanged.

Parallel diffusion as a same-step independence approximation. In parallel diffusion decoding, multiple positions are updated simultaneously from the same pre-update state s_t. This is typically implemented by computing token-wise conditional distributions independently and committing them together. Assume $\widetilde{p}_{\theta_D}$ denote the approximated joint distribution, which factorizes as,

$$\widetilde{p}_{\theta_D}(Y_{U_t} \mid x, s_t) = \prod_{i \in U_t} p_{\theta_D}(Y_i \mid x, s_t). \tag{1}$$

This product form implies that the simultaneously generated tokens are not explicitly conditioned on each other within the update step. For a pair of concurrently updated positions $i, j \in U_t$, this parallel approximation is only reliable when their same-step conditional dependence is negligible,

$$p_{\theta_D}(Y_i, Y_j \mid C_{ij,t}) \approx p_{\theta_D}(Y_i \mid C_{ij,t}) p_{\theta_D}(Y_j \mid C_{ij,t}), \quad p_{\theta_D}(Y_i \mid C_{ij,t}, Y_j) \approx p_{\theta_D}(Y_i \mid C_{ij,t}). \tag{2}$$

When this independence assumption fails, parallel diffusion suffers from a joint–marginal mismatch: although each token may be locally plausible under its marginal distribution, jointly sampled tokens are globally inconsistent (Kang et al., 2025; Wu et al., 2025b; Kim et al., 2026; Ringel et al., 2026).

2.2 SAME-STEP DEPENDENCY PROFILE OF FAILURE TOKENS.

To understand dLLM errors, we analyze **failure tokens** that drive decoding away from correct answers. We focus on earliest unmasked failures, which may propagate errors to subsequent tokens.

Dependency Measurement. To characterize these tokens, we propose two distributional metrics based on Kullback-Leibler (KL) divergence. First, ***In-KL*** quantifies a token's contextual sensitivity to other tokens generated concurrently in the same diffusion step. Let U_t denote the set of tokens updated at step t, and $|U_t|$ be the number of such tokens. For a specific token $i \in U_t$, we evaluate its dependency on a concurrent token $j \in U_t \setminus \{i\}$ by computing the KL divergence between the predictive distributions of Y_i when token j is masked and unmasked. Given the input prompt x and the current sequence state s_t, In-KL averages this divergence over all other concurrent tokens,

$$\mathrm{InKL}_t(i) = \frac{1}{|U_t| - 1} \sum_{j \in U_t \setminus \{i\}} D_{\mathrm{KL}}\left(p_{\theta_D}(Y_i \mid x, s_t \backslash \{i\}) \,\|\, p_{\theta_D}(Y_i \mid x, s_t \backslash \{i, j\})\right). \tag{3}$$

A large $\mathrm{InKL}_t(i)$ indicates that the predictive distribution of token i changes substantially when another token generated in the same step is removed. In other words, the prediction of token i depends on other tokens generated in the same diffusion step.

Second, ***Out-KL*** measures the token's influence on other same-step tokens. For each peer $j \in U_t \setminus \{i\}$, we mask the token j, and compare the prediction of Y_j when token i is visible against masked,

$$\mathrm{OutKL}_t(i) = \frac{1}{|U_t| - 1} \sum_{j \in U_t \setminus \{i\}} D_{\mathrm{KL}}\left(p_{\theta_D}(Y_j \mid x, s_t \backslash \{j\}) \,\|\, p_{\theta_D}(Y_j \mid x, s_t \backslash \{i, j\})\right). \tag{4}$$

A large $\mathrm{OutKL}_t(i)$ indicates that token i substantially changes the predictive distributions of other tokens committed in the same step. Thus, such a token acts as an anchor for same-step generation.

Empirical findings. For each sample, we use o3 to identify the earliest logically incorrect span, avoiding confounding from cascading errors. On 200 GSM8K failures from LLaDA-8B-Instruct, failure tokens show higher In-KL and Out-KL than normal tokens across all three diffusion stages,

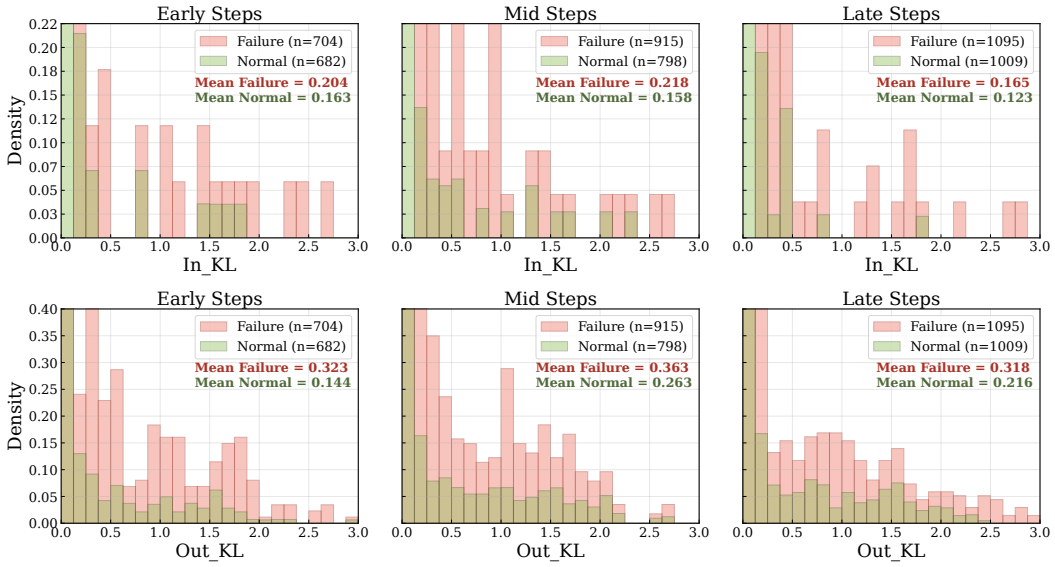

Figure 2: **Comparison of In-KL and Out-KL between failure and normal tokens.** In 200 erroneous GSM8K solutions generated by LLaDA-8B-Instruct ($L = 256$, $T = 64$), o3 identifies the earliest erroneous spans as *failure tokens*. We compute their In-KL and Out-KL relative to other same-step tokens across early ($t = 0$–20), mid ($t = 21$–41), and late ($t = 42$–63) generation steps.

with means roughly 1.5–$2\times$ larger (Figure 2). For example, in the early stage, their mean In-KL/Out-KL is $0.204/0.323$, versus $0.163/0.144$ for normal tokens. Out-KL also exceeds In-KL, suggesting that failure tokens strongly influence same-step peers.

Conclusion. Failure tokens are not uncertain positions, but **high-coupling positions that both depend on and influence simultaneously generated tokens, making the same-step independence approximation (Eq. 2) particularly weak.** They are poorly suited for simulte diffusion updates.

2.3 Failure Tokens Are Best Repaired Sequentially

Highly coupled failure tokens are ill-suited for parallel diffusion decoding; instead of slow left-to-right generation, we autoregress only short failure spans, leaving most tokens to fast diffusion decoding.

Oracle Refinement Analysis. We conduct an oracle analysis on LLaDA-8B-Instruct. For each sample, we run diffusion decoding and use o3 (OpenAI, 2025) solely to detect the earliest incorrect decoding step and its contiguous failure span. At this step, we serialize the dLLM state as text, preserving decoded tokens and representing unresolved positions with `[MASK]`. Qwen3-4B-Instruct (Qwen Team, 2025) receives this state and the failure-span index, then autoregressively generates a replacement span. We overwrite the erroneous span and resume standard LLaDA diffusion decoding.

Figure 3 reveals two findings. First, *repairing few tokens yields large gains*: correcting the first failure span improves performance by 28.9%, 27.0%, and 47.3% across the three tasks, while AR generates fewer than 10% of tokens. Second, *failure-targeted repair is more effective*: random-span refinement achieves smaller gains despite using a larger AR token budget. Thus, expensive AR computation is most useful on high-impact failure tokens, while parallel diffusion remains sufficient for most tokens.

Conclusion. By converting a locally coupled joint prediction into sequential conditional refinement, **AR can effectively repair failure tokens. Applying SC-AR to only a small fraction of tokens yields substantial gains while preserving fast parallel diffusion for the majority.**

3 FlexDLLM Framework

Motivated by above observations, in this section, we introduce **FlexDLLM**, a hybrid decoding framework that dynamically switches between parallel diffusion decoding and autoregressive refinement.

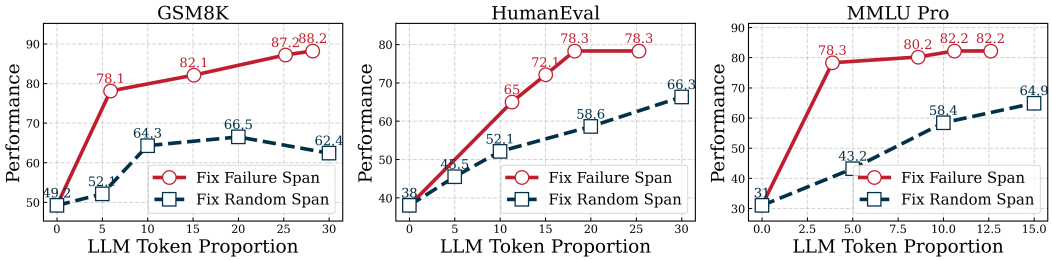

Figure 3: **Oracle SC-AR repair of failure spans.** We evaluate LLaDA-8B-Instruct across three tasks. For *Fix Failure Span*, o3 identifies the first erroneous span, which Qwen3-4B-Instruct then refines via SC-AR conditioned on the full dLLM state. Red points denote a maximum of $0/1/2/3/4$ repairs per sample, with the x-axis reporting the resulting proportion of AR-generated tokens. As a control, *Fix Random Span* applies this exact same SC-AR procedure to randomly selected token spans.

3.1 Adaptive Inference

During inference, FlexDLLM dynamically alternates between two modes: *Refinement-Gated Diffusion* (RG-Diff) and *State-Conditioned Autoregressive Refinement* (SC-AR). RG-Diff primarily performs parallel diffusion decoding but emits an <ASK> token at high-risk positions. SC-AR is triggered when <ASK> appear, and it repairs the span through autoregressive refinement.

RG-Diff Mode. RG-Diff is FlexDLLM's default mode. At each diffusion step t, it performs parallel token prediction over state s_t while emitting a special refinement token <ASK> at high-risk positions likely to fail. Without <ASK>, decoding proceeds normally; when <ASK> appears at position f, FlexDLLM switches to SC-AR to repair the failure-prone span from f.

SC-AR Mode. Section 2.3 shows that autoregressive refinement can resolve failure tokens. To reproduce this behavior, SC-AR gives the dLLM the same state and autoregressive decoding procedure. As shown in Figure 1, rather than replacing tokens at diffusion positions, SC-AR treats the denoising state as context, appends a temporary workspace, and autoregressively generates a refinement span.

Formally, let N denote the sequence length of the original diffusion region. During diffusion step t, if the RG-Diff emits an ASK token, SC-AR resolves the failure through a four-step refinement protocol:

1.Workspace Allocation. Rather than directly modifying position f, SC-AR appends a temporary workspace with a short, predefined maximum capacity C to the end of the current diffusion state s_t ($C = 10$ in our setting). This bounded workspace isolates the AR generation from diffusion region.

2. Autoregressive Generation. Within this workspace, SC-AR predicts the refinement sequence autoregressively. At the k-th SC-AR step, the augmented state $\bar{s}_t^{(k)}$ consists of the N original diffusion tokens followed by the k previously generated refinement tokens. To generate the next token $\hat{y}_k$ at position $N + k$, the SC-AR adapter (parameterized by θ_A) conditions on the source context x, the current augmented state $\bar{s}_t^{(k)}$, and the target failure position f:

$$\hat{y}_k = \arg\max_y p_{\theta_A}\left(Y_{N+k} = y \mid x,\, \bar{s}_t^{(k)},\, f\right), \tag{5}$$

3. Dynamic Termination. Notably, SC-AR does not rigidly generate all C tokens. The autoregressive step continues until the model emits an <EOS> token or reaches the capacity limit C. This mechanism allows the model to dynamically control the actual refinement length L, ensuring $L \leq C$.

4. Integration and Resumption. Once terminated, the unutilized positions in the workspace are discarded. The newly generated refinement span $(\hat{y}_0, \ldots, \hat{y}_{L-1})$ is then integrated back into the original diffusion state to replace the sequence from position f to $f + L - 1$. This operation directly overwrites the existing tokens in the original diffusion region. Finally, the temporary workspace is cleared, and the standard diffusion process resumes from the updated state.

SC-AR offers two advantages for handling failure tokens. First, it converts locally coupled joint predictions into tractable sequential decisions, better accommodating their high contextual sensitivity and steering decoding toward correct trajectories. Second, SC-AR provides an effective paradigm

for knowledge distillation by aligning the dLLM with the teacher LLM: in autoregressive mode, the dLLM shares the same causal context and sequential output states, enabling faithful capability transfer and the substantial gains shown in Section 2.3.

Mode Alternation. Overall, FlexDLLM uses RG-Diff as the default decoding mode and invokes SC-AR only when RG-Diff emits <ASK>. After the generated repair span is inserted back into the original diffusion state, decodingn switches back to RG-Diff. The autoregressive computation is therefore restricted to the additional AR workspace and is spent only on failure-prone spans.

3.2 Two-Stage Training

FlexDLLM equips a shared dLLM backbone with two specialized and switchable LoRA adapters via two-stage training procedure. Stage I trains the Refinement-Gated Diffusion (***RG-Diff***) adapter to determine when to emit <ASK>, Stage II trains the SC-AR adapter to generate the refinement span.

Stage I: Training RG-Diff Mode. Stage I aims to train the model to maintain standard parallel generation while learning to emit the <ASK> on failure positions. For each training pair (x, y), we analyze the decoding trajectory of the base dLLM. When the first error occurs at step t, we let s_t represent the corresponding pre-update state and f denotes the starting failure position. We also use $\mathcal{V}_t$ denotes the set of visible positions in s_t, and p_{θ_D} denote token distribution under RG-Diff mode.

We train RG-Diff with three complementary objectives.

Objective 1: Normal Diffusion Generation. RG-Diff should behave like a standard diffusion decoder before any failure is detected. Starting from s_t, we re-mask a subset $\mathcal{M}_{\text{pre}} \subseteq \mathcal{V}_t$ using the standard MDLM corruption process, and denote the corrupted state by $\tilde{s}_t^{\text{pre}}$. The model is trained to recover the re-masked target tokens, while being penalized for emitting <ASK> on normal visible positions:

$$\mathcal{L}_{\text{n}} = \frac{1}{|\mathcal{M}_{\text{pre}}|} \sum_{i \in \mathcal{M}_{\text{pre}}} -\log p_{\theta_D}\left(Y_i = y_i \mid x, \tilde{s}_t^{\text{pre}}\right) - \lambda_{\text{noask}} \frac{1}{|\mathcal{V}_t|} \sum_{i \in \mathcal{V}_t} \log\left(1 - p_{\theta_D}\left(Y_i = \texttt{<ASK>} \mid x, \tilde{s}_t^{\text{pre}}\right)\right), \tag{6}$$

where λ_{noask} controls the strength of the unnecessary-ASK penalty. The first term preserves ordinary diffusion denoising, and the second term prevents the model from overusing the <ASK> token.

Objective 2: Failure Localization. At failure step, RG-Diff should request SC-AR refinement at the failure position. We therefore supervise the model to predict <ASK> at starting failure position f. Since exact single-position supervision can be brittle in parallel diffusion decoding, we also allow a small localization tolerance. Let $\mathcal{W}_f = \{i : |i - f| \le h\}$ be a local window around f, where h is the window radius. The failure-localization loss is

$$\mathcal{L}_{\text{ask}} = -\log p_{\theta_D}\left(Y_f = \texttt{<ASK>} \mid x, s_t\right) - \lambda_{\text{win}} \left[\log\left(1 - \prod_{i \in \mathcal{W}_f} \left(1 - p_{\theta_D}\left(Y_i = \texttt{<ASK>} \mid x, s_t\right)\right)\right)\right], \tag{7}$$

where λ_{win} weights the local-window presence term. The first term teaches the exact failure anchor, while the second term encourages at least one <ASK> prediction near the failure position.

Objective 3: Post-refinement Continuation. After SC-AR inserts a span, RG-Diff should resume diffusion decoding from the repaired state. To simulate this, we integrate the ground-truth target span y_F into positions F of s_t to form an updated state s_t^+. We then apply the MDLM corruption only to the remaining unobserved tokens. Over this newly masked subset $\mathcal{M}_{\text{post}}$, the post-refinement loss is:

$$\mathcal{L}_{\text{post}} = \frac{1}{|\mathcal{M}_{\text{post}}|} \sum_{i \in \mathcal{M}_{\text{post}}} -\log p_{\theta_D}\left(Y_i = y_i \mid x, \tilde{s}_t^+\right). \tag{8}$$

This teaches the model to treat the inserted span as a stable anchor for subsequent decoding.

The overall Stage-I objective is $\mathcal{L}_{\text{RG}} = \mathcal{L}_{\text{n}} + \mathcal{L}_{\text{ask}} + \mathcal{L}_{\text{post}}$, which train RG-Diff to generate normally on easy tokens, localize failure spans with <ASK>, and continue coherently after refinement.

Stage II: Training SC-AR Mode. Stage II trains the SC-AR mode to generate a refinement span at the appended workspace when requested by an token. To construct the training set, we run the trained RG-Diff mode over the training samples. Whenever RG-Diff emits an token at position f under diffusion state s_t, we extract a refinement query (x, s_t, f). This query is provided to a

Table 1: **Performance Comparison with state-of-the-art Baselines.** All methods are evaluated with a unified setting of $K = 4$, where K denotes the number of tokens unmasked per diffusion step. We set task-specific maximum generation lengths to match the expected output length: 256 for GSM8K, HumanEval, and MBPP; 512 for MATH; and 64 for MMLU-Pro and ARC-C. All baselines are evaluated under the same evaluation protocol, and evaluation conduct on OpenCompass platform. FlexDLLM (oracle) denotes the variant that uses Qwen3-4B-Instruct for refinement after Stage-I training. The best result in each column is marked in **green**, and the second-best result is blue.

Method	Inference Strategy		Math		Code		General Reasoning	
	Self-Correct	Token Schedule	GSM8K	MATH	HumanEval	MBPP	MMLU-Pro	ARC-C
Autoregressive LLM References								
Llama3.1-8B-Instruct	–	–	76.8	29.6	58.5	57.8	48.3	83.4
DeepSeek-7B	–	–	63.0	15.8	48.2	35.2	39.8	49.4
LLaDA-8B-Base dLLM Variants								
LLaDA-8B-Base	–	–	56.1	22.4	20.7	24.6	30.3	45.9
+ BackPlay v2	✓	✗	67.5	28.4	27.4	33.8	–	–
+ ProSeCo	✓	✗	72.0	46.0	52.2	48.2	25.2	60.8
LLaDA-8B-Instruct dLLM Variants								
LLaDA-8B-Instruct	–	–	49.2	36.5	38.0	20.0	31.0	87.0
+ Vanilla SFT	–	–	56.8	43.1	50.6	42.3	49.5	89.2
+ ReMeDi	✓	✗	62.1	46.2	34.2	45.1	30.5	78.3
+ ReMDM	✓	✗	52.2	24.7	32.6	28.2	30.8	**89.6**
+ LLaDOU	✗	✓	77.1	44.6	53.1	**51.6**	43.4	77.8
+ FLEXDLLM	✓	✓	**79.6**	**53.4**	**58.2**	51.2	**54.5**	89.3
+ FLEXDLLM (Oracle)	✓	✓	85.1	59.2	66.5	55.8	63.2	90.2

teacher LLM, which generates an adaptive-length refined span. Let $\bar{y} = (\bar{y}_0, \ldots, \bar{y}_{L-1})$ denote this teacher-generated span of length L. Finally, span $\bar{y}$ serves as the supervision signal to train SC-AR.

To strictly align with the inference mechanism, we train SC-AR using an appended workspace. Let N denote the length of the original diffusion state s_t. For each autoregressive step $k \in \{0, \ldots, L-1\}$, we construct an augmented state $\bar{s}_t^{(k)}$ by explicitly concatenating the original state s_t (occupying the first N positions) with the teacher's prefix $\bar{y}_{0:k-1}$ (occupying positions $N, \ldots, N+k-1$). The model is trained to predict the next target token $\bar{y}_k$ at the active workspace position $N + k$, conditioned on the input prompt x, the augmented state $\bar{s}_t^{(k)}$, and the failure position f:

$$\mathcal{L}_{\text{SC-AR}} = \frac{1}{L} \sum_{k=0}^{L-1} -\log p_{\theta_A} \left(Y_{N+k} = \bar{y}_k \mid x,\, \bar{s}_t^{(k)},\, f \right), \tag{9}$$

We sample an active position k per teacher span, giving an unbiased single-token estimate of the objective. This stage distills the teacher's state-conditioned refinement behavior into SC-AR, enabling it to generate reliable spans in the temporary workspace for positions requested by RG-Diff mode.

4 EXPERIMENTS

Models and Datasets. We conduct our main experiments with LLaDA-8B-Instruct (Nie et al., 2025). For training, we combine four datasets: GSM8K training set (Cobbe et al., 2021), MATH training set (Hendrycks et al., 2021), Magicoder-OSS-Instruct (Wei et al., 2023), and CommonsenseQA (Talmor et al., 2019). For evaluation, we consider six benchmarks: mathematical, including GSM8K and MATH; code generation, including HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021b); and general reasoning, including MMLU-Pro (Wang et al., 2024) and ARC-C (Clark et al., 2018).

Implementation Details. Training data is constructed via o3 (OpenAI, 2025) by identifying the first error step, failure span, and corrected span in dLLM trajectories. We utilize Qwen3-4B-Instruct (Qwen Team, 2025) as Stage II teacher. Training spans four epochs (batch size 128) on four H200 GPUs. Hyperparameters are $\lambda_{\text{noask}} = 0.1$, $\lambda_{\text{win}} = 0.5$, and local window $h = 4$, with LoRA rank 32 and $\alpha = 64$. By default, inference unmasks $K = 4$ tokens per step. Details are in Appendix D.

Table 2: **Hardware Efficiency of FlexDLLM.** We measure hardware efficiency on a single H200 GPU with batch size 1. To reduce measurement variance, we inference 200 samples for each setting and report the average memory usage, peak memory usage, and per-sample latency. "Adapter" denotes the average per-sample time spent by FlexDLLM switching between the two adapters.

Method	Setting		Memory		Latency		Accuracy	
	Generate Length	Sampling Steps	Avg. Mem. (GB)	Peak Mem. (GB)	Per Sample (ms)	Adapter (ms/sample)	GSM8K	HumanEval
LLaDA-8B-Instruct	L=256, T=64 (r=4)		14.97	16.03	1403	–	49.2	38.2
	L=256, T=128 (r=2)		14.97	16.03	2790	–	54.0	43.1
	L=256, T=256 (r=1)		14.95	16.21	4852	–	76.2	47.0
FlexDLLM	Diff. r=4 + AR fallback		15.22	16.27	2065	61.1	**79.6**	**58.2**

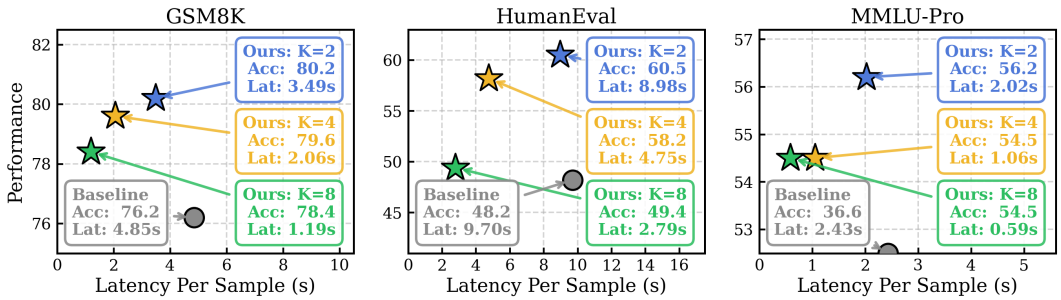

Figure 4: **Accuracy-Latency Trade-off of FlexDLLM.** We evaluate both latency and accuracy when the diffusion branch unmasks different numbers of tokens per diffusion step, with $K \in \{2, 4, 8\}$. Lat is average per-sample inference latency on the corresponding benchmark, measured on a single H200 with batch size 1. For baseline, we report performance of the LLaDA-8B-Instruct under $K = 1$.

Baselines. We compare FlexDLLM against six baselines: (1) four self-correction methods—BackPlay v2 (Liu et al., 2026), ProSeCo (Schiff et al., 2026), ReMeDi (Huang et al., 2025a), and ReMDM (Wang et al., 2025)—that revise previously generated tokens during decoding; (2) LLaDOU (Huang et al., 2025b), a token-scheduling method dynamically optimizing unmasking orders based on context; and (3) Vanilla SFT (Ouyang et al., 2022), serving as a standard baseline to quantify gains from inference-time interventions. See Appendix E for details.

4.1 Main Results

Overall Performance. Table 1 shows that FlexDLLM consistently improves performance across all six benchmarks and achieves the best overall results among all methods. The gains are especially clear on the challenging MATH and MBPP benchmarks, where FlexDLLM reaches 53.4% and 54.2% accuracy, outperforming the second-best method by 8.6% and 5.0%, respectively. This suggests that refining a small number of critical hard tokens is particularly effective for complex reasoning tasks where the original DLLM struggles. Moreover, FlexDLLM remains close to the oracle setting using an external LLM, indicating that Stage-II training successfully adapts the LoRA adapter to this refinement paradigm and delivers large gains with minor overhead.

Hardware Efficiency. Evaluated on a single GPU (Table 2), FlexDLLM's lightweight LoRA adapters incur negligible memory overhead ($+1.6\%$ average, $+1.5\%$ peak), while its adaptive combination of diffusion decoding and AR refinement achieves a better latency–accuracy trade-off than both the original model and standard LLM baselines. FlexDLLM outperforms LLaDA-8B-Instruct's strongest setting on GSM8K ($+3.4\%$) and HumanEval ($+11.2\%$) with a $2.3\times$ speedup. Performance also remains stable across different K values (Figure 4): increasing K from 2 to 8 reduces accuracy by only 1.8% on GSM8K and 1.7% on MMLU-Pro, while improving inference speed by $2.9\times$ and $3.4\times$, respectively. This robustness comes from refining decisive hard tokens with the AR branch as reliable anchors, allowing the diffusion decoder to parallelize the remaining easy tokens with little accuracy loss.

Table 3: **Effectiveness of different <ASK> strategies.** We compare the positioning accuracy of different <ASK> strategies and their performance on GSM8K. For Oracle ASK, we use o3 to correct the erroneous spans directly and treat the corresponding oracle <ASK> positions as the reference set for computing ASK precision and ASK recall. For Random ASK, o3 inserts <ASK> at random decoding steps. For Low-Confidence ASK, we insert <ASK> at low-confidence positions.

Method	Performance	Avg. ASK / Sample	ASK Precision	ASK Recall	Gain / AR Token
Oracle ASK	88.2	0.83	100%	100%	1.78%
No ASK	49.2	–	–	–	–
Random ASK	57.1	1.61	16.2%	31.4%	0.25%
Low-Confidence ASK	52.3	1.82	12.1%	26.8%	0.12%
Learned ASK (Ours)	**85.1**	**1.51**	**42.5%**	**77.3%**	**1.26%**

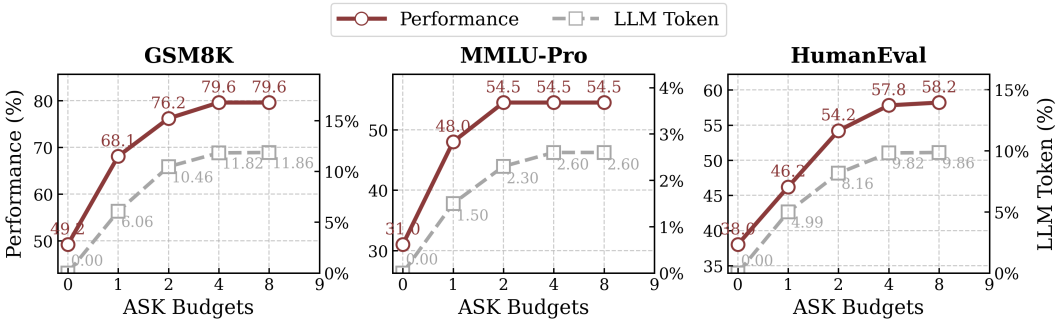

Figure 5: **Effect of the <ASK> budget.** We evaluate the performance under different budgets. A budget of n means that the model can output <ASK> at most n times for each sample. For each setting, we report the proportion of tokens in the final response that are generated by the AR branch.

4.2 Ablation Study

Correctness of <ASK> positions. We examine whether FlexDLLM learns to place <ASK> at positions, using the LLM refiner across selection strategies for fair comparison. As shown in Table 3, the learned strategy approaches the oracle, with only a 3.1% performance gap and 77.3% ASK recall. This suggests that failure-prone positions, which cannot be reliably handled by parallel decoding, are reflected in dLLM hidden states and can be learned. In contrast, refining low-confidence positions performs even worse than random <ASK> selection. Failure tokens are therefore not low-confidence tokens; the model can be highly confident in incorrect predictions. Thus, output distributions alone are insufficient for failure detection, motivating explicit training of <ASK> placement.

Effect of the <ASK> budget. We further evaluate FlexDLLM under different <ASK> budgets, reporting both task performance and the fraction of AR-generated tokens in Figure 5. Even a budget of 1 yields substantial improvements across all three tasks, while additional gains diminish as the budget increases, indicating that only a few well-placed AR refinements are needed. Stable improvements require roughly 10% AR-generated tokens on GSM8K and HumanEval, but only about 3% on MMLU-Pro. This difference mainly reflects MMLU-Pro's shorter and simpler outputs, which trigger refinement less frequently; moreover, its AR branch typically generates only a few key words rather than longer mathematical or code spans. Overall, FlexDLLM achieves substantial gains while generating only a small fraction of tokens autoregressively.

5 Conclusion

In this paper, we introduced FlexDLLM, an adaptive hybrid inference framework designed to overcome the fundamental accuracy-latency trade-off in diffusion language models. By autonomously identifying high-risk token positions and selectively applying State-Conditioned Autoregressive Refinement (SC-AR), our approach seamlessly integrates the speed of parallel diffusion with the reliability of AR generation. Extensive evaluations demonstrate that FlexDLLM establishes a superior accuracy-latency Pareto frontier, yielding substantial performance gains on complex reasoning tasks.

AI USE STATEMENT

We used OpenAI o3 to assist with identifying the earliest logically incorrect spans in generated solutions and with constructing the corresponding annotations used in our analysis and training pipeline. We used GPT-5.5 to assist with editing the manuscript for readability and clarity. All AI-assisted annotations, analyses, and manuscript edits were reviewed by the authors, who take responsibility for the final content and results.

ETHICS STATEMENT

This work does not involve human subjects, personally identifiable information, or other sensitive data. We are not aware of any specific ethical concerns arising from the methods, datasets, or experiments presented in this work.

REFERENCES

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.

Enkelejda Kasneci, Kathrin Seßler, Stefan Küchemann, Maria Bannert, Daryna Dementieva, Frank Fischer, Urs Gasser, Georg Groh, Stephan Günnemann, Eyke Hüllermeier, et al. Chatgpt for good? on opportunities and challenges of large language models for education. *Learning and individual differences*, 103:102274, 2023.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

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.

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

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.

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

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.

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

Wonjun Kang, Kevin Galim, Seunghyuk Oh, Minjae Lee, Yuchen Zeng, Shuibai Zhang, Coleman Hooper, Yuezhou Hu, Hyung Il Koo, Nam Ik Cho, et al. Parallelbench: Understanding the trade-offs of parallel decoding in diffusion llms. *arXiv preprint arXiv:2510.04767*, 2025.

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

Daniel Israel, Tian Jin, Ellie Cheng, Guy Van den Broeck, Aditya Grover, Suvinay Subramanian, and Michael Carbin. Planned diffusion. *arXiv preprint arXiv:2510.18087*, 2025.

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

Bumjun Kim, Dongjae Jeon, Moongyu Jeon, and Albert No. Dependency-aware parallel decoding via attention for diffusion llms. *arXiv preprint arXiv:2603.12996*, 2026.

Liran Ringel, Ameen Ali, and Yaniv Romano. Dependency-guided parallel decoding in discrete diffusion language models. *arXiv preprint arXiv:2604.02560*, 2026.

OpenAI. OpenAI o3 and o4-mini system card, 2025. URL `https://openai.com/index/o3-o4-mini-system-card/`. Accessed: 2026-05-06.

Qwen Team. Qwen3 technical report, 2025. URL `https://arxiv.org/abs/2505.09388`.

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.

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

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with oss-instruct. *arXiv preprint arXiv:2312.02120*, 2023.

Alon Talmor, Jonathan Herzig, Nicholas Lourie, and Jonathan Berant. Commonsenseqa: A question answering challenge targeting commonsense knowledge. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4149–4158, 2019.

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.

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, et al. Mmlu-pro: A more robust and challenging multi-task language understanding benchmark. *Advances in Neural Information Processing Systems*, 37: 95266–95290, 2024.

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.

Liming Liu, Binxuan Huang, Zixuan Zhang, Xin Liu, Bing Yin, and Tuo Zhao. Backplay: Head-only look-back self-correction for diffusion language models, 2026. URL `https://arxiv.org/abs/2601.06428`.

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

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

Guanghan Wang, Yair Schiff, Subham Sekhar Sahoo, and Volodymyr Kuleshov. Remasking discrete diffusion models with inference-time scaling. *arXiv preprint arXiv:2503.00307*, 2025.

Zemin Huang, Zhiyang Chen, Zijun Wang, Tiancheng Li, and Guo-Jun Qi. Reinforcing the diffusion chain of lateral thought with diffusion language models. *arXiv preprint arXiv:2505.10446*, 2025b.

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.

Wenrui Bao, Zhiben Chen, Dan Xu, and Yuzhang Shang. Learning to parallel: Accelerating diffusion large language models via learnable parallel decoding. *arXiv preprint arXiv:2509.25188*, 2025.

Parikshit Bansal and Sujay Sanghavi. Enabling approximate joint sampling in diffusion lms. *arXiv preprint arXiv:2509.22738*, 2025.

Shuibai Zhang, Fred Zhangzhi Peng, Yiheng Zhang, Jin Pan, and Grigorios G Chrysos. Corrective diffusion language models. *arXiv preprint arXiv:2512.15596*, 2025.

Jinwei Zhang, Dimitri von Rütte, Yuhui Ding, and Thomas Hofmann. Denoising is not the end: Discrete diffusion language models with self-correction.

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.

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. *arXiv preprint arXiv:2203.11171*, 2022.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023.

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 *The twelfth international conference on learning representations*, 2023.

A Appendices Organization

Organization In this Appendix, we provide in-depth descriptions of the materials that are not covered in the main paper, and report additional visualization results. The document is organized as follows:

- **B**- Related Work
- **C**- Algorithm of FlexDLLM
- **D**- Detail Training Setting
  - **D.1** Data Construction and Training of Stage I
  - **D.2** Data Construction and Training of Stage II
- **E**- Detail Evaluation Setting
- **F**- Discussion and Future Work

B Related Work

Diffusion language models and their inference limitations. Diffusion language models (dLLMs) generate text by gradually denoising a corrupted sequence, providing a non-autoregressive alternative to left-to-right factorization. Early discrete and continuous text diffusion models established the feasibility of denoising-based language generation (Austin et al., 2021a; Li et al., 2022), and recent masked diffusion objectives and large-scale training have narrowed the gap to autoregressive LLMs (Sahoo et al., 2024; Nie et al., 2025; Ye et al., 2025). Their central advantage is that multiple masked positions can be updated within a single denoising step. However, practical parallel decoding usually commits tokens from token-wise marginal distributions, which implicitly treats same-step predictions as conditionally independent given the current state. This approximation becomes fragile when positions are semantically or syntactically coupled, causing a pronounced speed–quality trade-off under aggressive parallelism (Wu et al., 2025a; Kang et al., 2025). Existing mitigation strategies either restrict parallel updates to high-confidence positions (Wu et al., 2025a), learn a token-wise filter for adaptive unmasking (Bao et al., 2025), approximate joint multi-token sampling (Bansal and Sanghavi, 2025), or explicitly avoid co-updating dependent tokens by constructing dependency graphs. A related line studies self-correction in dLLMs: although bidirectional denoising can in principle revise previous tokens, standard masked diffusion objectives often fail to reliably detect visible errors, motivating correction-oriented post-training or look-back correction mechanisms (Zhang et al., 2025; Liu et al., 2026; Zhang et al.). These methods improve the robustness of diffusion decoding, but they still primarily rely on confidence, remasking, or auxiliary correction heads rather than providing strict sequential conditioning for the small set of highly coupled failure-prone positions.

Hybrid autoregressive–diffusion generation. Another line of work attempts to combine the reliability of autoregressive factorization with the parallelism of diffusion decoding. Block Diffusion defines an autoregressive distribution over blocks of tokens while applying diffusion-based denoising within each block, thereby supporting flexible-length generation, KV caching, and parallel token sampling (Arriola et al., 2025). Planned Diffusion further introduces an explicit planning stage: the model first autoregressively generates a plan that partitions the response into semantically independent spans, and then denoises those spans in parallel (Israel et al., 2025). These approaches demonstrate that coarse-grained autoregressive structure can substantially improve the efficiency and controllability of diffusion generation. Nevertheless, their autoregressive guidance is imposed mainly at the block or chunk level. Once a block is selected for diffusion denoising, tokens inside the block are still predicted through parallel updates, so strong token-level dependencies may again be weakened by the same-step independence approximation. This limitation motivates finer-grained hybridization, where autoregressive computation is not uniformly assigned to every block, but is instead selectively invoked at the specific positions where diffusion parallelism is most likely to fail.

Autoregressive reasoning and test-time computation. A complementary body of work on LLM reasoning shows that difficult problems often benefit from explicit sequential computation. Chain-of-thought prompting and zero-shot reasoning elicit intermediate reasoning steps from autoregressive LLMs (Wei et al., 2022; Kojima et al., 2022), while self-consistency improves robustness by sampling and aggregating multiple autoregressive reasoning trajectories (Wang et al., 2022). More deliberate

inference methods, such as Tree-of-Thoughts, expand reasoning from a single left-to-right trajectory to search over coherent intermediate thoughts (Yao et al., 2023). Verifier- and process-supervision-based methods further demonstrate that allocating additional test-time computation to intermediate reasoning states can improve reliability on challenging mathematical tasks (Lightman et al., 2023; Snell et al., 2024). The common strength of these methods lies in sequential conditioning: later reasoning tokens are explicitly conditioned on earlier decisions, making them well suited for handling long-range dependencies and correcting local mistakes. However, applying autoregressive reasoning to the entire output sacrifices the parallel efficiency that motivates dLLMs in the first place. This suggests a more targeted design principle for hybrid dLLM inference: preserve diffusion decoding for easy, weakly coupled regions, while reserving autoregressive refinement for the small set of difficult spans where sequential dependence is essential.

C Algorithm of FlexDLLM

Algorithm 1: FlexDLLM Framework

Require : training set $\mathcal{D}$, test prompt $x^\star$, dLLM backbone ϕ, RG-Diff adapter θ_D, SC-AR adapter θ_A, teacher LLM $\mathcal{T}$, maximum diffusion steps K

Return : trained adapters θ_D, θ_A and generated output

Training Phase:

Stage I: Train Refinement-Gated Diffusion Mode.

for $(x, y) \sim \mathcal{D}$ **do**

  Dataset Construction: Run the base dLLM trajectory and locate the first failure state (s_t, f)

  Objective Update:

  Objective 1: Normal Diffusion Generation. $\mathcal{L}_{\text{norm}}$ via Eq. equation 6 ▷ `Preserve standard diffusion.`

  Objective 2: Failure Localization. $\mathcal{L}_{\text{ask}}$ via Eq. equation 7 ▷ `Learn localize failure position.`

  Objective 3: Post-refinement Continuation.$\mathcal{L}_{\text{post}}$ via Eq. equation 8 ▷ `Learn coherent continuation.`

  Update θ_D by minimizing $\mathcal{L}_{\text{RG}} = \mathcal{L}_{\text{norm}} + \mathcal{L}_{\text{ask}} + \mathcal{L}_{\text{post}}$

Stage II: Train State-Conditioned Mode.

for $(x, y) \sim \mathcal{D}$ **do**

  Dataset Construction: Run RG-Diff with teacher model refine, collect teacher output $\bar{y}_F \leftarrow \mathcal{T}(x, s_t, f)$

  Objective Update:

  Objective: Autoregressive Refinement. $\mathcal{L}_{\text{SC-AR}}$ via Eq. equation 9 ▷ `Learn refine target span.`

  Update θ_A by minimizing $\mathcal{L}_{\text{SC-AR}}$

Inference Phase:

for $t \leftarrow 0$ **to** $K - 1$ **do**

  Run RG-Diff mode on $(x^\star, s_t)$ to predict parallel token updates

  if *<ASK> is generated at* f **then**

    Switch to SC-AR mode and generate refine span $\hat{y}_F$ via Eq. equation 5.

    $s_{t+1} \leftarrow \text{Overwrite}(s_t, F, \hat{y}_F)$ ▷ `Local autoregressive repair`

  else $s_{t+1} \leftarrow \text{Commit}(s_t)$ ▷ `Standard parallel diffusion`

D Detail Training Setting

D.1 Data Construction and Training of Stage I

In this section, we detail the data construction and training specifics for Stage I (RG-Diff).

D.1.1 Data Construction

This section describes how we construct the supervision data used to train the refinement-gated diffusion mode. The goal of the construction procedure is to convert raw diffusion rollouts into localized intervention examples: for each failed rollout, we identify the earliest harmful commitment made by the diffusion model, determine the last decoding state at which the model could still have

asked for help, and generate a minimally edited target output that represents what the model should have produced after receiving such help.

Raw trajectory dump. We start from a clean trajectory dump produced by the base dLLM. Each record contains the source dataset identifier, split, source index, task question x, reference answer y, the original prompt given to the dLLM, the final rollout $\hat{y}$, the total number of diffusion steps T, the number of tokens unmasked per step, and the full decoding trajectory

$$\mathcal{S} = \{s_1, s_2, \ldots, s_T\}.$$

Each state s_t is a partially denoised textual sequence in which visible positions contain committed tokens and unresolved positions are represented by the mask symbol <MASK>. We do not pre-filter the trajectory dump by correctness. Instead, every rollout is passed to an external teacher model, which decides whether the rollout is already correct, fixable through a localized intervention.

Step I: trajectory-aware localization. The first annotation stage identifies the earliest point at which the dLLM commits to an error. For each record, the teacher receives the question, reference answer, original prompt, final rollout, the number of diffusion steps, and the full trajectory. It returns either a non-fixable decision or a tuple

$$a_1 = (\texttt{stop_step}, B, C),$$

where B is the wrong span and C is the corresponding corrected span. The span B must be a verbatim contiguous substring of the final rollout. Its semantic type is domain dependent: a concise arithmetic expression for math tasks, a complete erroneous code line or statement for coding tasks, and a short answer-bearing phrase or answer marker for commonsense multiple-choice tasks. The corrected span C mirrors the local role of B and is inferred from the reference answer.

The most important field is `stop_step`. We define it as the step immediately before the first token or character of B becomes visible in the diffusion trajectory. Concretely, the teacher first locates B in the final rollout, then scans the trajectory in chronological order to find the smallest step τ such that at least one position in the region corresponding to B is no longer masked. We then set

$$\texttt{stop_step} = \max(1, \tau - 1).$$

Thus, at state $s_{\texttt{stop_step}}$, the erroneous span is still fully masked. This definition is deliberately causal: the supervision marks the last state at which the dLLM could have emitted <ASK> before committing to the wrong span. It prevents the annotation from using a step at which the error is already visible, which would make the refinement request too late.

Step II: minimally edited repaired output. Only records labeled `fixable` in Stage I are passed to Stage II. The second stage receives the question, reference answer, final rollout, `stop_step`, B, and C, but not the full trajectory. This separation makes the second stage a pure repair step rather than another opportunity to relocalize the error. It produces

$$a_2 = (\texttt{ideal_generate_after_help}, \texttt{minimal_edit_possible}),$$

where `ideal_generate_after_help` is the output the dLLM should have generated if it had deferred to the external refinement model at `stop_step`. For math and commonsense tasks, the repair is required to be surgical: replace B by C, update only directly dependent downstream numbers, formulas, answer phrases, or final answer markers, and otherwise copy the original rollout character-for-character. This constraint preserves the original dLLM reasoning style and prevents the teacher from rewriting the entire answer.

Coding tasks require a slightly different repair policy. Because an incorrect code line can invalidate the entire remaining program, the repaired output preserves the correct code prefix before B exactly, inserts C in place of B, and then allows the teacher to regenerate the remaining code to obtain a single functionally correct implementation. The teacher is forbidden from modifying the committed code prefix, adding explanations, emitting markdown fences, or returning multiple definitions of the same function. The output is required to be bare code.

Filtering and retained metadata. The resulting annotation file preserves the original record fields together with the Stage-I annotation, Stage-II repair, token usage, and possible error messages. Records are accepted as positive refinement examples only when Stage I returns `fixable`, Stage II returns `minimal_edit_possible=true`, and the repaired output contains the corrected span in the intended local position. Records marked as already correct or not localizable are retained in the intermediate annotation file for auditing and ablation, but they are not used as positive <ASK>-triggered repair examples.

Annotation prompts. Below we show the system prompts used in the two-step annotation pipeline. Few-shot demonstrations are omitted. In the implementation, <ASK_LLM> is the annotation-time placeholder for the same intervention action represented as <ASK> in the model training objective.

System prompt for Step I: trajectory-aware error localization

```
You are a data annotation model for cooperative decoding between a dLLM
(diffusion language model) and an external LLM.

You will receive ONE JSON sample containing:
- question
- standard_answer
- prompt
- final_output
- trajectory: a list of {step, state} entries showing the dLLM's progressive
  decoding. step is 1-based. Each state uses the mask symbol to mark positions
  that are still unresolved.
- n_steps, unmask_per_step

TASK: annotate the earliest local harmful commitment with:
1. stop_step: the step just BEFORE the first token/character of Wrong_span is
   decoded.
2. Wrong_span: a verbatim contiguous substring of final_output.
3. Fixed_span: the corresponding corrected local span.

To compute stop_step, locate Wrong_span in final_output, scan the trajectory
from early to late, find the first step at which any token/character in that
region becomes non-mask, and set stop_step to that step minus one, clamped to 1.
At stop_step, the Wrong_span region must still be fully masked. This is the last
moment at which the dLLM could have emitted <ASK_LLM> before committing to the
wrong span.

Domain policy:
- Math: choose the earliest harmful mathematical formula, equation, or
  arithmetic expression. Prefer concise formula-like spans over prose.
- Code: judge only the program. Choose the earliest erroneous complete code
  line or statement according to trajectory time, not merely textual order.
  Ignore explanation/prose errors. Downstream bugs do not block fixability if
  the earliest wrong line can be named.
- Commonsense: choose the earliest wrong answer-bearing key phrase or answer
  marker. Ignore grammar, repetition, awkward wording, and formatting noise if
  the answer-bearing content is correct.

Return "fixable" when the output is wrong and a single earliest local harmful
span can be identified. Return "not_fixable" when the rollout is already
correct, incomplete, badly formatted, unrelated, not localizable, or uncertain.

Output STRICT JSON only.

If fixable:
{
  "decision": "fixable",
  "stop_step": <int>,
  "Wrong_span": "<verbatim wrong span from final_output>",
  "Fixed_span": "<corresponding corrected span>"
}

If not fixable:
{
  "decision": "not_fixable",
  "is_correct": <true|false>,
  "non_fixable_type": "<already_correct | incomplete_generation |
                       bad_output_format | requires_global_rewrite |
                       harmful_point_not_localizable | uncertain>"
}
```

System prompt for Step 2: minimally edited repair

```
You are a data annotation model producing minimally-edited corrections of a
dLLM rollout.

You will receive a JSON input with:
- question
- standard_answer
- final_output
- stop_step
- Wrong_span
- Fixed_span

TASK: Produce ideal_generate_after_help, the output the dLLM should have
generated had it deferred to an external LLM at stop_step, just before
Wrong_span was decoded.

General requirements:
1. Fixed_span must appear verbatim in place of Wrong_span.
2. The final answer or program behavior must match standard_answer.
3. Preserve as much of final_output as possible.
4. Do not add or remove <ASK_LLM> tags.
5. Output STRICT JSON only.

For math:
- Copy final_output character-for-character except for Wrong_span, directly
  dependent downstream numbers, formulas, and forced unit changes.
- Do not rephrase, add sentences, fix grammar, or improve style.

For commonsense:
- Copy final_output character-for-character except for the wrong answer-bearing
  phrase and directly dependent answer letters or final answer markers.
- Do not rewrite the rationale or clean up unrelated wording/formatting.

For code:
- Output bare code only: no prose, no markdown fences, no explanations.
- Preserve the code prefix before Wrong_span character-for-character.
- Insert Fixed_span in place of Wrong_span.
- Regenerate any code after Wrong_span as needed to produce one canonical,
  functionally correct implementation.
- Do not modify code before Wrong_span.
- Do not emit duplicate definitions of the same function/class/method.

If the minimal-edit constraint cannot be satisfied, set
minimal_edit_possible=false and return the best salvageable output. Otherwise
set minimal_edit_possible=true.

{
  "ideal_generate_after_help": "<repaired output>",
  "minimal_edit_possible": <true|false>
}
```

D.1.2 Training Setting

This part summarizes the implementation of Stage-I training, where we train the Refinement-Gated Diffusion (RG-Diff) adapter. The purpose of this stage is to preserve the normal parallel diffusion behavior of the base dLLM while enabling the model to emit <ASK> at localized failure-prone positions. The three training objectives are described in the main paper; here we focus on how the corresponding training examples are encoded and optimized.

Training sample format. We use a unified format for RG-Diff training. Each example is constructed from two aligned annotation files: one file contains the pre-refinement diffusion state and the localized <ASK> position, and the other contains the teacher-repaired output after refinement. Concretely, the pre-call part contains a token-level diffusion canvas `canvas_token_ids`, where already revealed tokens remain visible and unresolved positions are represented by the diffusion mask token. The field `ask_token_idx` indicates the canonical failure anchor. We discard examples in which this anchor is not masked, since RG-Diff must learn to ask before wrong token is committed.

The pre-call input is formed by concatenating the task prompt with the masked diffusion canvas. Around the anchor, we define a small local window and a short set of jittered anchor candidates. In our default setting, the local window is ± 4 tokens around the anchor and the anchor jitter is 2,

Setting	GSM8K	Magicoder	MATH	Commonsense
Shared model and optimization settings				
Backbone	GSAI-ML/LLaDA-8B-Instruct			
LoRA rank / alpha / dropout	32 / 64 / 0.05			
LoRA target modules	q_proj,k_proj,v_proj,attn_out,ff_proj,up_proj			
Precision	bfloat16			
Base learning rate	1×10^{-4}			
New-token/output-head learning rate	5×10^{-4}			
Weight decay	0.0			
Warmup / scheduler	0.05 / cosine			
Max gradient norm	1.0			
Validation split / seed	5% / 42			
Anchor jitter / local window	2 / 4			
GPUs	4 H200 GPUs			
Task-specific schedules				
Training examples kept	1805/3019	8209/9137	2730/3976	1094/2165
Max output length	256	256	512	64
Epochs	8	4	8	12
Per-device train batch size	2	2	1	4
Gradient accumulation steps	16	16	32	8
Effective global batch size	96	96	96	96
Eval/save interval	15	40	25	15
Approx. optimization steps	150	342	228	137

Table 4: Key hyperparameters for Stage-I RG-Diff training. The effective global batch size is computed as per-device batch size $\times$ number of GPUs $\times$ gradient accumulation steps.

so the trainer can randomly choose among up to three valid masked positions near the canonical failure location. This avoids overfitting to a single exact token slot while keeping the supervision localized. The post-call part simulates the state after the external refinement model has inserted the corrected span. The corrected span is wrapped with <LLM_PATCH> and </LLM_PATCH>, and the loss is applied only to the remaining continuation and EOS token. This allows RG-Diff to learn not only when to ask for help, but also how to resume diffusion decoding after the refined span has been written back into the sequence.

Model and optimization. RG-Diff is initialized from GSAI-ML/LLaDA-8B-Instruct. We add special tokens to the tokenizer: <ASK>. We train a LoRA adapter on the attention and feed-forward projection modules, and additionally make the output head rows of the newly added special tokens trainable. All runs use bfloat16 training, gradient checkpointing, cosine learning-rate decay, and a 5% warmup ratio. The main training hyperparameters are summarized in Table 4.

Data filtering. For the main experiments, we use mask-right sanitized data. This preprocessing keeps examples whose local left context around the anchor is informative, while masking committed tokens immediately to the right of the anchor. The motivation is to prevent RG-Diff from relying on already-visible right-context templates instead of learning to place <ASK> at the true failure site. This filtering keeps most examples for code and math tasks, while producing smaller but cleaner training sets for GSM8K and commonsense reasoning.

D.2 Data Construction and Training of Stage II

In this section, we detail the data construction and training specifics for Stage I (RG-Diff).

D.2.1 Data Construction

Stage II trains the State-Conditioned Autoregressive Refinement (SC-AR) adapter to replace an emitted <ASK> token with a short corrective span. Unlike Stage I, where the supervision is derived

from annotated off-policy diffusion trajectories, the main Stage-II corpus is collected from the online distribution induced by the trained RG-Diff adapter. This design is important because SC-AR should see exactly the type of partial canvas it will receive at inference time: a state produced by the same dLLM backbone, the same RG-Diff adapter, and the same blockwise decoding schedule.

Online collection setup. We use `GSAI-ML/LLaDA-8B-Instruct` as the dLLM backbone. For data collection, the Stage-I RG-Diff checkpoint is loaded as a frozen PEFT adapter on top of this backbone, and the saved output-head rows for the new special tokens are restored before decoding. The resulting model is used only as a data-generating policy. It performs blockwise diffusion decoding with a maximum generation length of 256 tokens, commits the top 4 highest-confidence masked positions per step, and runs for at most 128 diffusion steps. At most three <ASK> events are handled for each problem. We do not use a soft threshold during data collection; a refinement event is recorded only when <ASK> is actually committed into the canvas.

The external teacher model is `Qwen/Qwen3-4B-Instruct-2507`, loaded in bfloat16 as a local causal language model. Whenever RG-Diff commits <ASK> at position f, we render the entire response canvas into a plain-text partial reasoning state. Visible tokens are decoded normally, masked spans are rendered with the mask placeholder, and the call site is rendered inline as <ASK>. This rendered canvas, together with the original math problem, is provided to Qwen under the same local-correction protocol used at inference time. The teacher is asked to output a short local formula patch rather than a full solution. The raw teacher response is stored for auditing, while a cleaned one-line patch is extracted and used as the training target.

Patch insertion and continuation. After a teacher patch is obtained, the data collection procedure immediately inserts it back into the dLLM canvas at the <ASK> position. Tokens after the inserted patch are reset to the diffusion mask token, and RG-Diff continues blockwise decoding from this repaired state. This produces a closed online loop: each subsequent <ASK> event is conditioned on all previous patches actually inserted into the canvas. Therefore, every recorded example reflects the state distribution that SC-AR will encounter when deployed inside the full FlexDLLM inference.

Each teacher call is saved as one Stage-II training record. The record contains a stable identifier, the source dataset and source index, the original question, the refinement round index, the diffusion step at which <ASK> fired, the <ASK> position, the rendered dLLM state with <ASK>, the raw Qwen output, the extracted patch target, the number of patch tokens, and diagnostic fields such as the total number of <ASK> events for that problem. If Qwen produces a duplicate patch that has already been used for the same problem, the collection script marks the record with `skipped_duplicate=true` and does not insert it into the canvas. These duplicate records are dropped by default during SC-AR training, because their targets do not correspond to patches that were actually applied to the dLLM.

Question selection and sharding. The collection script reads candidate problems from the previously constructed B1 file, deduplicates them by source dataset and source index, and selects up to 2000 unique questions. It prioritizes GSM8K examples and falls back to Hendrycks MATH examples if additional questions are needed. The full collection run is sharded across four GPUs. Each shard runs one frozen RG-Diff dLLM and one Qwen teacher on the same A6000 GPU in bfloat16. Records are flushed to JSONL periodically for crash safety and are merged into a single JSON file, `data/online_patches/online_patches.json`, after all shards finish. Since each problem may trigger zero, one, or multiple <ASK> events, the final number of Stage-II training records is determined by the number of teacher calls rather than by the number of original questions.

Training record encoding. The training dataset uses the online records directly. For each record, the input prompt is constructed from the original question and the full rendered dLLM canvas containing <ASK>. We intentionally keep the full canvas rather than only a short local window, so the SC-AR adapter can condition on both the left context and the visible right-side diffusion context. The prompt is formatted with the same Qwen-style local-correction protocol used during teacher collection; however, for efficiency the final training run uses the rules-only no-few-shot variant of this prompt. The prompt text is then rendered through the LLaDA tokenizer's chat template, so the trained SC-AR adapter sees the same conversational structure as the teacher model.

The target side is the extracted `patch_target`. Empty targets are discarded. Targets longer than the maximum patch budget are truncated, and targets shorter than the budget are padded with

EOS tokens. This EOS padding is intentional: it trains SC-AR to stop once the patch is complete. Examples whose prompt plus padded response exceed the maximum sequence length are dropped rather than prompt-truncated, because truncating the dLLM state could remove the local evidence around the <ASK> position. The resulting record contains only tokenized prompt ids, fixed-length response ids, the supervised patch length, and a truncation flag.

D.2.2 TRAINING SETTING

SC-AR is trained as a lightweight LoRA adapter on the same LLaDA backbone used by RG-Diff. Before attaching the SC-AR adapter, we first load the `GSAI-ML/LLaDA-8B-Instruct` backbone, register <ASK> as an additional special token, resize the embedding and output layers if necessary, and merge the trained Stage-I RG-Diff checkpoint into the base model. This merge bakes the RG-Diff LoRA deltas and the trained <ASK> output-head row into the shared backbone. After this step, all base parameters are frozen, and a fresh LoRA adapter named `patch_adapter` is attached for SC-AR training. We do not update the input embeddings or the language-model head during Stage II.

Single-token autoregressive blockwise training. Although the underlying model is a diffusion language model, SC-AR is trained with a single-token autoregressive blockwise interface. For each example, the prompt region is kept fully visible and never contributes to the loss. The response region has a fixed length of 30 tokens. At each training step, the collator samples one active response position j uniformly from this region. All tokens before j are exposed as a clean decoded prefix, the token at j is replaced by the diffusion mask token and supervised with cross-entropy, and all future response positions after j are also masked but receive no loss. Thus, each batch gives a one-token estimate of the left-to-right refinement factorization while still using the dLLM's masked-token interface. There is no diffusion-time sampling, no Bernoulli corruption, and no division by masking probability in Stage II.

At inference time, the same interface is used deterministically. SC-AR starts from a fixed number of masked patch slots, predicts the first token, writes it back, and repeats for the next slot. Decoding stops at the first EOS token or when the maximum patch length is reached. This makes the trained adapter a drop-in replacement for the Qwen teacher inside the online correction loop: RG-Diff detects <ASK>, SC-AR generates a short patch, the patch is inserted into the canvas, the subsequent positions are re-masked, and decoding switches back to RG-Diff.

E DETAIL EVALUATION SETTING

In this section, we provide a detailed description of our experimental setup, including the baselines, models, and test settings used.

Baseline. We compare FlexDLLM with six representative baselines that cover three complementary families of approaches for improving diffusion language model decoding. The first family consists of self-correction methods that allow a diffusion model to revise tokens that have already been generated. BackPlay v2 (Liu et al., 2026) is a head-only look-back self-correction method for diffusion language models. Its main motivation is that tokens decoded early in a parallel denoising trajectory may look locally plausible under sparse context but become inconsistent after later tokens are revealed. BackPlay freezes the underlying fine-tuned diffusion generator and trains only a lightweight correction head, avoiding generator drift and preserving the deployed model's original generation behavior. Its look-back correction mechanism constructs training examples in which predictions made under earlier, noisier contexts are judged under later, richer contexts; at inference time, the learned correctness scores are converted into a conservative remasking policy that periodically revisits previously generated tokens. ProSeCo (Schiff et al., 2026), or Progressive Self-Correction, takes a more integrated approach: it augments the standard masked diffusion objective with an auxiliary correction objective so that the same model learns both to fill masked tokens and to repair potentially incorrect visible tokens. During generation, ProSeCo interleaves ordinary unmasking steps with corrective refinement steps, enabling the model to update arbitrary positions in the sequence, including tokens that were committed in earlier denoising iterations. ReMeDi (Huang et al., 2025a) introduces remasking as an explicit self-reflective operation inside a mask-based diffusion language model.

Category	Stage-II SC-AR setting
Data construction	
Base dLLM for collection	`GSAI-ML/LLaDA-8B-Instruct`
RG-Diff checkpoint	`outputs/llada_ask_lora_v2_noclean/checkpoint-final`
Teacher LLM	`Qwen/Qwen3-4B-Instruct-2507`
Collection data source	B1 annotation file, deduplicated by source problem
Number of selected questions	2000
Collection sharding	4 shards on 4 A6000 GPUs
Max generated tokens	256
Diffusion commit size	4 tokens per step
Maximum diffusion steps	128
Maximum `<ASK>` patches per problem	3
Soft-`<ASK>` threshold	0.0; only committed `<ASK>` events are used
Teacher target	Extracted one-line local patch
Training data file	`data/online_patches/online_patches.json`
Duplicate teacher patches	Dropped by default during training
Dataset encoding	
Prompt format	Qwen-style local-correction prompt, no-few-shot variant
Conditioning state	Full rendered dLLM canvas with `<ASK>` and mask placeholders
Patch budget	30 tokens
Patch block size	1 token
Maximum sequence length	4096 tokens
Validation split / seed	5% / 42
Padding multiple	8
Model and adapter	
Training backbone	`GSAI-ML/LLaDA-8B-Instruct`
Inherited RG-Diff weights	Merged into the frozen base before SC-AR training
SC-AR adapter	Fresh LoRA adapter, `patch_adapter`
LoRA rank / alpha / dropout	32 / 64 / 0.05
LoRA target modules	`q_proj,k_proj,v_proj,attn_out,ff_proj,up_proj`
Train embeddings / LM head	No
Precision	bfloat16
Gradient checkpointing	Enabled
Optimization	
Number of GPUs	4 H200 GPUs with DDP
Epochs	3
Per-device train batch size	2
Gradient accumulation steps	16
Effective global batch size	128 examples
Learning rate	1×10^{-4}
Weight decay	0.0
Warmup ratio	0.05
Learning-rate schedule	Cosine
Maximum gradient norm	1.0
Logging interval	10 steps
Evaluation / save interval	20 steps / 20 steps
Save total limit	5 checkpoints
Random seed	42

Table 5: Key data-construction and training hyperparameters for Stage-II SC-AR. The effective global batch size is computed as per-device batch size $\times$ number of GPUs $\times$ gradient accumulation steps.

Instead of treating visible tokens as permanently fixed, ReMeDi jointly predicts token distributions and per-token confidence scores; low-confidence visible tokens can be remasked and resampled in later steps when more bidirectional context is available. The method is trained with a remask-aware pipeline, including supervised learning for detecting and remasking erroneous tokens and reinforcement learning for improving complete generation trajectories. ReMDM (Wang et al., 2025) is a more general inference-time remasking sampler for discrete diffusion models. It modifies the reverse sampling process so that already decoded tokens can transition back to the mask state with a controlled remasking probability, thereby restoring the iterative-refinement property that ordinary masked diffusion lacks. Unlike trainable correction-head methods, ReMDM can be applied on top of pretrained masked diffusion checkpoints and mainly changes the sampler rather than the

model parameters. Together, these four baselines test whether generic token revision and remasking mechanisms are sufficient to recover from early diffusion errors.

The second family is represented by LLaDOU (Huang et al., 2025b), a token-scheduling baseline. LLaDOU is built around the Diffusion Chain of Lateral Thought framework, which treats intermediate diffusion states as latent reasoning actions rather than merely as noisy partial strings. For masked diffusion models such as LLaDA, LLaDOU focuses on the order in which tokens are predicted and unmasked. It introduces an unmasking policy module that models token-selection order and is optimized with outcome-based reinforcement learning, so the model learns to choose more effective denoising trajectories for reasoning and code generation. This baseline is important because it improves the decoding policy without explicitly invoking an external refinement step: it asks whether better scheduling of the same diffusion model can match the gains obtained by FlexDLLM's explicit <ASK>-based intervention.

Finally, we include Vanilla SFT (Ouyang et al., 2022) as a standard supervised fine-tuning baseline. In this setting, the same backbone is trained on supervised instruction-response data using the conventional language-model fine-tuning objective, but no special intervention token, refinement gate, external helper, or remasking-based correction module is introduced. Vanilla SFT therefore isolates the gains from ordinary task adaptation and provides a reference point for measuring the added value of inference-time intervention. Compared with the above baselines, FlexDLLM differs in both the granularity and the source of correction: it does not globally resample previously decoded tokens or only optimize the unmasking order; instead, it learns to emit <ASK> at localized high-risk positions and then invokes a state-conditioned autoregressive refinement module to generate a short corrective patch conditioned on the current bidirectional diffusion canvas.

Models and Datasets All main experiments are conducted with LLaDA-8B-Instruct (Nie et al., 2025), an 8B-parameter diffusion language model trained under the masked-denoising paradigm and post-trained for instruction following. We choose LLaDA-8B-Instruct because it is a strong open diffusion-language-model backbone and naturally supports the bidirectional canvas representation used by FlexDLLM. For training, we combine four datasets that cover the major failure modes targeted by our method. GSM8K (Cobbe et al., 2021) provides grade-school mathematical word problems requiring multi-step arithmetic and natural-language reasoning, making it suitable for learning localized corrections to intermediate numerical computations. MATH (Hendrycks et al., 2021) contains more challenging competition-style problems with step-by-step solutions and boxed final answers, exposing the model to longer derivations, symbolic manipulation, and more difficult mathematical dependencies. Magicoder-OSS-Instruct (Wei et al., 2023) provides code-oriented instruction data generated from open-source code contexts, allowing the model to learn refinement behavior for programming tasks where an early incorrect statement can invalidate a later implementation. CommonsenseQA (Talmor et al., 2019) contributes multiple-choice commonsense reasoning examples that require background knowledge and answer-option discrimination, complementing the mathematical and coding datasets with shorter but semantically sensitive reasoning trajectories.

For evaluation, we use six benchmarks spanning three categories. For mathematical reasoning, we evaluate on GSM8K and MATH, which respectively test robust elementary arithmetic reasoning and more difficult competition-level problem solving. For code generation, we use HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021b). HumanEval measures functional correctness on Python programming problems specified by docstrings and checked with unit tests, while MBPP contains short entry-level Python programming tasks described in natural language and also evaluated through executable tests. These two benchmarks assess whether a method can maintain syntactic and semantic program correctness under diffusion decoding. For general reasoning, we evaluate on MMLU-Pro (Wang et al., 2024) and ARC-C (Clark et al., 2018). MMLU-Pro extends MMLU with more reasoning-focused questions and a larger answer-choice set, making it less susceptible to shallow memorization or random guessing. ARC-C, the Challenge subset of the AI2 Reasoning Challenge, consists of difficult grade-school science questions that were not solved by simple retrieval or co-occurrence baselines, and therefore tests knowledge-intensive reasoning beyond the training domains. This training/evaluation design lets us study whether FlexDLLM's localized intervention mechanism transfers from supervised construction data to held-out mathematical reasoning, executable code generation, and general multiple-choice reasoning benchmarks.

Evaluation Setting Scoring. All scores are produced by the OpenCompass evaluator in baselines/common/evaluate.py using the official metric for each task: exact-match accuracy on numerical answers (GSM8K, MATH, AMC23) extracted from boxed, multiple-choice letter accuracy (MMLU-Pro, ARC-Challenge), and pass-rate on unit tests via human_eval / mbpp (HumanEval, MBPP). Per-task results are written to baselines/results/<method>/<task>.score.json and aggregated into the main table by baselines/common/aggregate_full.py.

F Discussion and Future Work

FlexDLLM opens up several promising directions for further improving hybrid diffusion–autoregressive inference. One natural extension is to make the refinement policy more adaptive, allowing the model to dynamically adjust when, where, and how much autoregressive computation to allocate according to the difficulty of the current generation state. Beyond token-level triggering, future versions could incorporate span-level or reasoning-step-level refinement, enabling FlexDLLM to handle more complex dependencies with fewer mode switches. Another interesting direction is to combine SC-AR with external verifiers, reward models, or process supervision, so that refinement is guided not only by local decoding uncertainty but also by global correctness signals. FlexDLLM may also benefit from richer training objectives that jointly optimize diffusion efficiency and autoregressive correction quality, encouraging the two modes to cooperate more tightly. Finally, we plan to extend FlexDLLM to longer-context generation, code synthesis, mathematical reasoning, and agentic tasks, where selective sequential refinement over a largely parallel decoding process could provide an even stronger balance between inference efficiency and generation quality.