FLARE: Diffusion for Hybrid Language Model

Yuchen Zhu[1,2,‡] **Jing Shi**[1,‡] **Chongjian Ge**[1] **Hao Tan**[1] **Yiran Xu**[1]
Wanrong Zhu[1] **Jason Kuen**[1] **Ryan A. Rossi**[1] **Koustava Goswami**[1]
Rajiv Jain[1] **Yongxin Chen**[2] **Molei Tao**[2] **Jiuxiang Gu**[1,‡,†]

[1]Adobe Research [2]Georgia Institute of Technology
[‡]Core contributors [†]Project lead

Project page: https://tokflare.github.io
Code: https://github.com/yuchen-zhu-zyc/HybridDiffusion
Checkpoints: https://huggingface.co/collections/yuchen-zhu-zyc/hybrid-diffusion

Abstract

Autoregressive (AR) large language models (LLMs) have achieved broad practical success, but sequential decoding remains a key bottleneck for low-latency deployment. Recent efficient-inference work has progressed along two axes: reducing the cost of each model invocation through efficient architectures, and reducing serial decoding steps through parallel generation. Hybrid attention backbones address the former, while diffusion language models (dLLMs) pursue the latter via iterative parallel denoising. Combining these advantages remains challenging: AR-to-dLLM conversion often fails to preserve seed-checkpoint capability, and hybrid-attention recurrent states and masking constraints make diffusion training and serving nontrivial. We present **FLARE**, a systematic conversion framework for hybrid-attention LLMs. Our analysis identifies transfer data quality as the primary determinant of capability preservation, outweighing loss formulation and attention-mask design. The resulting framework combines a token-equal AR-and-diffusion objective, hardware-aware kernels, and unified inference, enabling one checkpoint to support both AR-style verified decoding and diffusion-style parallel denoising. Starting from strong AR checkpoints with limited post-training data, **FLARE** is competitive with leading open-source dLLMs across model scales and delivers consistent throughput gains over open-source dLLM baselines in single-GPU concurrent serving. Our results further suggest that practical dLLMs are limited not only by decoding algorithms, but also by transfer data quality and the training inefficiency of current block-diffusion objectives, motivating joint design of data, objectives, architectures, and inference systems.

1 Introduction

Modern large language models (LLMs) (Singh et al., 2025a; Anthropic, 2026; The Gemini Team, 2026) have become increasingly central to interactive, agentic, and embodied AI systems, spanning general-purpose assistants, personal agents (OpenClaw, 2026), embodied AI, robotics control, and autonomous driving (Jiang et al., 2026; Yang et al., 2026a; Jiang et al., 2025; Ma et al., 2025). As these models move toward end-user, edge-device, and real-time service deployment, inference must preserve model capability while satisfying system constraints on latency, throughput, memory footprint, and energy consumption (Zheng et al., 2024; Kwon et al., 2023; ggml-org, 2026). These constraints are especially pronounced in latency-sensitive applications and directly affect deployability on personal devices, edge hardware, and closed-loop decision-making systems. Consequently, serving efficiency and inference latency have become central bottlenecks for practical AR LLM deployment.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

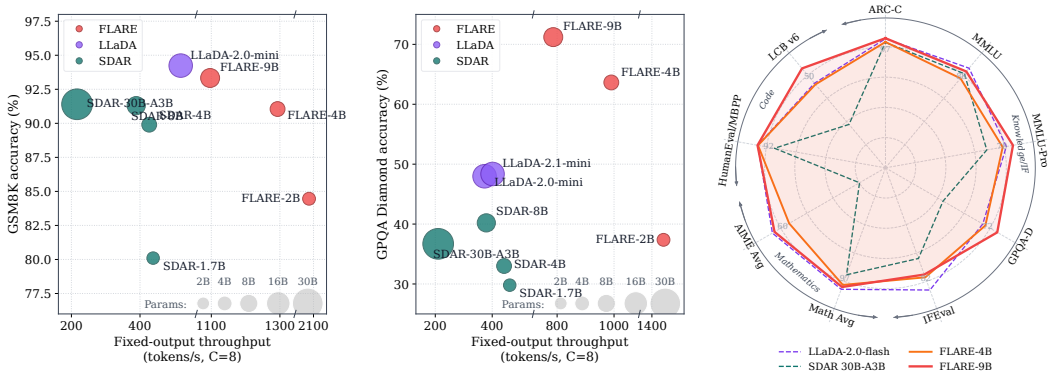


Figure 1: **FLARE** is a state-of-the-art dLLM with high sampling throughput and near AR performance.

Efforts to improve LLM serving efficiency have proceeded along two complementary directions: making each model invocation cheaper, and reducing the number of serial invocations. The first direction relies on architectural changes. Compressed or sparse variants of softmax attention (Vaswani et al., 2017; Ainslie et al., 2023; Liu et al., 2024, 2025b) and linear-attention or related recurrent-memory modules (Katharopoulos et al., 2020; Schlag et al., 2021; Gu and Dao, 2023; Dao and Gu, 2024; Yang et al., 2024a) reduce the storage and computation required by long-context and incremental decoding. Since purely linear memory can trade off retrieval fidelity, recent efficient LLMs increasingly use hybrid-attention backbones that combine softmax and linear-attention layers, balancing model capability, cache cost, and serving efficiency (Li et al., 2025a; Team et al., 2025; Basant et al., 2025; Blakeman et al., 2025b,c; Qwen Team, 2025, 2026). Hybrid attention lowers per-forward cost, but does not remove the serial nature of AR generation. The second direction reduces serial decoding steps. Multi-token prediction and speculative decoding retain the AR factorization while advancing multiple tokens per round through auxiliary heads, drafting, or parallel verification (Gloeckle et al., 2024; Li et al., 2025c; Cai et al., 2024; Leviathan et al., 2023; Kumar et al., 2026). Diffusion large language models (dLLMs) go further by relaxing strict next-token generation and using iterative parallel denoising, offering a distinct route to lower decoding latency (Khanna et al., 2025; Song et al., 2025; Google DeepMind, 2025; Wang et al., 2025c; Chen et al., 2026). As illustrated in Figure 5 (Appendix B), these trends have largely evolved in parallel in the open-source LLM ecosystem, with hybrid-attention AR models improving architectural efficiency and dLLMs improving generation-side parallelism.

Research on dLLMs has advanced diffusion-style objectives, masked or uniform diffusion, block diffusion, and large-scale diffusion LMs (Lou et al., 2023; Sahoo et al., 2024; Shi et al., 2024; Ou et al., 2024; Arriola et al., 2025; Sahoo et al., 2025; Nie et al., 2025; Bie et al., 2025), with recent studies beginning to characterize their scaling and data-efficiency behavior (Ni et al., 2025b; Sahoo et al., 2026; Ni et al., 2025a). In parallel, AR-initialized transfer and related conversion recipes obtain parallel-generation capability from existing AR checkpoints through continued training, shifted-token losses, block-diffusion training, or lightweight post-training (Gong et al., 2024; Fu et al., 2025; Cheng et al., 2025; Liu et al., 2025c,a; Yu et al., 2026). Nevertheless, AR-to-dLLM transfer remains at an early stage: the roles of *transfer-stage data*, *scaling behavior*, and *joint loss–mask design* in capability preservation remain poorly understood. As a result, existing conversion methods often struggle to robustly inherit the capabilities of their AR seed checkpoints. At the same time, higher tokens-per-forward does not automatically translate into higher tokens-per-second; realized serving throughput depends on inference-system support for prefix reuse, KV or recurrent-state caching, and low-overhead parallel decoding (Wu et al., 2025b; Fu et al., 2025; Chen et al., 2026). On modern hybrid-attention backbones, these issues become more complex because diffusion training and inference must also handle recurrent-state handoff, mask realization, and serving-path design. How to convert a strong AR model into a capable and serving-efficient hybrid-backbone dLLM under a limited training budget therefore remains an open problem. A detailed discussion of related work is provided in Appendix B.

Motivated by these challenges, we present **FLARE**, a systematic recipe for converting hybrid-attention AR LLMs into capable and serving-efficient dLLMs under a practical training budget. **FLARE** examines the full conversion pipeline, from transfer-data construction and loss–mask design to hybrid-backbone training and serving-time inference. Concretely, **FLARE** makes three contributions. (1) **Transfer diagnosis.** To understand the performance degradation observed in prior AR-to-dLLM methods, we conduct controlled ablations over transfer data, objective design, and

attention-mask choices, and identify transfer-data quality and distribution match as the dominant factors for preserving AR capability, outweighing loss formulation and mask design. This addresses a factor that has often been under-discussed in prior transfer work (Cheng et al., 2025; Wu et al., 2025a; Fu et al., 2025; Liu et al., 2025c). (2) **Efficient training.** To enable diffusion training on modern softmax-plus-linear-attention backbones, we develop hardware-aware algorithms for computing linear attention under diffusion-specific visibility patterns (Yang et al., 2024a; Dao and Gu, 2024; Team et al., 2025), realizing these patterns through recurrent-state handoff rather than dense masking. (3) **Efficient generation.** To turn parallel-generation capability into wall-clock throughput, we build a unified inference system that supports both AR-style verified decoding and diffusion-style parallel denoising from the same checkpoint, extending efficient LLM serving ideas (Zheng et al., 2024) to hybrid-backbone dLLMs.

Taken together, the study yields four main takeaways, expanded in Appendix C.1: 1) **objective and mask design matter for capability preservation**; 2) **transfer data explains the remaining variation**, making AR-SFT a useful low-cost proxy for data selection; 3) **hybrid-backbone diffusion requires specialized training** through recurrent-state scheduling and hardware-aware training kernels; and 4) **one checkpoint supports two generation regimes**: starting from Qwen3.5 hybrid-attention checkpoints (Qwen Team, 2026), **FLARE**-2B/4B/9B achieves competitive diffusion-LM quality while supporting both AR-style verified speculative decoding and diffusion-style parallel denoising (Figure 1).

2 Method

2.1 Preliminaries on Attention Mechanisms

To ground the subsequent analysis of hybrid-attention backbones and non-causal visibility patterns, we view softmax and linear attention as *associative memories* (Zhong et al., 2025; Wang et al., 2025b) that share a memory–retrieval interface but differ in how past key-value pairs are stored and queried (full treatment in Appendix D.1). Each layer receives per-token queries and keys $\mathbf{q}_t, \mathbf{k}_t \in \mathbb{R}^{d_k}$ and values $\mathbf{v}_t \in \mathbb{R}^{d_v}$, and emits an output $\mathbf{o}_t \in \mathbb{R}^{d_v}$. A softmax layer explicitly stores all visible key-value pairs in a KV cache and retrieves by normalizing query-key similarities over a visible index set $\mathcal{V}_t$, which provides accurate token-level and long-range retrieval, but its KV-cache footprint grows as $\mathcal{O}(T(d_k + d_v))$. Linear attention (Katharopoulos et al., 2020; Choromanski et al., 2020) instead replaces the explicit KV cache with a bounded-size recurrent state $\mathbf{S}_t \in \mathbb{R}^{d_k \times d_v}$. We use Gated DeltaNet (GDN) (Yang et al., 2024a) as the representative instantiation; its memorization and retrieval rule, the Gated Delta Rule (GDR), is

$$\mathbf{S}_t = \alpha_t \left(\mathbf{I} - \beta_t \, \mathbf{k}_t \mathbf{k}_t^\top\right) \mathbf{S}_{t-1} + \beta_t \, \mathbf{k}_t \mathbf{v}_t^\top, \qquad \mathbf{o}_t = \mathbf{S}_t^\top \mathbf{q}_t / \sqrt{d_k}. \tag{1}$$

Hybrid-attention backbones combine these mechanisms at the architecture level: softmax layers provide high-fidelity access to selected context, whereas recurrent-memory layers act as efficient compressed memory (Dong et al., 2025; Glorioso et al., 2024; Li et al., 2025a; Qwen Team, 2025, 2026; Blakeman et al., 2025c; Team et al., 2025). Our main **FLARE** instantiation follows this setting. A hybrid backbone contains both explicit key-value memory and recurrent-state memory, so the same non-causal visibility pattern must be implemented through different mechanisms across layer types; the next subsection instantiates this issue for the clean/noisy training mask.

2.2 Training Objective and Mask Design

Let $\mathbf{x} = (x^1, \ldots, x^L) \in \mathcal{V}^L$ be a training sample partitioned into K contiguous blocks $\mathbf{x}_b = (x^{(b-1)B+1}, \ldots, x^{bB})$ of size B ($L = KB$); we write $\mathbf{x}^{<b}$ for all tokens strictly before block b. For each block, we sample a masked subset $\mathcal{M}_b \subseteq \{(b-1)B+1, \ldots, bB\}$ and form the noisy block $\tilde{\mathbf{x}}_b$ by replacing x^ℓ with `[MASK]` for $\ell \in \mathcal{M}_b$. The complement $\mathcal{M}_b^c$ defines a second, disjoint noisy view $\tilde{\mathbf{x}}_b^c$. Based on this block-level corruption process, **FLARE** maximizes a token-balanced clean/noisy objective that preserves the AR path while adding diffusion-style supervision. The clean stream follows the original causal order and provides the next-token prediction term $\mathcal{L}_{\mathrm{AR}}$. The noisy stream provides the diffusion term $\mathcal{L}_{\mathrm{diff}}$: the two complementary noisy views predict disjoint token subsets, both conditioned on the same preceding clean context. The overall objective is

$$\mathcal{L}_{\textbf{FLARE}}(\theta) = \mathcal{L}_{\mathrm{AR}}(\theta) + \mathcal{L}_{\mathrm{diff}}(\theta). \tag{2}$$

Concretely, the clean-stream term is the usual next-token log-likelihood over the full sequence, and the noisy-stream term sums the denoising log-likelihood over the two complementary masked views:

$$\mathcal{L}_{\mathrm{AR}}(\theta) = \sum_{\ell=1}^{L} \log p_\theta\left(x^\ell \mid \mathbf{x}^{<\ell}\right),$$

$$\mathcal{L}_{\mathrm{diff}}(\theta) = \sum_{b=1}^{K} \left[\underbrace{\sum_{\ell \in \mathcal{M}_b} \log p_\theta\left(x^\ell \mid \tilde{\mathbf{x}}_b, \mathbf{x}^{<b}\right)}_{\text{primary noisy view}} + \underbrace{\sum_{\ell \in \mathcal{M}_b^c} \log p_\theta\left(x^\ell \mid \tilde{\mathbf{x}}_b^c, \mathbf{x}^{<b}\right)}_{\text{complementary noisy view}} \right]. \tag{3}$$

The clean/noisy construction in Figure 2 realizes the two terms in Eq. (3): the causal clean stream supplies $\mathcal{L}_{\mathrm{AR}}$, while the block-bidirectional noisy stream supplies $\mathcal{L}_{\mathrm{diff}}$ under two complementary views. This differs from standard mask-dLLM training (Arriola et al., 2025; Nie et al., 2025; Cheng et al., 2025), where masked positions are predicted only from noisy contexts. In **FLARE**, every token contributes one AR signal and one diffusion signal at unit weight because $\mathcal{M}_b$ and $\mathcal{M}_b^c$ partition each block. This keeps the two supervision sources balanced under varying mask fractions while giving the same checkpoint both causal generation and block-level parallel denoising capabilities. Figure 2 shows the document-packed clean/noisy visibility pattern used to compute the objective above in a packed forward pass: the clean stream remains token-causal and isolated from the noisy stream, each noisy block is bidirectional within the block and attends to preceding clean context, and document boundaries isolate packed samples to prevent cross-document leakage while avoiding padding overhead.

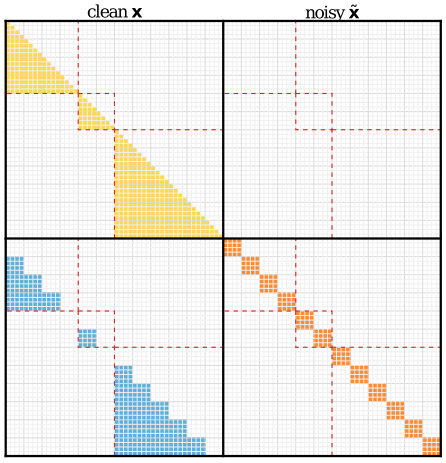


Figure 2: **FLARE** document-packed clean/noisy mask: causal clean attention (yellow), bidirectional noisy blocks (orange), noisy-to-clean visibility (blue), and document boundaries (red dashed).

Algorithmically, this construction is closest to TiDAR (Liu et al., 2025c) and concurrent work I-DLM (Yu et al., 2026), but differs in two choices that matter for decoding. Unlike TiDAR, we apply logit shift to the noisy-stream diffusion terms to align them with AR semantics. Unlike I-DLM, we keep the noisy stream block-bidirectional and use random rather than fully masked noisy views during training, which preserves Diffusion-Trust decoding in addition to AR-style decoding.

The two memory implementations in a hybrid backbone realize this same pattern differently. In softmax layers, an additive mask directly removes forbidden entries from the visible set $\mathcal{V}_t$; in recurrent-memory layers, the same non-causal clean/noisy mask becomes a schedule of writes, state propagation, and state resets rather than a post-hoc masking matrix. For a GDN layer, the clean stream follows the standard causal recurrence of Eq. (1), while each noisy block b is initialized from the preceding clean boundary state $\mathbf{S}_{(b-1)B}$, updated only with noisy tokens from the same block, and read out from its block-end state. This block-end readout realizes bidirectional visibility within the noisy block, and resetting the noisy state to the corresponding clean boundary state prevents information flow across noisy blocks (Appendix D.2). The same state-scheduling idea extends to other linear-attention variants such as Mamba-2 (Dao and Gu, 2024) and Kimi Delta Attention (Team et al., 2025); its chunk-parallel realization is discussed in Section 4.2.

2.3 Decoding Interfaces

The clean/noisy objective in Section 2.2 gives a trained **FLARE** checkpoint two usable output distributions under the same forward interface: clean-stream logits preserve causal AR prediction, while noisy-stream logits denoise a bidirectional block conditioned on clean context. This separation is important for transfer: the seed checkpoint's capability is organized around next-token prediction, and a pure diffusion path would need to replace that causal distribution under limited post-training

data. The clean stream therefore anchors the transferred AR capability and provides a verification reference, while the noisy stream adds block-level parallel denoising. This yields two decoding interfaces. In *Diffusion-Trust*, noisy-stream samples are the reference and are committed through block-denoising confidence. In *AR-Trust*, noisy-stream samples serve as drafts and clean-stream logits verify them left-to-right. Algorithms 9 and 10 in Appendix E.1 give the operational view; below, we state the corresponding commitment and verification rules.

We use x for the input prompt, y for the generated sequence, f_θ for the trained checkpoint's forward map, z for pre-softmax logits, and $\sigma_{k,p,T}$ for temperature-scaled top-k/top-p sampling. Diffusion-Trust uses noisy-stream distributions π_i; AR-Trust uses clean-stream target distributions p_i, noisy proposal laws q_i, and draft tokens d. Decoding schematics and path-specific masks are provided in Appendix E, while system-level serving details are discussed in Section 4.3.

Diffusion-Trust: block iterative denoising and noisy-stream commitment. Consider an active block $\mathcal{B}_{t,B} = \{t, \dots, t+B-1\}$ behind the causal prefix, decoded over steps $s = 1, \dots, S$, and let R_s be its unresolved positions before step s ($R_1 = \mathcal{B}_{t,B}$). For each $i \in R_s$, the noisy stream defines $\pi_i = \sigma_{k,p,T}(\tilde{z}_i^{\text{shift}})$, samples $\hat{y}_i \sim \pi_i$, and computes a confidence score $c_i = \text{score}(\pi_i, \hat{y}_i)$. Since the noisy stream is trusted in this path, selected candidates are written to y without clean-stream verification:

$$A_s = \{i \in R_s : c_i \geq \gamma_s\} \ (s < S), \quad A_S = R_S, \quad y_i \leftarrow \hat{y}_i \ (i \in A_s), \quad R_{s+1} = R_s \setminus A_s, \tag{4}$$

so the block is finalized after at most S steps; Eq. (4) covers confidence-based (Nie et al., 2025) and margin-based (Kim et al., 2025) unmasking by changing how c_i is computed. Because intermediate denoising forwards contain provisional block contents, their recurrent states are not written into the live causal state; once the block is finalized, a causal replay commits it to the hybrid backbone state (Appendix E.3).

AR-Trust: speculative verification. At verify position i, let $p_i = \sigma_{k,p,T}(z_i)$ be the clean-stream target distribution and q_i the proposal law that first sampled the held draft token d_i. The reference verifier is

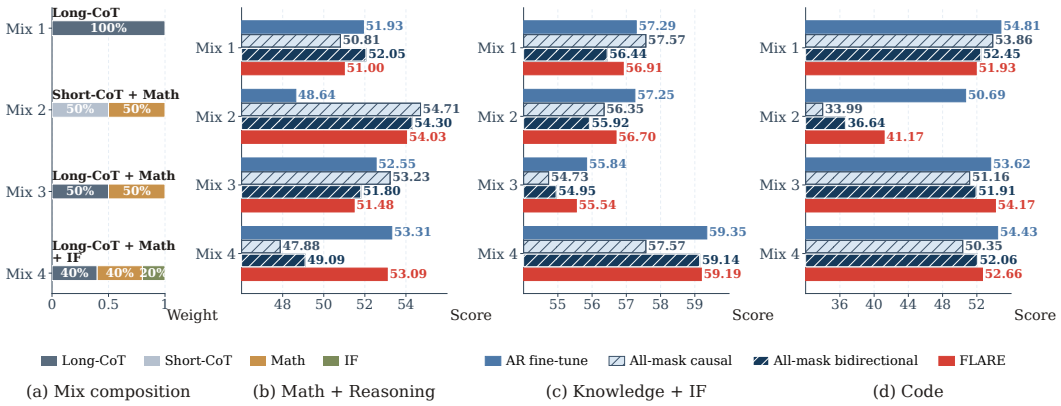
$$a_i \sim \text{Bern}\Big(\min\Big\{1, \tfrac{p_i(d_i)}{q_i(d_i)}\Big\}\Big), \qquad y_i = \begin{cases} d_i, & a_i = 1, \\ y^\star \sim \text{Norm}_{\mathcal{V}}([p_i - q_i]_+), & a_i = 0, \end{cases} \tag{5}$$

where a_i is the accept indicator, $[t]_+ = \max\{t, 0\}$, and $\text{Norm}_{\mathcal{V}}$ normalizes over the vocabulary. The exactness condition is that q_i must be the same proposal law that sampled d_i, since noisy proposals are generated in parallel under the clean/noisy block mask; lower-overhead policies may approximate q_i or use argmax drafts, trading strict equivalence for serving efficiency (Appendix E.2). The resulting ideal speedups are analyzed in Appendix E.1 (Figure 15): AR-Trust is governed by the acceptance rate α and horizon K, and Diffusion-Trust by the block-to-step ratio B/S.

3 Experimental Analysis

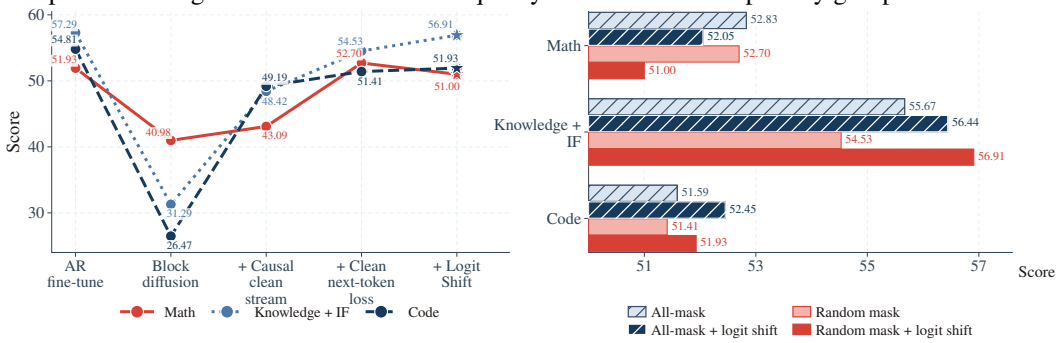
We use this section to identify the key algorithmic and data ingredients of AR-to-dLLM transfer. While existing conversion methods (Cheng et al., 2025; Wu et al., 2025a; Fu et al., 2025) report a range of transfer outcomes, several design choices, from the training objective and clean/noisy visibility pattern to transfer-data composition, have not yet been examined systematically and in isolation. We therefore start from a smaller model, Qwen3-1.7B, and conduct a controlled study under a fixed training budget and evaluation suite, varying algorithmic ingredients and transfer data separately, so as to surface the recipe components that drive successful transfer. The resulting recipe is then carried over to stronger hybrid-attention models in Section 4.

All transfer runs initialize from the same pretrained Qwen3-1.7B checkpoint (Yang et al., 2025). In our SFT format, the prompt tokens act as the leading clean-stream context (visible through attention but excluded from the loss), while the assistant response tokens are partitioned into blocks of size B that simultaneously receive the $\mathcal{L}_{\text{AR}}$ and $\mathcal{L}_{\text{diff}}$ terms defined in Section 2.2 — i.e., the clean/noisy dual-stream supervision is applied only on response tokens. All runs use a maximum sequence length of 4096 with the document-level packing mask of Figure 2, a global batch size of 256, and 9000 optimizer steps, corresponding to roughly 10B training tokens. We evaluate all checkpoints on the same 12-task suite, covering Math + Reasoning (GSM8K (Cobbe et al., 2021), MATH-500 (Hendrycks et al., 2021), AIME 24/25 (Mathematical Association of America,



(a) Mix composition (b) Math + Reasoning (c) Knowledge + IF (d) Code

Figure 3: Transfer performance across data compositions and capability groups. Panel (a) shows the per-mix sampling weights (mix composition); panels (b)–(d) plot scores for all four transfer recipes under the same Qwen3-1.7B seed and training budget on Math + Reasoning, Knowledge + Instruction Following, and Code, respectively, with a local x-axis range for readability. The trends show that converted dLLMs largely track their AR fine-tuning counterparts under the same data mix, while data composition changes the attainable transfer quality across the three capability groups.



(a) Incremental recipe ablation (b) Logit-shift ablation

Figure 4: Controlled ablations of AR-to-dLLM transfer ingredients on the fixed Mix 1 (`Long-CoT`) data condition. **(a)** The left panel follows a cumulative recipe: AR fine-tuning, pure block-diffusion transfer, adding a causal clean stream, adding clean-stream next-token loss, and adding logit-shifted noisy supervision. **(b)** Logit shift has a limited effect on category-level scores under both all-masked and random noisy masking, while the random noisy mask preserves Diffusion-Trust decoding.

2024), ARC-Challenge (Clark et al., 2018), GPQA-Diamond (Rein et al., 2023)), Knowledge + IF (MMLU (Hendrycks et al., 2020), MMLU-Pro (Wang et al., 2024), IFEval (Zhou et al., 2023)), and Code (HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), LiveCodeBench v6 (Jain et al., 2024)). All checkpoints share one decoding protocol; detailed settings are in Appendix G.

3.1 Effects of Transfer Data Composition

We first study transfer data because it determines the regime in which the algorithmic ablations should be interpreted. We construct four controlled transfer-data mixes from curated open-source post-training SFT corpora released with the Llama-Nemotron (Bercovich et al., 2025), Nemotron Nano 2 (Basant et al., 2025), Nemotron 3 (Blakeman et al., 2025b), and Nemotron-Cascade-2 (Yang et al., 2026b) model families; the detailed curation, filtering, packing, and mixture-sampling procedure is provided in Appendix F. The mixes vary reasoning-trace length, math coverage, and instruction-following coverage while keeping the seed checkpoint, training budget, and evaluation protocol fixed.

Concretely, built from long chain-of-thought, short-reasoning, math, and instruction-following pools: **Mix 1** (`Long-CoT`) serves as a reasoning-heavy baseline; **Mix 2** (`Short-CoT+Math`) separates the

effect of trace length from math coverage; **Mix 3** (`Long-CoT+Math`) tests whether stronger math coverage helps without shortening traces; and **Mix 4** (`Long-CoT+Math+IF`) further adds instruction-following data. Sources, per-mix sampling weights (Figure 3(a)), and an automatic instance-level data-selection pipeline are detailed in Appendix F.

Figure 3(b)–(d) compares these mixes across four transfer recipes under the same Qwen3-1.7B seed and training budget. AR fine-tuning (AR-SFT) denotes continued next-token fine-tuning on response tokens; all-mask causal (AM-Causal) and all-mask bidirectional (AM-Bidir) share the joint AR/diffusion supervision used by **FLARE** but replace random noisy masking with all-masked noisy views, differing only in whether the noisy stream is token-causal or bidirectional within each block. On the recipe axis, the all-mask variants tend to trail both AR fine-tuning and **FLARE**, most visibly on Code, while AR fine-tuning and **FLARE** track each other closely within each mix; on the data axis, no single data change is uniformly beneficial, and Mix 4 gives the strongest Knowledge + Instruction Following results while keeping Math + Reasoning and Code competitive. Two conclusions follow (extended analysis in Appendix G.2.3). First, conditional on a properly aligned clean/noisy objective, the attainable transfer quality is governed primarily by the transfer data mix rather than by the algorithmic recipe, so AR fine-tuning provides a low-cost proxy for screening data mixes prior to dLLM conversion. Second, comparisons of objective and attention-mask choices across different mixes are confounded by data quality; Section 3.2 therefore fixes Mix 1 (`Long-CoT`), while Section 4 adopts Mix 4 (`Long-CoT+Math+IF`) for the final hybrid-checkpoint conversions.

3.2 Objective and Attention-Mask Ablations

After isolating the data effect in Section 3.1, we fix the transfer data condition to Mix 1 (`Long-CoT`) and ask which objective and attention-mask ingredients prevent AR-to-dLLM conversion from losing the seed model's capability, treating document-level packing as a fixed efficiency condition rather than an ablated factor.

Figure 4(a) studies the clean-stream side through a cumulative recipe. Starting from AR fine-tuning, we replace next-token fine-tuning with a pure block-diffusion objective following the spirit of BD3-LM (Arriola et al., 2025) and SDAR (Cheng et al., 2025), then add back a token-causal AR clean stream, an auxiliary next-token-prediction loss on clean-stream logits, and the logit-shifted noisy-stream supervision used by **FLARE**. The results show that pure block-diffusion transfer alone is not capability-preserving: it substantially degrades all three capability groups (averaging -21.8 points relative to AR fine-tuning, with the heaviest drops on Code and Knowledge + IF). The largest single-step recovery comes from restoring a token-causal AR clean stream ($+14.0$ points on average), which alone closes most of the Code and Knowledge + IF gap. Adding the clean-stream next-token-prediction loss further pulls Math + Reasoning back to the AR fine-tuning level by anchoring the converted model to the pretrained checkpoint's AR semantics, and logit shift on top yields a near-saturated final score.

Figure 4(b) isolates the noisy-stream side: at fixed AR-aligned clean-stream supervision, all-masked and randomly masked noisy views yield comparable category-level scores, so noisy-mask sampling is not a primary driver of benchmark recovery. Its role is instead decoding compatibility: random masking trains the noisy stream as a genuine denoising path and preserves Diffusion-Trust decoding, while logit shift aligns the block-diffusion logits with inference-time token positions. Capability-preserving AR-to-dLLM conversion is therefore governed primarily by clean-stream alignment; combined with Section 3.1, this isolates the residual transfer gap to the quality of the data mix.

4 Hybrid-Backbone Conversion and Evaluation

Section 3 decomposed AR-to-dLLM transfer into two complementary parts: an AR-aligned clean stream that determines whether capability is preserved, and a noisy-stream design that determines whether the converted checkpoint remains compatible with diffusion-style parallel decoding. This section applies the resulting configuration to strong hybrid-attention backbones and asks three concrete questions: (i) whether the transfer configuration selected by the controlled study scales to Qwen3.5 checkpoints (Qwen Team, 2026) so that the converted models retain the capability of their AR source; (ii) whether two-stream supervision can be executed on a hybrid-attention backbone at hardware efficiency close to pure-AR training; and (iii) whether a single checkpoint can support both

AR-Trust and Diffusion-Trust decoding under one unified serving stack. The three questions are addressed in turn in the subsections that follow.

We train **FLARE**-2B/4B/9B from the post-trained Qwen3.5-2B/4B/9B AR checkpoints, and compare the converted models against a set of leading dLLM systems: the commercial Mercury-2 (Ermon, 2026), the LLaDA-2.0/2.1 mini and flash families (Bie et al., 2025, 2026), and the SDAR family (Cheng et al., 2025). All three **FLARE** checkpoints are trained in a single supervised-finetuning stage on Mix 4 (`Long-CoT+Math+IF`), the data condition selected in Section 3.1. We keep the main protocol aligned with Section 3: sequence length 4096, document-level packing masks, global batch size 256, and 9000 optimizer steps, corresponding to roughly 10B training tokens. The objective is the token-balanced AR/diffusion loss $\mathcal{L}_{\text{FLARE}}$ in Eq. (2), with block size $B = 4$. This conversion budget is markedly more modest than prior AR-to-dLLM transfer work, which typically uses 50B–200B tokens (Cheng et al., 2025; Fu et al., 2025). Evaluation follows the 12-benchmark protocol from Section 3; we report **FLARE** under both AR-Trust and Diffusion-Trust sampling, with the original Qwen3.5 checkpoints serving as AR source-model references. The LLaDA and SDAR rows are taken from their technical reports, Mercury-2 is evaluated through its official API, and the Qwen3.5 and **FLARE** rows are measured under our SGLang-based inference stack (Zheng et al., 2024).

4.1 Capability Retention Across Model Scales

We first evaluate whether the converted checkpoints preserve the capability of their AR initialization. Tables 1 and 2 report results at 9B and at smaller scales, with the original Qwen3.5 checkpoints serving as source-model references and the dLLM systems listed above as external baselines.

Table 1: **Benchmark performance of FLARE-9B**. **Bold**: best open-source dLLM; underline: second best; *: potentially under-reported.

	Mercury-2	**LLaDA-2.0 -mini**	**LLaDA-2.1 -mini**	**LLaDA-2.0 -flash**	**LLaDA-2.1 -flash**	**SDAR 30B-A3B**	**FLARE -9B**	**FLARE -9B**	**Qwen3.5 -9B**
Params	*Commercial*	*16B-A1B*	*16B-A1B*	*100B-A5B*	*100B-A5B*	*30B-A3B*	*9B*	*9B*	*9B*
Sampling mode	*Diffusion*	*Diffusion*	*Diffusion*	*Diffusion*	*Diffusion*	*Diffusion*	*AR-Trust*	*Diffusion-Trust*	*AR*
Knowledge & Instruction Following									
ARC-Challenge	—	93.56	—	95.93	—	93.2	**96.33**	95.65	97.70
MMLU	—	80.53	—	**87.69**	—	82.8	84.80	80.75	88.21
MMLU-Pro	—	63.22	63.42	73.36	75.31	61.5	**77.39**	74.73	81.39
GPQA-Diamond	73.00	47.98	48.36	61.98	66.67	36.7	**71.21**	64.65	80.30
IFEval	—	80.78	81.33	81.70	**83.36**	60.6	71.35	63.22	91.31
Math									
GSM8K	90.62	94.24	—	**96.06**	—	91.4	93.33	93.10	89.16
MATH-500	81.00	—	—	—	—	77.8	**95.20**	93.60	96.60
AIME-24	51.10	—	—	—	—	16.7	**63.33**	60.00	65.56
AIME-25	—	36.67	36.67	60.00	**63.33**	10.8	54.44	53.33	60.00
Code									
HumanEval	—	86.59	—	**94.51**	—	87.2	92.07	82.32*	95.12
MBPP	—	81.50	—	88.29	—	71.6	**91.05**	82.10*	89.11
LiveCodeBench v6	67.30	31.50	28.85	42.29	44.05	21.7	**49.71**	5.71*	49.71

At the 9B scale, **FLARE**-9B matches or exceeds LLaDA-2.1-flash on most shared benchmarks despite using roughly $1/10$ of its total parameters, e.g., GPQA-Diamond (71.21 vs. 66.67), MMLU-Pro (77.39 vs. 75.31), and LiveCodeBench v6 (49.71 vs. 44.05), and is clearly stronger than the commercial Mercury-2 on math reasoning (MATH-500 by $+14.2$ points, AIME-24 by $+12.2$ points), while trailing it on LiveCodeBench v6. Relative to the source-model Qwen3.5-9B, **FLARE**-9B retains the majority of the original capability across most tasks, e.g., 98.6% on MATH-500 and 95.1% on MMLU-Pro, while *exceeding* the original AR checkpoint on MBPP (91.05 vs. 89.11). The same pattern extends to the smaller scales: **FLARE**-4B outperforms SDAR-30B-A3B on nearly all reported benchmarks (e.g., AIME-24: 58.89 vs. 16.7), and **FLARE**-2B improves substantially over the parameter-matched SDAR-1.7B on math and code (e.g., MATH-500: 84.40 vs. 63.2). Extended discussion is given in Appendix G.2.5.

Each **FLARE** row reports a single trained checkpoint under both AR-Trust and Diffusion-Trust sampling. The two modes are close on math and most knowledge benchmarks, while Diffusion-Trust is consistently weaker on code generation, where long, strictly structured outputs are committed block by block without left-to-right syntactic verification and are more sensitive to brittle answer extraction and long-output truncation; results marked with * should therefore be interpreted conservatively. Even

Table 2: **Benchmark performance of FLARE-2B and FLARE-4B. Bold**: best open-source dLLM; underline: second best; *: potentially under-reported.

	SDAR 1.7B	SDAR 4B	SDAR 8B	FLARE -2B	FLARE -2B	Qwen3.5 -2B	FLARE -4B	FLARE -4B	Qwen3.5 -4B
Params	*1.7B*	*4B*	*8B*	*2B*	*2B*	*2B*	*4B*	*4B*	*4B*
Sampling mode	*Diffusion*	*Diffusion*	*Diffusion*	*AR-Trust*	*Diffusion-Trust*	*AR*	*AR-Trust*	*Diffusion-Trust*	*AR*
Knowledge & Instruction Following									
ARC-Challenge	85.4	90.5	91.9	85.07	85.84	92.15	93.52	**94.62**	96.33
MMLU	62.9	74.9	78.6	67.60	64.14	73.59	78.73	**79.54**	85.22
MMLU-Pro	37.0	50.9	56.9	53.57	53.63	59.53	**71.14**	70.95	77.88
GPQA-Diamond	29.8	33.0	40.2	37.37	35.35	62.12	63.64	**64.65**	80.30
IFEval	43.4	56.6	61.4	68.95	62.66	79.48	73.20	**73.57**	90.02
Math									
GSM8K	80.1	89.9	91.3	84.46	82.79	77.63*	91.05	**91.58**	89.16
MATH-500	63.2	72.8	78.6	84.40	82.20	72.20*	**94.20**	91.60	95.40
AIME-24	10.0	10.0	10.0	31.11	31.11	8.89*	**58.89**	55.56	63.33
AIME-25	2.1	7.5	10.0	26.67	26.67	12.22*	43.33	**46.67**	48.89
Code									
HumanEval	61.6	72.8	78.7	64.02	50.61*	48.17	**93.29**	83.54*	87.80
MBPP	61.1	65.4	72.0	68.09	55.25*	53.31	**89.11**	77.82*	82.49
LiveCodeBench v6	5.7	13.1	16.6	15.43	9.71*	17.71	**41.71**	12.57*	50.86

with this caveat, a single FLARE checkpoint supports both a clean-stream verified causal sampling path and a block-diffusion sampling path, which addresses the feasibility side of question (iii). A residual gap to the original AR checkpoints remains, especially on instruction following and some coding benchmarks; we attribute it partly to continued SFT on external data that shifts the output distribution away from the post-trained source model, consistent with Section 3.

4.2 Efficient Training on Hybrid Backbones

This subsection addresses question (ii): whether the transfer configuration can run on a hybrid-attention backbone at hardware efficiency close to pure-AR training. In softmax-attention layers the clean/noisy mask of Section 2.2 is just a visibility mask over key-value pairs, but in recurrent linear-attention layers visibility is encoded by the state trajectory itself: the same algorithmic mask thus becomes a state-scheduling problem inside the hybrid backbone. On Qwen3.5, it affects both sequence-mixing components of each GDN layer, the GDR recurrence and the width-W causal ShortConv that prepares its inputs, and must keep packed documents isolated despite recurrent state propagation (Appendix D.3).

We compare two realizations of this schedule (Appendix D.4.4). *Route I (chunk-then-refine)* materializes every block-boundary clean state $\mathbf{S}_{(b-1)B}$ in HBM and runs each noisy block as a seeded local recurrence; it reuses the standard AR kernel structure but introduces L/B-scaling state hand-offs that dominate memory at small B. *Route II (fused two-stream)* stores only strided clean-state checkpoints, reconstructs the required boundary state in registers, and immediately consumes the corresponding noisy block; the ShortConv branch likewise fuses lag selection, document-boundary masking, and noisy-block convolution into one kernel. At the kernel level, Route II is most effective in the small-block regime diffusion training requires: for the GDR at $B{=}1$, it cuts total latency from 135.10 ms to 37.69 ms and peak memory from 18.14 GiB to 0.45 GiB, whereas Route I can win at larger B (Figure 13 in Appendix D.7). We therefore use Route II for FLARE's small diffusion blocks ($B{=}4$).

These kernel-level gains carry to end-to-end training (Figure 14 in Appendix D.8), measured as model FLOPs utilization (MFU). Replacing the GDR path with Route II removes the small-B memory failure and raises MFU at $B{=}4$ from 13.80% to 17.93%. The recovered memory headroom supports a larger per-GPU local batch, lifting MFU to 21.40%. Adding the fused ShortConv path brings FLARE-2B to 24.81% MFU at $B{=}4$, *matching and slightly exceeding* the pure-AR Qwen3.5-2B reference (24.04%) on the same hardware. Since two-stream supervision processes more work per step than single-stream AR training, the cost of recurrent-state scheduling is fully absorbed by the hardware-aware implementation rather than becoming the dominant training bottleneck. This affirmatively answers question (ii).

4.3 Unified Serving for AR-Trust and Diffusion-Trust

This subsection addresses question (iii): whether a single serving stack can support both decoding paths efficiently. Inference is the other stage where the hybrid backbone changes the standard dLLM serving problem. A pure softmax Transformer can treat speculative decoding largely as KV-cache management, while **FLARE** must coordinate a softmax KV cache, a GDN recurrent state, and the clean/noisy attention patterns used by the two decoding paths. We implement this in an SGLang-based serving stack (Zheng et al., 2024) that runs the AR-Trust and Diffusion-Trust interfaces of Section 2.3 from the same checkpoint, with path-specific masks and shared hybrid-attention kernels. Full pseudocode for both paths is given in Appendix E.

Beyond a standard speculative-decoding engine, the stack adds four mechanisms, detailed in Appendix E.4: recurrent-state commit, native dLLM mask modes, fused verification and top-k kernels, and CUDA-graph replay safety. The most hybrid-specific is recurrent-state commit: accepting only r of K speculative tokens also requires rewinding the GDN state to $\mathbf{S}^{(r)}$, which **FLARE** performs by recording the verify-position states inside the recurrent kernel and committing the accepted offset with one fused gather-scatter kernel, instead of replaying accepted tokens through every GDN layer.

On a single A100-80GB at $C{=}8$ under AR-Trust, **FLARE**-2B reaches 2,087 tokens/s on GSM8K, a $2.2\times$ gain over LLaDA-2.1-mini and a $4.8\times$ gain over SDAR-1.7B, and 1,441 tokens/s on GPQA-Diamond, a $3.6\times$ gain over LLaDA-2.1-mini (Figure 19 and Table 10 in Appendix G.2). Additional H100 measurements at 2,048 and 16,384 output tokens, including a controlled comparison with causal AR decoding of the same checkpoint and tokens-per-forward statistics, show the same trend (Appendix G.2.2).

Together, these results answer the three questions posed at the start of this section: capability is preserved at all three scales, two-stream supervision reaches pure-AR-level MFU, and a single checkpoint serves both decoding paths with substantial throughput advantages over other dLLMs.

5 Conclusion, Limitations, and Future Work

Conclusion. We presented **FLARE**, a systematic recipe for converting strong hybrid-attention AR models into capable, high-throughput diffusion LLMs. The work contributes several mutually reinforcing designs: a token-balanced clean/noisy *two-stream training objective*; a document-packed clean/noisy *attention-mask design*; *transfer onto hybrid-attention backbones* through recurrent-state scheduling and matching training kernels, reaching training efficiency comparable to pure-AR training; and a *unified inference system* in which a single checkpoint supports both parallel denoising and causal verified decoding. Our controlled study further identifies *transfer-data quality* as one of the key factors for capability preservation. Together, these results show that, under a modest conversion budget of roughly 10B tokens, AR-to-dLLM transfer can both retain the source model's capability and deliver real parallel-decoding throughput, indicating that practical dLLMs are limited less by the decoding algorithm itself than by the joint design of objective, data, and inference system.

Limitations. We group the limitations of this work into three points. **(i) Training overhead.** Block-diffusion training concatenates a clean and a noisy view of every sequence into a single $2L$-length input, which roughly doubles the per-step compute and memory relative to size-matched AR training; our dedicated kernels mitigate but do not eliminate this overhead. **(ii) Residual gap to the source model.** Even with our best data mix, **FLARE** still trails its AR source model on several benchmarks, consistent with a distribution mismatch between the external-teacher transfer data and the post-trained Qwen3.5 seed that more aggressive data filtering does not close. **(iii) Limited scale and post-training scope.** We validate **FLARE** only on dense checkpoints below 10B parameters and a single SFT stage; MoE backbones and post-training beyond SFT remain untested. Appendix C.2 discusses each point in detail.

Future work. These limitations outline open directions for the next generation of dLLM systems (Li et al., 2025b): single-stream training that removes the two-stream overhead, transfer data aligned with both the source model and the diffusion objective, up-cycling dense AR checkpoints into MoE-dLLMs, and reinforcement learning across **FLARE**'s multiple decoding paths; we discuss them in Appendix C.3.

Acknowledgments

YC is grateful for the support from NSF Grant DMS-2450378. MT acknowledges support from NSF Grant DMS-2513699, DOE Grant SC0026274, Adobe Gift, and the Richard Duke Fellowship. Part of this work was done while MT was at the Simons Institute for the Theory of Computing, for which he is grateful.

References

Joshua Ainslie, James Lee-Thorp, Michiel De Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. GQA: Training generalized multi-query Transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 4895–4901, 2023.

Anthropic. Introducing Claude Opus 4.7, 2026. https://www.anthropic.com/news/claude-opus-4-7.

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.

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

Aarti Basant, Abhijit Khairnar, Abhijit Paithankar, Abhinav Khattar, Adithya Renduchintala, Aditya Malte, Akhiad Bercovich, Akshay Hazare, Alejandra Rico, Aleksander Ficek, et al. NVIDIA Nemotron Nano 2: An accurate and efficient hybrid Mamba-Transformer reasoning model. *arXiv preprint arXiv:2508.14444*, 2025.

Ali Behrouz, Zeman Li, Praneeth Kacham, Majid Daliri, Yuan Deng, Peilin Zhong, Meisam Razaviyayn, and Vahab Mirrokni. ATLAS: Learning to optimally memorize the context at test time. *arXiv preprint arXiv:2505.23735*, 2025.

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

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.

Tiwei Bie, Maosong Cao, Xiang Cao, Bingsen Chen, Fuyuan Chen, Kun Chen, Lun Du, Daozhuo Feng, Haibo Feng, Mingliang Gong, et al. LLaDA2.1: Speeding up text diffusion via token editing. *arXiv preprint arXiv:2602.08676*, 2026.

Aaron Blakeman, Aarti Basant, Abhinav Khattar, Adithya Renduchintala, Akhiad Bercovich, Aleksander Ficek, Alexis Bjorlin, Ali Taghibakhshi, Amala Sanjay Deshmukh, Ameya Sunil Mahabaleshwarkar, et al. Nemotron-H: A family of accurate and efficient hybrid Mamba-Transformer models. *arXiv preprint arXiv:2504.03624*, 2025a.

Aaron Blakeman, Aaron Grattafiori, Aarti Basant, Abhibha Gupta, Abhinav Khattar, Adi Renduchintala, Aditya Vavre, Akanksha Shukla, Akhiad Bercovich, Aleksander Ficek, et al. Nemotron 3 Nano: Open, efficient mixture-of-experts hybrid Mamba-Transformer model for agentic reasoning. *arXiv preprint arXiv:2512.20848*, 2025b.

Aaron Blakeman, Aaron Grattafiori, Aarti Basant, Abhibha Gupta, Abhinav Khattar, Adi Renduchintala, Aditya Vavre, Akanksha Shukla, Akhiad Bercovich, Aleksander Ficek, et al. NVIDIA Nemotron 3: Efficient and open intelligence. *arXiv preprint arXiv:2512.20856*, 2025c.

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

Jian Chen, Yesheng Liang, and Zhijian Liu. DFlash: Block diffusion for flash speculative decoding. *arXiv preprint arXiv:2602.06036*, 2026.

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

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

Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. Rethinking attention with Performers. *arXiv preprint arXiv:2009.14794*, 2020.

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.

Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

Tri Dao and Albert Gu. Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. *arXiv preprint arXiv:2405.21060*, 2024.

Xin Dong, Yonggan Fu, Shizhe Diao, Wonmin Byeon, Zijia Chen, Ameya Sunil Mahabaleshwarkar, Shih-Yang Liu, Min-Hung Chen, Yoshi Suhara, Yingyan Celine Lin, et al. Hymba: A hybrid-head architecture for small language models. In *The Thirteenth International Conference on Learning Representations*, 2025.

Stefano Ermon. Introducing Mercury 2, 2026. https://www.inceptionlabs.ai/blog/introducing-mercury-2.

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

ggml-org. llama.cpp: LLM inference in C/C++, 2026. https://github.com/ggml-org/llama.cpp. GitHub repository.

Fabian Gloeckle, Badr Youbi Idrissi, Baptiste Rozière, David Lopez-Paz, and Gabriel Synnaeve. Better & faster large language models via multi-token prediction. *arXiv preprint arXiv:2404.19737*, 2024.

Paolo Glorioso, Quentin Anthony, Yury Tokpanov, James Whittington, Jonathan Pilault, Adam Ibrahim, and Beren Millidge. Zamba: A compact 7B SSM hybrid model. *arXiv preprint arXiv:2405.16712*, 2024.

Shansan Gong, Shivam Agarwal, Yizhe Zhang, Jiacheng Ye, Lin Zheng, Mukai Li, Chenxin An, Peilin Zhao, Wei Bi, Jiawei Han, et al. Scaling diffusion language models via adaptation from autoregressive models. *arXiv preprint arXiv:2410.17891*, 2024.

Google DeepMind. Gemini Diffusion: Our state-of-the-art, experimental text diffusion model, 2025. https://deepmind.google/models/gemini-diffusion/.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

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

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

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Sicong Jiang, Zilin Huang, Kangan Qian, Ziang Luo, Tianze Zhu, Yang Zhong, Yihong Tang, Menglin Kong, Yunlong Wang, Siwen Jiao, et al. A survey on vision-language-action models for autonomous driving. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4524–4536, 2025.

Wenqi Jiang, Jason Clemons, Karu Sankaralingam, and Christos Kozyrakis. How fast can I run my VLA? Demystifying VLA inference performance with VLA-Perf. *arXiv preprint arXiv:2602.18397*, 2026.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are RNNs: Fast autoregressive Transformers with linear attention. In *International conference on machine learning*, pages 5156–5165. PMLR, 2020.

Samar Khanna, Siddhant Kharbanda, Shufan Li, Harshit Varma, Eric Wang, Sawyer Birnbaum, Ziyang Luo, Yanis Miraoui, Akash Palrecha, Stefano Ermon, et al. Mercury: Ultra-fast language models based on diffusion. *arXiv e-prints*, pages arXiv–2506, 2025.

Jaeyeon Kim, Kulin Shah, Vasilis Kontonis, Sham Kakade, and Sitan Chen. Train for the worst, plan for the best: Understanding token ordering in masked diffusions. *arXiv preprint arXiv:2502.06768*, 2025.

Tanishq Kumar, Tri Dao, and Avner May. Speculative speculative decoding. *arXiv preprint arXiv:2603.03251*, 2026.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626, 2023.

Aakash Lahoti, Kevin Y Li, Berlin Chen, Caitlin Wang, Aviv Bick, J Zico Kolter, Tri Dao, and Albert Gu. Mamba-3: Improved sequence modeling using state space principles. *arXiv preprint arXiv:2603.15569*, 2026.

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.

Aonian Li, Bangwei Gong, Bo Yang, Boji Shan, Chang Liu, Cheng Zhu, Chunhao Zhang, Congchao Guo, Da Chen, Dong Li, et al. MiniMax-01: Scaling foundation models with Lightning Attention. *arXiv preprint arXiv:2501.08313*, 2025a.

Tianyi Li, Mingda Chen, Bowei Guo, and Zhiqiang Shen. A survey on diffusion language models. *arXiv preprint arXiv:2508.10875*, 2025b.

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

Aiwei Liu, Minghua He, Shaoxun Zeng, Sijun Zhang, Linhao Zhang, Chuhan Wu, Wei Jia, Yuan Liu, Xiao Zhou, and Jie Zhou. WeDLM: Reconciling diffusion language models with standard causal attention for fast inference. *arXiv preprint arXiv:2512.22737*, 2025a.

Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. DeepSeek-V2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. DeepSeek-V3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025b.

Jingyu Liu, Xin Dong, Zhifan Ye, Rishabh Mehta, Yonggan Fu, Vartika Singh, Jan Kautz, Ce Zhang, and Pavlo Molchanov. TiDAR: Think in diffusion, talk in autoregression. *arXiv preprint arXiv:2511.08923*, 2025c.

Aaron Lou, Chenlin Meng, and Stefano Ermon. Discrete diffusion modeling by estimating the ratios of the data distribution. *arXiv preprint arXiv:2310.16834*, 2023.

Yunchao Ma, Yizhuang Zhou, Yunhuan Yang, Tiancai Wang, and Haoqiang Fan. Running VLAs at real-time speed. *arXiv preprint arXiv:2510.26742*, 2025.

Mathematical Association of America. AIME. AIME problems and solutions, 2024. https://artofproblemsolving.com/wiki/index.php/AIME_Problems_and_Solutions, 2024.

Chien Van Nguyen, Chaitra Hegde, Van Cuong Pham, Ryan A. Rossi, Franck Dernoncourt, and Thien Huu Nguyen. Orthrus: Memory-efficient parallel token generation via dual-view diffusion. *arXiv preprint arXiv:2605.12825*, 2026.

Jinjie Ni, Qian Liu, Longxu Dou, Chao Du, Zili Wang, Hang Yan, Tianyu Pang, and Michael Qizhe Shieh. Diffusion language models are super data learners. *arXiv preprint arXiv:2511.03276*, 2025a.

Jinjie Ni, Qian Liu, Chao Du, Longxu Dou, Hang Yan, Zili Wang, Tianyu Pang, and Michael Qizhe Shieh. Training optimal large diffusion language models. *arXiv preprint arXiv:2510.03280*, 2025b.

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.

OpenClaw. OpenClaw: Personal AI assistant, 2026. https://github.com/openclaw/openclaw. GitHub repository.

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. *arXiv preprint arXiv:2406.03736*, 2024.

Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Leon Derczynski, et al. RWKV: Reinventing RNNs for the Transformer era. In *Findings of the association for computational linguistics: EMNLP 2023*, pages 14048–14077, 2023.

Bo Peng, Ruichong Zhang, Daniel Goldstein, Eric Alcaide, Xingjian Du, Haowen Hou, Jiaju Lin, Jiaxing Liu, Janna Lu, William Merrill, et al. RWKV-7 "Goose" with expressive dynamic state evolution. *arXiv preprint arXiv:2503.14456*, 2025.

Qwen Team. Qwen3-Next-80B-A3B-Instruct, 2025. https://qwen.ai/blog?id=4074cca80393150c248e508aa62983f9cb7d27cd&from=research.latest-advancements-list.

Qwen Team. Qwen3.5: Towards native multimodal agents, 2026. https://qwen.ai/blog?id=qwen3.5.

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

Liliang Ren, Yang Liu, Yadong Lu, Yelong Shen, Chen Liang, and Weizhu Chen. Samba: Simple hybrid state space models for efficient unlimited context language modeling. *arXiv preprint arXiv:2406.07522*, 2024.

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 Chiu, and Volodymyr Kuleshov. The diffusion duality. *arXiv preprint arXiv:2506.10892*, 2025.

Subham Sekhar Sahoo, Jean-Marie Lemercier, Zhihan Yang, Justin Deschenaux, Jingyu Liu, John Thickstun, and Ante Jukic. Scaling beyond masked diffusion language models. *arXiv preprint arXiv:2602.15014*, 2026.

Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear Transformers are secretly fast weight programmers. In *International conference on machine learning*, pages 9355–9366. PMLR, 2021.

Jiaxin Shi, Kehang Han, Zhe Wang, Arnaud Doucet, and Michalis Titsias. Simplified and generalized masked diffusion for discrete data. *Advances in neural information processing systems*, 37:103131–103167, 2024.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. OpenAI GPT-5 system card. *arXiv preprint arXiv:2601.03267*, 2025a.

Vaibhav Singh, Oleksiy Ostapenko, Pierre-André Noël, Eugene Belilovsky, and Torsten Scholak. DiffuMamba: High-throughput diffusion LMs with Mamba backbone. *arXiv preprint arXiv:2511.15927*, 2025b.

Yuxuan Song, Zheng Zhang, Cheng Luo, Pengyang Gao, Fan Xia, Hao Luo, Zheng Li, Yuehang Yang, Hongli Yu, Xingwei Qu, et al. Seed Diffusion: A large-scale diffusion language model with high-speed inference. *arXiv preprint arXiv:2508.02193*, 2025.

Yu Sun, Xinhao Li, Karan Dalal, Jiarui Xu, Arjun Vikram, Genghan Zhang, Yann Dubois, Xinlei Chen, Xiaolong Wang, Sanmi Koyejo, et al. Learning to (learn at test time): RNNs with expressive hidden states. *arXiv preprint arXiv:2407.04620*, 2024.

Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive network: A successor to Transformer for large language models. *arXiv preprint arXiv:2307.08621*, 2023.

Kimi Team, Yu Zhang, Zongyu Lin, Xingcheng Yao, Jiaxi Hu, Fanqing Meng, Chengyin Liu, Xin Men, Songlin Yang, Zhiyuan Li, et al. Kimi Linear: An expressive, efficient attention architecture. *arXiv preprint arXiv:2510.26692*, 2025.

The Gemini Team. Gemini 3.1 Pro: A smarter model for your most complex tasks, 2026. https://blog.google/innovation-and-ai/models-and-research/gemini-models/gemini-3-1-pro/.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Chenyu Wang, Paria Rashidinejad, DiJia Su, Song Jiang, Sid Wang, Siyan Zhao, Cai Zhou, Shannon Zejiang Shen, Feiyu Chen, Tommi Jaakkola, Yuandong Tian, and Bo Liu. SPG: Sandwiched policy gradient for masked diffusion language models. *arXiv preprint arXiv:2510.09541*, 2025a.

Ke Alexander Wang, Jiaxin Shi, and Emily B Fox. Test-time regression: a unifying framework for designing sequence models with associative memory. *arXiv preprint arXiv:2501.12352*, 2025b.

Xu Wang, Chenkai Xu, Yijie Jin, Jiachun Jin, Hao Zhang, and Zhijie Deng. Diffusion LLMs can do faster-than-AR inference via discrete diffusion forcing. *arXiv preprint arXiv:2508.09192*, 2025c.

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.

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. *arXiv preprint arXiv:2509.26328*, 2025a.

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.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Chen Yang, Yucheng Hu, Yunchao Ma, Yunhuan Yang, Jing Tan, and Haoqiang Fan. Realtime-VLA V2: Learning to run VLAs fast, smooth, and accurate. *arXiv preprint arXiv:2603.26360*, 2026a.

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving Mamba2 with delta rule. *arXiv preprint arXiv:2412.06464*, 2024a.

Songlin Yang, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. Parallelizing linear Transformers with the delta rule over sequence length. *Advances in neural information processing systems*, 37:115491–115522, 2024b.

Zhuolin Yang, Zihan Liu, Yang Chen, Wenliang Dai, Boxin Wang, Sheng-Chieh Lin, Chankyu Lee, Yangyi Chen, Dongfu Jiang, Jiafan He, et al. Nemotron-Cascade 2: Post-training LLMs with Cascade RL and multi-domain on-policy distillation. *arXiv preprint arXiv:2603.19220*, 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.

Yifan Yu, Yuqing Jian, Junxiong Wang, Zhongzhu Zhou, Donglin Zhuang, Xinyu Fang, Sri Yanamandra, Xiaoxia Wu, Qingyang Wu, Shuaiwen Leon Song, et al. Introspective diffusion language models. *arXiv preprint arXiv:2604.11035*, 2026.

Tianyuan Zhang, Sai Bi, Yicong Hong, Kai Zhang, Fujun Luan, Songlin Yang, Kalyan Sunkavalli, William T Freeman, and Hao Tan. Test-time training done right. *arXiv preprint arXiv:2505.23884*, 2025.

Siyan Zhao, Devaansh Gupta, Qinqing Zheng, and Aditya Grover. d1: Scaling reasoning in diffusion large language models via reinforcement learning. *arXiv preprint arXiv:2504.12216*, 2025.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. SGLang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

Shu Zhong, Mingyu Xu, Tenglong Ao, and Guang Shi. Understanding Transformer from the perspective of associative memory. *arXiv preprint arXiv:2505.19488*, 2025.

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

Fengqi Zhu, Zebin You, Yipeng Xing, Zenan Huang, Lin Liu, Yihong Zhuang, Guoshan Lu, Kangyu Wang, Xudong Wang, Lanning Wei, Hongrui Guo, Jiaqi Hu, Wentao Ye, Tieyuan Chen, Chenchen Li, Chengfu Tang, Haibo Feng, Jun Hu, Jun Zhou, Xiaolu Zhang, Zhenzhong Lan, Junbo Zhao, Da Zheng, Chongxuan Li, Jianguo Li, and Ji-Rong Wen. LLaDA-MoE: A sparse MoE diffusion language model. *arXiv preprint arXiv:2509.24389*, 2025a.

Yuchen Zhu, Wei Guo, Jaemoo Choi, Petr Molodyk, Bo Yuan, Molei Tao, and Yongxin Chen. Enhancing reasoning for diffusion LLMs via distribution matching policy optimization. *arXiv preprint arXiv:2510.08233*, 2025b.

A Broader Impact

This paper presents work aimed at advancing the field of machine learning. It proposes a new diffusion language model capable of generating text, thereby inheriting the full potential of LLMs. For example, it may be abused to create various harmful and offensive content. We strongly caution the community against such use cases.

B Related Work

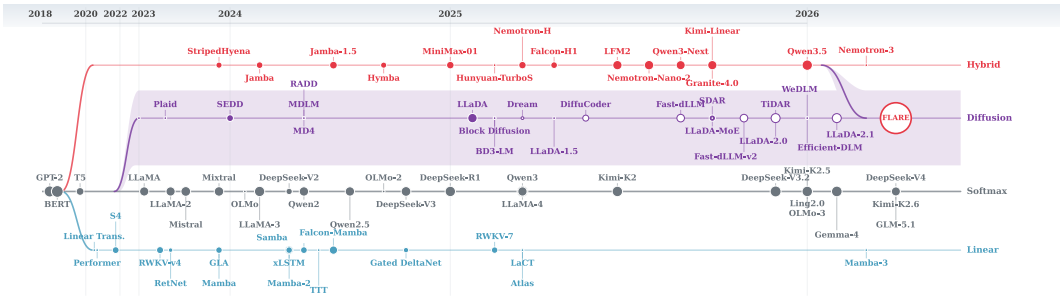


Figure 5: Timeline of open-source LLM architectures. **FLARE** is a fast and competent LLM as a confluence of algorithmic efficiency (diffusion) and architectural efficiency (hybrid attention).

Within the AR factorization, multi-token prediction and speculative decoding relax the one-token-per-forward bottleneck by drafting several future tokens and verifying them with a target model (Gloeckle et al., 2024; Li et al., 2025c; Cai et al., 2024; Leviathan et al., 2023; Kumar et al., 2026). These methods preserve the AR distribution under verification rules, but they still rely on a left-to-right reference model. Diffusion language models instead learn to fill masked positions through iterative denoising, allowing multiple token positions to be committed in one or a few forward passes. This line includes discrete diffusion models (Lou et al., 2023; Sahoo et al., 2024; Shi et al., 2024; Ou et al., 2024; Sahoo et al., 2025), large-scale dLLMs such as LLaDA, Dream, Mercury, Seed Diffusion (Nie et al., 2025; Bie et al., 2025, 2026; Ye et al., 2025; Khanna et al., 2025; Google DeepMind, 2025; Song et al., 2025), and inference-oriented systems such as DFlash (Chen et al., 2026). Block-diffusion LMs (Arriola et al., 2025) provide a middle ground: blocks are generated sequentially, while positions inside a block can be denoised in parallel. A group of works reduces the training cost of this paradigm by converting or adapting pretrained AR checkpoints into dLLMs (Gong et al., 2024; Fu et al., 2025; Wu et al., 2025a; Cheng et al., 2025; Liu et al., 2025c,a; Yu et al., 2026). These works make AR-to-diffusion transfer practical, but the sources of transfer degradation remain entangled across objective design, attention masks, transfer data, and evaluation protocol.

A separate line of work reduces the cost of each forward pass. Linear-attention, state-space, and recurrent-memory models replace the growing softmax-attention KV cache with a compact state (Katharopoulos et al., 2020; Choromanski et al., 2020; Schlag et al., 2021; Peng et al., 2023; Sun et al., 2023; Gu and Dao, 2023; Dao and Gu, 2024; Peng et al., 2025; Lahoti et al., 2026). Delta-rule and gated linear-attention variants further improve the recurrent update used by modern efficient LLMs (Yang et al., 2024b,a; Team et al., 2025). Related test-time-state models such as TTT and Atlas also fit the broader recurrent-memory trend shown in Figure 5 (Sun et al., 2024; Zhang et al., 2025; Behrouz et al., 2025). This improves memory scaling, especially for long contexts, but pure linear stacks can still be weaker than softmax attention on some tasks. As a result, recent efficient LLMs increasingly use hybrid backbones that interleave softmax and linear-attention layers, including Samba, Hymba, Zamba, MiniMax, Qwen3-Next/Qwen3.5, Kimi Linear, and Nemotron 3 (Ren et al., 2024; Dong et al., 2025; Glorioso et al., 2024; Li et al., 2025a; Qwen Team, 2025, 2026; Team et al., 2025; Basant et al., 2025; Blakeman et al., 2025a,b,c). These architectures reduce cache size and long-context cost while largely retaining AR training and decoding.

Closest in architecture, DiffuMamba (Singh et al., 2025b) trains diffusion LMs on Mamba and hybrid (DiffuMamba-H) backbones from scratch. It therefore does not address AR-to-dLLM transfer or the retention of an existing checkpoint's capability, which is the central problem **FLARE** targets. **FLARE** instead converts a strong post-trained hybrid AR checkpoint into a block-diffusion LLM;

to our knowledge, it is the first recipe to do so while supporting both AR-Trust and Diffusion-Trust decoding from a single checkpoint.

Taken together, the timeline in Figure 5 shows that open-source LLM efficiency has advanced along two largely separate threads: generation-side parallelism, including dLLMs, and architecture-side reductions in per-forward cost through hybrid attention. Yet these threads have rarely been joined: dLLM work mostly assumes full-attention backbones, while hybrid LLMs largely retain AR training and decoding. **FLARE** studies this missing intersection as AR-to-diffusion post-training for hybrid checkpoints, adding a diffusion-style generation path to the same parameters while preserving causal generation. This requires jointly specifying the objective, clean/noisy mask, transfer data, recurrent-state implementation, and decoding interfaces developed in Section 2.

C Extended Discussion

This section collects discussion moved from the main text for space.

C.1 Key Takeaways

Taken together, the study yields four main takeaways. 1) **Objective and mask design matter for capability preservation:** a token-balanced clean/noisy loss combines AR next-token supervision with block-diffusion denoising, while the document-packed clean/noisy mask enables both causal and bidirectional training signals without cross-document leakage. 2) **Transfer data explains the remaining variation:** once the core loss–mask design is aligned, transfer-data quality and distribution match dominate the residual performance gap, making AR-SFT a useful low-cost proxy for data selection. 3) **Hybrid-backbone diffusion requires specialized training:** diffusion visibility on linear-attention components cannot be imposed as a dense mask alone, and **FLARE** realizes it through recurrent-state scheduling and hardware-aware training kernels. 4) **One checkpoint supports two generation regimes:** starting from Qwen3.5 hybrid-attention checkpoints (Qwen Team, 2026), **FLARE**-2B/4B/9B achieves competitive diffusion-LM quality while supporting both AR-style verified speculative decoding and diffusion-style parallel denoising (Figure 1).

C.2 Limitations

We group the limitations of this work into three points. **(i) Training overhead.** Block-diffusion training concatenates a clean and a noisy view of every sequence into a single $2L$-length input under a custom attention mask, which roughly doubles the per-step compute and memory relative to size-matched AR training and makes long-context training more expensive; the cost is amplified on hybrid backbones, where linear-attention layers require the noisy stream to read clean intermediate recurrent states at every block boundary — the root cause of the residual training-efficiency gap in Section 4.2. Our dedicated kernels mitigate but do not eliminate this underlying $2\times$ overhead. **(ii) Residual gap to the source model.** Even with our best data mix, **FLARE** still trails its AR source model on several benchmarks. This is consistent with source-distribution shift: continuing SFT on long-CoT traces from *external* teachers (DeepSeek-R1, Qwen3, GPT-OSS-120B) moves the output distribution away from the post-trained Qwen3.5 seed. Notably, more aggressive *filtering* via our automatic IFD-based selection pipeline (Appendix F.5) does not close this gap, indicating that the bottleneck is a distribution mismatch between the transfer data and the source model rather than insufficient data filtering, and pointing to data more tightly aligned with both the source AR teacher and the diffusion objective. **(iii) Limited scale and post-training scope.** We validate **FLARE** only on dense checkpoints below 10B parameters and a single SFT stage; MoE backbones and post-training beyond SFT (e.g., reinforcement learning) remain untested, and whether they preserve capability under the same low conversion budget is open.

C.3 Future Directions

The limitations discussed in Section 5 outline open directions for the next generation of dLLM systems (Li et al., 2025b). The most fundamental is to move beyond the concatenated two-stream formulation toward *single-stream training* in which clean and noisy supervision share one forward pass, lowering this overhead along two complementary routes: the backbone side, as in DiffuMamba (Singh et al., 2025b), and the training-cost side, as in Orthrus (Nguyen et al., 2026) (tuning only $\sim$16% of

parameters). On the data side, the next step is transfer data more tightly aligned with the source-model distribution and the diffusion objective — e.g., diffusion-friendly traces harvested from the same-family AR teacher and matched to the block structure — rather than only finer filtering of existing external corpora. On the scaling side, the recipe can be used to up-cycle dense AR checkpoints into MoE-dLLMs, where LLaDA-MoE (Zhu et al., 2025a) already shows feasibility, raising the question of how expert routing interacts with the diffusion mask. Finally, closing the residual quality gap will likely require RL: the per-sequence likelihood of a masked-diffusion model is intractable, so policy-gradient methods need ELBO surrogates (Zhao et al., 2025; Zhu et al., 2025b; Wang et al., 2025a), and **FLARE**'s multiple decoding paths raise the further question of which path to roll out under and how to credit-assign reward across the clean and noisy streams.

D Hardware-Aware Kernels for Hybrid-Backbone Diffusion Training

This appendix specifies the kernel-level implementation of the two-stream recurrence defined in Section 2.2: its forward computation on the hybrid GDN backbone, its gradients, and the shipped implementation. A GDN layer contains two sub-components that both require re-implementation under **FLARE**'s two-stream training: the chunkwise-parallel GDR recurrence, and the width-W depthwise *1D Causal ShortConv* that feeds it. The rest of the appendix proceeds as follows.

- §D.3 states the three challenges that separate **FLARE**'s two-stream training from single-stream training.
- §D.4 develops the GDR kernels: it reviews the single-stream chunkwise form, lifts it to two streams via the block-boundary clean seed and block-level chain rule, derives the backward primitives, and specifies and compares the two implementation routes.
- §D.5 maps the same construction onto the 1D Causal ShortConv at lower cost, again as two implementation routes.
- §D.6 adds the document-level guards that keep packed samples isolated under recurrent-state propagation.
- §D.8 reports the end-to-end training-MFU gains of the shipped kernel stack on **FLARE**-2B.

D.1 Background: attention as associative memory

This subsection gives the full preliminaries summarized in Section 2.1, on which the kernel derivations below build.

To ground the subsequent analysis of hybrid-attention backbones and non-causal visibility patterns, we introduce softmax and linear attention under a unified *associative-memory* view (Zhong et al., 2025; Wang et al., 2025b), which abstracts attention layers as modules that memorize and retrieve. For notation, we denote the per-token queries and keys $\mathbf{q}_t, \mathbf{k}_t \in \mathbb{R}^{d_k}$ and values $\mathbf{v}_t \in \mathbb{R}^{d_v}$, received by each layer. It emits an output $\mathbf{o}_t \in \mathbb{R}^{d_v}$. Under this view, softmax and linear attention share a memory–retrieval interface, but differ in how past $(\mathbf{k}_i, \mathbf{v}_i)$ pairs are stored and queried by $\mathbf{q}_t$.

Softmax attention as explicit key-value memory. A softmax layer explicitly stores all visible key-value (KV) pairs $(\mathbf{k}_i, \mathbf{v}_i)$ in a KV cache. Given a query $\mathbf{q}_t$, retrieval is performed by normalizing query-key similarities over a visible index set and taking a weighted sum of the corresponding values. Using a generic visible set $\mathcal{V}_t$, the softmax retrieval is

$$\mathbf{o}_t = \sum_{i \in \mathcal{V}_t} \frac{\exp(\mathbf{q}_t^\top \mathbf{k}_i / \sqrt{d_k})}{\sum_{j \in \mathcal{V}_t} \exp(\mathbf{q}_t^\top \mathbf{k}_j / \sqrt{d_k})} \mathbf{v}_i, \tag{6}$$

where $\mathcal{V}_t$ denotes the visible index set for position t; for full attention, $\mathcal{V}_t$ contains all positions, while for causal attention it contains positions no later than t. This explicit key-value memory provides accurate token-level access and long-range retrieval capability, but its KV-cache footprint grows as $\mathcal{O}(T(d_k + d_v))$; dense full-sequence attention costs $\mathcal{O}(T^2)$, while incremental decoding still requires $\mathcal{O}(T)$ retrieval work per generated token.

Linear attention as recurrent memory. Linear attention (Katharopoulos et al., 2020; Choromanski et al., 2020) replaces the explicit KV cache with a bounded-size recurrent state $\mathbf{S}_t \in \mathbb{R}^{d_k \times d_v}$. In its

simplest form, the state accumulates past key-value information through outer-product updates,

$$\mathbf{S}_t = \mathbf{S}_{t-1} + \mathbf{k}_t \mathbf{v}_t^\top, \qquad \mathbf{o}_t = \mathbf{S}_t^\top \mathbf{q}_t. \tag{7}$$

Retrieval is therefore performed from this compressed state rather than through explicit attention over stored key-value pairs. Modern efficient LLMs often instantiate recurrent memory with gating, decay, or delta-rule updates, including Gated DeltaNet (Yang et al., 2024a) and Kimi Delta Attention (Team et al., 2025). These mechanisms share a bounded-state interface but differ in their concrete parameterizations. In this work, we use Gated DeltaNet (GDN) as the representative instantiation for exposition. Its memorization and retrieval rule, the Gated Delta Rule (GDR), is given in Eq. (1) (Section 2.1). The state-scheduling principle developed later relies on this recurrent-state interface rather than the specific parameterization of GDN, and can extend to other linear-attention variants. Compared with softmax attention, recurrent-memory attention maintains an $\mathcal{O}(d_k d_v)$ state independent of sequence length; processing a length-T sequence costs $\mathcal{O}(T)$ time, and incremental decoding uses constant-size state updates and retrieval per generated token, at the cost of a bounded and lossy memory.

Hybrid attention backbones. The two mechanisms above occupy different points on the spectrum of retrieval fidelity, memory compression, and serving efficiency. Softmax attention preserves explicit key-value memory and supports accurate token-level and long-range retrieval, as in Eq. (6), while recurrent-memory attention compresses history into bounded states, as in Eq. (1), reducing the dependence of memory on context length at the cost of lossy retrieval. Hybrid-attention backbones combine these mechanisms at the architecture level: softmax layers provide high-fidelity access to selected context, whereas recurrent-memory layers act as efficient compressed memory. By controlling the ratio and placement of these layer types, recent efficient LLM families seek to preserve model capability while reducing cache size and long-context serving cost (Dong et al., 2025; Glorioso et al., 2024; Li et al., 2025a; Qwen Team, 2025, 2026; Blakeman et al., 2025c; Team et al., 2025). Our main **FLARE** instantiation follows this setting. A hybrid backbone contains both explicit key-value memory and recurrent-state memory, so the same non-causal visibility pattern must be implemented through different mechanisms across layer types; Section 2.2 instantiates this issue for the clean/noisy training mask.

D.2 Recurrent-state schedule for the clean/noisy mask

Under the memory view above, an attention mask specifies which token information is accessible when a query reads from memory. Figure 2 shows the document-packed clean/noisy visibility pattern used to compute the objective above in a packed forward pass: the clean stream remains token-causal and isolated from the noisy stream, each noisy block is bidirectional within the block and attends to preceding clean context, and document boundaries isolate packed samples to prevent cross-document leakage while avoiding padding overhead. The two memory implementations in a hybrid backbone realize this same pattern differently: in softmax layers, an additive mask $\mathbf{M} \in \{0, -\infty\}^{T \times T}$ inside the softmax of Eq. (6) directly removes forbidden entries from the visible set $\mathcal{V}_t$; in recurrent-memory layers, making a past pair $(\mathbf{k}_i, \mathbf{v}_i)$ inaccessible to a later query means preventing its information from entering or propagating through the relevant state $\mathbf{S}_t$. Thus the same non-causal clean/noisy mask becomes a schedule of writes, state propagation, and state resets rather than a post-hoc masking matrix; the corresponding recurrent-state schedule is derived below.

The clean/noisy mask of Figure 2 can be applied as a visibility matrix in softmax layers, but must be translated into a state schedule for linear-attention layers. We use the generic form $\mathbf{S}_\ell = \mathcal{T}_\ell \mathbf{S}_{\ell-1} + \mathcal{W}_\ell$ with per-position output $\mathbf{o}_\ell = f(\mathbf{S}_\ell, \mathbf{q}_\ell)$, which covers Mamba-2 (Dao and Gu, 2024), Gated DeltaNet (Yang et al., 2024a), and Kimi Delta Attention (Team et al., 2025). We instantiate the construction with GDN below, while the same state-scheduling idea extends to other members of this family by substituting the corresponding transition, write, and output maps. Let $e(b) = bB$; we index clean positions by t, noisy positions by ℓ, and write noisy-side quantities with a tilde. For clarity, the equations below are written for a single document segment; under document packing, the same schedule is applied independently within each segment, with states reset at document boundaries.

For a GDN layer, the clean stream follows the standard causal recurrence of Eq. (1). For noisy block b, the recurrent state is initialized from the preceding clean boundary state, $\tilde{\mathbf{S}}_{e(b-1)} := \mathbf{S}_{e(b-1)}$, and then updated only with noisy tokens from the same block:

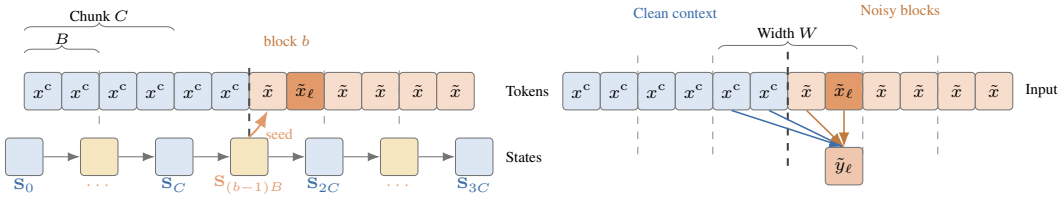
$$\begin{cases} \mathbf{S}_t = \alpha_t \big(\mathbf{I} - \beta_t \mathbf{k}_t \mathbf{k}_t^\top\big) \mathbf{S}_{t-1} + \beta_t \mathbf{k}_t \mathbf{v}_t^\top, & \text{clean stream,} \\ \tilde{\mathbf{S}}_\ell = \tilde{\alpha}_\ell \big(\mathbf{I} - \tilde{\beta}_\ell \tilde{\mathbf{k}}_\ell \tilde{\mathbf{k}}_\ell^\top\big) \tilde{\mathbf{S}}_{\ell-1} + \tilde{\beta}_\ell \tilde{\mathbf{k}}_\ell \tilde{\mathbf{v}}_\ell^\top, & \text{noisy block } b, \end{cases} \tag{8}$$

where $\ell \in \{e(b-1)+1, \ldots, e(b)\}$. The clean branch uses the standard causal readout $\mathbf{o}_t = \mathbf{S}_t^\top \mathbf{q}_t / \sqrt{d_k}$, while the noisy branch reads every position from the shared block-end state $\tilde{\mathbf{o}}_\ell = \tilde{\mathbf{S}}_{e(b)}^\top \tilde{\mathbf{q}}_\ell / \sqrt{d_k}$. This block-end readout realizes bidirectional visibility within the noisy block, and resetting the noisy state to the corresponding clean boundary state prevents information flow across noisy blocks. These recurrences define the intended visibility pattern, but the direct sequential form is not used for training; the chunk-parallel realization and kernel implementation are discussed in Section 4.2 and the remainder of this appendix.

D.3 Why the two-stream case needs a new kernel

Three challenges that FLARE's training setup imposes on both sub-components of a GDN layer (the GDR recurrence and the 1D Causal ShortConv) force the design choices of §D.4–§D.6. Standard chunkwise-parallel GDR kernels and standard causal 1D convolution kernels, as shipped by single-stream linear-attention implementations, provide no mechanism for these requirements, since each assumes a single token-causal stream.

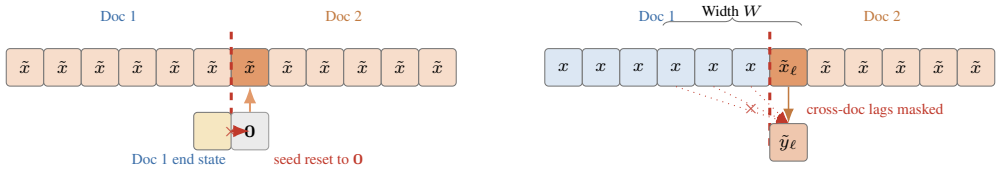
Challenge 1: clean and noisy streams need asymmetric visibility. FLARE's objective (Eq. (2)) couples two streams: clean tokens are token-causal over the whole clean stream, whereas a noisy token at position ℓ in block b sees only (i) noisy tokens in the same block and (ii) clean tokens strictly before the block start $(b-1)B$. Both sub-components must realize this asymmetry. The GDR seeds the noisy recurrence on block b from the block-boundary clean state $\mathbf{S}_{(b-1)B}$; since this state lies *inside* a chunk whenever the block size B differs from the chunk size C, a correct implementation must compute L/B mid-chunk clean states that the single-stream kernel never produces. The 1D Causal ShortConv instead splits its receptive field: a noisy output at offset $j \in \{0, \ldots, B-1\}$ from the block start reads its first $j+1$ lags from the noisy stream and the remaining $W-1-j$ lags from the clean stream, whereas a standard causal convolution reads all W lags from one stream.



(a) GDR **(b) 1D Causal ShortConv**

Figure 6: **Challenge 1: the clean/noisy asymmetric visibility for the two GDN sub-components. (a) GDR.** A single-stream kernel materializes the clean state only at chunk boundaries (blue, $\mathbf{S}_0, \mathbf{S}_C, \mathbf{S}_{2C}, \ldots$). Each noisy block b (e.g. the highlighted $\tilde{x}_\ell$) must instead be seeded from the block-boundary clean state $\mathbf{S}_{(b-1)B}$ *inside* a chunk (amber) whenever $B \neq C$ (here C=4, B=2); there are L/B such mid-chunk states and the single-stream kernel never produces them. **(b) 1D Causal ShortConv.** The width-W receptive field of a noisy output $\tilde{y}_\ell$ at offset j (here W=4, j=1) splits across the block boundary: the first $j+1$ lags read the noisy stream in-block, the remaining $W-1-j$ read the clean stream before the boundary, unlike a standard convolution that reads all W lags from one stream.

Challenge 2: document packing requires state resets. For throughput, multiple documents are packed into each L-token sequence, with per-document starts encoded as $\mathbf{cu} = (c_0, c_1, \ldots, c_{N_{\text{doc}}})$; the packed run must reproduce the per-document single-sequence result. This forces a document-level mask to compose with both the two-stream mask and the chunkwise state each sub-component tracks. For the GDR, the noisy recurrence's initial state at each document's first block must be zeroed rather than inherited from the previous document's chunk-boundary state, and the backward's cross-chunk scan must zero its state-gradient hand-off at document boundaries. For the 1D Causal ShortConv, any lag read crossing a document boundary must be masked to zero instead of reading the previous document's trailing tokens. Single-stream kernels need none of these guards, since a plain token-causal mask packed with a document-causal mask already forbids cross-document dependencies.



(a) GDR noisy-seed reset **(b) ShortConv cross-doc lag mask**

Figure 7: **Challenge 2: document packing requires state resets at document boundaries.** A packed sequence holds Doc 1 and Doc 2 back to back (red boundary). **(a) GDR.** The noisy recurrence for Doc 2's first block must be seeded from a *zeroed* state (**0**) rather than inheriting Doc 1's end-of-document state, which the cross-document carry ($\times$) would otherwise propagate. **(b) 1D Causal ShortConv.** A noisy output $\tilde{y}_\ell$ near Doc 2's start has lags whose width-W window reaches back across the boundary; those cross-document lags are masked to zero ($\times$), leaving only the in-document lag. Single-stream kernels need none of these guards.

Challenge 3: composing three granularities in the backward. Realizing the Challenge 1 visibility is cheap in the forward— either L/B extra bytes in HBM or $\mathcal{O}(B)$ register-replay steps per block for the GDR, and negligible for ShortConv (whose "state" is a raw $W-1$-token slice). The difficulty concentrates in the backward. A standard chunk-parallel clean backward carries a *single* cross-chunk gradient $\mathbf{dh}_{[c]} \equiv \partial\mathcal{L}/\partial\mathbf{S}_{[c]}$ through a matmul scan, delivering all per-token clean gradients at tensor-core throughput. In the two-stream case the noisy backward emits two further objects the chunk-level scan cannot directly accept: per-block clean-transition gradients (§D.4.3) and per-block initial-state gradients $\mathbf{d}\tilde{\mathbf{S}}_{b,\text{init}}$. Routing them into the scan without breaking its parallelism requires composing three granularities of reverse recurrence—token within a block, block within a chunk, and chunk within the sequence—in the correct order (§D.4.3). The ShortConv backward has only two of these granularities, because its noisy-side "state" is a static clean-stream slice rather than a recurrence terminus, which makes it cheaper than the GDR 's (§D.5.2). The routes of §D.4.5–§D.4.6 are two schedules of this composition onto a GPU; the resulting block-by-block backward dataflow is detailed in Figure 8.

AR-only training does not trigger these challenges. AR-only training of a GDN layer runs a single stream with uniform token-causal visibility. Challenge 1 does not arise because there is a single stream and no block-level granularity. Challenge 2 reduces to standard document-causal attention masking, which composes with the token-causal per-stream mask without any additional kernel-side guard. Challenge 3 reduces to a single cross-chunk state-gradient tensor at chunk-level granularity, which the standard chunkwise-parallel kernels of Yang et al. (2024a, 2025) already handle. **FLARE**'s block-seeded noisy-stream objective is what introduces the machinery described below.

Notation. Table 3 collects the symbols used in the remainder of this appendix. Positions in a stream are indexed by t (clean) or ℓ (noisy). The lengths B, C, L and derived counts M, N_C satisfy $L = C \cdot N_C$ and $C = B \cdot M$; we use C=64 throughout.

D.4 Gated Delta Rule: two-stream training and implementation routes

This subsection develops the chunkwise-parallel training routes for the GDR, the recurrence sub-component of the GDN layer. The objects and algorithms introduced here (chunk-boundary states, WY-corrected values, block-level chain rule, Route I and Route II) constitute the main technical content of the appendix; §D.5 reuses the same structure for the 1D Causal ShortConv at lower cost.

D.4.1 Single-stream chunkwise training

Both routes reuse the standard chunkwise-parallel training form of a single-stream GDR on the clean side unchanged. We restate it here to fix names for the four objects the clean-side forward produces and both routes reuse: the clean output, the chunk-boundary states, the chunk-corrected (WY) values, and the cumulative log-gates.

Table 3: Symbols used in this appendix. Clean-side quantities are plain; noisy-side quantities carry a tilde, matching the main text.

Symbol	Meaning
Lengths, indices, and tiling	
L	sequence length (one stream)
B	diffusion-block size
C	chunk size for chunkwise-parallel training; $C = 64$
$M = C/B$	blocks per chunk
$N_C = L/C$	chunks per stream
d_k, d_v	per-head key / value dimension
H	number of linear-attention heads per layer
b_v	kernel tile width in the value dimension ($b_v \le d_v$)
$c \in \{0, \dots, N_C{-}1\}$	chunk index
$b \in \{1, \dots, L/B\}$	block index; block b covers positions $(b{-}1)B{+}1, \dots, bB$
Per-position quantities	
$x_t^c \in \mathbb{R}^D$	clean-stream input to the 1D causal ShortConv at position t (§D.5)
$\tilde{x}_\ell \in \mathbb{R}^D$	noisy-stream input to the 1D causal ShortConv at position ℓ
$\mathbf{q}_t, \mathbf{k}_t, \mathbf{v}_t$	clean query / key / value at position t
$\tilde{\mathbf{q}}_\ell, \tilde{\mathbf{k}}_\ell, \tilde{\mathbf{v}}_\ell$	noisy query / key / value at position ℓ
$\tilde{g}_\ell, \tilde{\beta}_\ell, \tilde{\boldsymbol{\delta}}_\ell$	noisy log-gate / step-size / delta update at position ℓ
$g_t \in \mathbb{R}$	scalar log-gate at position t (clean); $\exp(g_t)$ is the multiplicative gate α_t of Eq. (1)
$\beta_t \in \mathbb{R}$	delta-rule step-size at position t (clean)
$G_t = \sum_{i \le t} g_i$	cumulative log-gate at position t (clean)
$\exp(G_j - G_i)$	decay factor from position i to position j (clean)
u_t	per-step prediction error $u_t = \mathbf{v}_t - (\mathbf{S}_t^{\text{pre}})^\top \mathbf{k}_t$
$\boldsymbol{\delta}_t$	per-step delta update $\boldsymbol{\delta}_t = \beta_t u_t$
$\hat{\mathbf{v}}_t$	chunk-corrected clean value (effective value produced by the chunkwise-parallel pre-step; see §D.4.1)
States and gradients	
$\mathbf{S}_t$	clean state after position t (after gating and delta update)
$\mathbf{S}_t^{\text{pre}}$	clean state after gating, before the delta update, at position t
$\tilde{\mathbf{S}}_\ell$	noisy state after position ℓ
$\mathbf{S}_{[c]} \equiv \mathbf{S}_{cC}$	chunk-c-boundary clean state
$\tilde{\mathbf{S}}_{b,\text{init}} \equiv \mathbf{S}_{(b-1)B}$	block-b noisy-recurrence initial state (seeded from clean)
$\mathbf{o}_t$	clean-stream output at position t (Eq. (9)); the full-sequence clean output is the stack $\mathbf{o} = (\mathbf{o}_t)_{t=1}^L$
$\tilde{\mathbf{o}}_\ell$	noisy-stream output at position ℓ under the block-end readout (Eq. (12)); the full-sequence noisy output is $\tilde{\mathbf{o}} = (\tilde{\mathbf{o}}_\ell)_{\ell=1}^L$
$\mathbf{O}_{[c]}$	stacked per-token clean outputs of chunk c (the chunk-level clean output tensor)
$\mathbf{do}_t \equiv \partial\mathcal{L}/\partial\mathbf{o}_t$	output-gradient input to the clean-side backward at position t; $\mathbf{do} = (\mathbf{do}_t)_{t=1}^L$ denotes the full-sequence clean output gradient
$\mathbf{d\tilde{o}}_\ell \equiv \partial\mathcal{L}/\partial\tilde{\mathbf{o}}_\ell$	output-gradient input to the noisy-side backward at position ℓ; $\mathbf{d\tilde{o}} = (\mathbf{d\tilde{o}}_\ell)_{\ell=1}^L$ denotes the full-sequence noisy output gradient
$\mathbf{dS}_t, \mathbf{d\tilde{S}}_\ell$	per-position state gradients
$\mathbf{dh}_b \equiv \partial\mathcal{L}/\partial\mathbf{S}_{bB}$	clean state gradient at the end of block b
$\mathbf{dh}_{[c]}^{\text{inject}}$	chunk-c within-chunk contribution to $\partial\mathcal{L}/\partial\mathbf{S}_{[c]}$
Block-level maps and checkpoint hyperparameters	
$f_b^{\text{blk}}(\mathbf{S})$	clean-state advance over block b (Eq. (11))
$\boldsymbol{\Delta}_b$	block b's outer-product write term (Eq. (11))
$S \in \{1, \dots, M\}$	checkpoint stride: every S-th block-boundary clean state is stored (§D.4.6)
$N_{\text{ckpt}} = M/S$	clean-state checkpoints per chunk; controls Route II peak HBM

Per-step recurrence. Expanding the GDR of Eq. (1) into its elementary per-step state and output updates yields the form used in both optimization and differentiation:

$$
\begin{aligned}
\mathbf{S}_t^{\text{pre}} &= \exp(g_t)\, \mathbf{S}_{t-1}, && \text{(gating decay)} \\
u_t &= \mathbf{v}_t - (\mathbf{S}_t^{\text{pre}})^\top \mathbf{k}_t, && \text{(prediction error)} \\
\boldsymbol{\delta}_t &= \beta_t\, u_t, && \text{(delta update)} \\
\mathbf{S}_t &= \mathbf{S}_t^{\text{pre}} + \mathbf{k}_t \boldsymbol{\delta}_t^\top, && \text{(outer-product write)} \\
\mathbf{o}_t &= s \cdot \mathbf{S}_t^\top \mathbf{q}_t, && \text{(output)}
\end{aligned}
\tag{9}
$$

where $\mathbf{S}_t^{\text{pre}} \in \mathbb{R}^{d_k \times d_v}$ is the state after the gating decay but before the delta update, $u_t \in \mathbb{R}^{d_v}$ is the per-step prediction error between the target value $\mathbf{v}_t$ and the value currently associated with $\mathbf{k}_t$ under $\mathbf{S}_t^{\text{pre}}$, $\boldsymbol{\delta}_t = \beta_t u_t \in \mathbb{R}^{d_v}$ is the β-weighted correction written into the state by the outer-product update, $\mathbf{S}_t \in \mathbb{R}^{d_k \times d_v}$ is the state after the update, $\mathbf{o}_t \in \mathbb{R}^{d_v}$ is the per-token output, and $s = 1/\sqrt{d_k}$ is the standard query-key scaling. The inputs $(g_t, \beta_t, \mathbf{q}_t, \mathbf{k}_t, \mathbf{v}_t)$ are produced by the linear projections and ShortConv branches of the GDN layer and are the same as in Eq. (1). §D.4.3 differentiates the five lines.

Chunkwise-parallel form. A token-by-token implementation of Eq. (9) is serial in the sequence dimension and underutilizes tensor cores. A fully materialized $L \times L$ form reaches tensor cores but incurs $\mathcal{O}(L^2)$ cost and loses the linear-attention advantage. The chunkwise-parallel form recovers both: split the sequence into $N_C = L/C$ chunks of size C (we set $C = 64$ to match tensor-core tile sizes), process each chunk with dense matrix multiplication, and carry a single compact state $\mathbf{S}_{[c]} \in \mathbb{R}^{d_k \times d_v}$ across chunks. Two pieces remain: how each chunk is processed in parallel despite the delta rule's within-chunk dependence, and how the compact state is advanced across chunks; the next two paragraphs address them in turn.

Chunk-corrected values. The delta-rule correction at each token depends on the partial state that earlier tokens in the same chunk have already written, so running Eq. (9) token-by-token within a chunk incurs C-step serial depth. The WY representation of the delta rule (Yang et al., 2024b,a) removes this depth by replacing each raw value $\mathbf{v}_t$ with a *chunk-corrected* (or *effective*) value $\hat{\mathbf{v}}_t$, computed from $\{(\mathbf{k}_t, \mathbf{v}_t, \beta_t, g_t)\}_{t \in \text{chunk}(c)}$ via a small triangular solve, so that the chunk's contribution to the state reduces to a single outer-product sum. We use $\hat{\mathbf{v}}_t$ throughout; in particular, the chunk-level state recurrence below is written directly in terms of $\hat{\mathbf{v}}_t$.

Chunk-level state recurrence. Let $G_t = \sum_{i \le t} g_i$ denote the per-token *cumulative log-gate* on the clean stream, and let $\mathbf{S}_{[c]} \equiv \mathbf{S}_{cC}$ denote the clean state at the boundary between chunks $c-1$ and c. In terms of $\hat{\mathbf{v}}$ and G, the single-stream GDR state recurrence over one chunk c admits the closed form

$$\mathbf{S}_{[c+1]} \;=\; \exp\left(G_{(c+1)C} - G_{cC}\right) \mathbf{S}_{[c]} \;+\; \sum_{t \in \text{chunk}(c)} \exp\left(G_{(c+1)C} - G_t\right) \mathbf{k}_t \, \hat{\mathbf{v}}_t^\top, \tag{10}$$

where $\exp\left(G_{(c+1)C} - G_{cC}\right)$ is the scalar decay factor produced by compounding the C per-token gates of chunk c, and $\exp\left(G_{(c+1)C} - G_t\right)$ is the residual decay from position t to the chunk end. Within-chunk work reduces to a dense sum of outer products (tensor-core friendly), and the only cross-chunk serial dependency is the single $\mathbf{S}_{[c]} \to \mathbf{S}_{[c+1]}$ hand-off. The chunk-level output $\mathbf{O}_{[c]}$ admits a similar closed form from the chunk's queries, $\mathbf{S}_{[c]}$, and in-chunk keys; since neither route modifies that kernel we omit it, though §D.4.3 differentiates its state-side contribution. Stacking $\mathbf{O}_{[c]}$ over all chunks gives the clean-stream output $\mathbf{o}$ listed below.

Clean-side forward outputs. The clean-side forward produces four objects, reused unchanged by both routes:

1. $\mathbf{o}$, the clean-stream output (the standard chunkwise attention output);
2. $\{\mathbf{S}_{[c]}\}_{c=0}^{N_C}$, chunk-boundary states from the scan in Eq. (10);
3. $\{\hat{\mathbf{v}}_t\}_{t=1}^{L}$, chunk-corrected clean values from the WY step;
4. $\{G_t\}_{t=1}^{L}$, per-position cumulative log-gates.

The route-specific work begins after these outputs: it concerns only the construction of the block-boundary clean states $\tilde{\mathbf{S}}_{b,\text{init}}$ that seed the noisy stream, which §D.4.2 takes up next.

D.4.2 Two-stream recurrence and block-level chain rule

The clean side of §D.4.1 is inherited unchanged from standard single-stream GDR training; the FLARE-specific work supplies the noisy stream with its block-boundary clean seed. This subsection defines the seed, specifies how the noisy recurrence consumes it, and derives the block-level chain rule that both routes implement.

Clean-transition map over one block. Specialising Eq. (10) to a single block of B tokens defines the *clean-transition map*. For block b spanning positions $(b-1)B+1, \ldots, bB$:

$$f_b^{\text{blk}}(\mathbf{S}) \;=\; \exp\left(G_{bB} - G_{(b-1)B}\right)\mathbf{S} \;+\; \mathbf{\Delta}_b, \qquad \mathbf{\Delta}_b \;=\; \sum_{t=(b-1)B+1}^{bB} \exp\left(G_{bB} - G_t\right)\mathbf{k}_t\,\hat{\mathbf{v}}_t^\top, \quad (11)$$

where $f_b^{\text{blk}} : \mathbb{R}^{d_k \times d_v} \to \mathbb{R}^{d_k \times d_v}$ is the block-b clean-transition map that advances the clean state by B tokens, $\exp\left(G_{bB} - G_{(b-1)B}\right)$ is the block's cumulative gate, and $\mathbf{\Delta}_b \in \mathbb{R}^{d_k \times d_v}$ is the block's outer-product write in terms of raw clean keys and chunk-corrected clean values.

Block-boundary clean states. Inside chunk c, the M block-boundary clean states $\mathbf{S}_{cM\cdot B+B}, \mathbf{S}_{cM\cdot B+2B}, \ldots, \mathbf{S}_{(c+1)C}$ are obtained by composing f^{blk} starting from the chunk-boundary state $\mathbf{S}_{[c]}$; applying f^{blk} all M times recovers $\mathbf{S}_{[c+1]}$ and Eq. (10) at the next chunk boundary. The two routes agree on the definition of these L/B states and differ only in the stage at which each state is materialized.

Noisy block forward. For block b, the noisy recurrence starts from the corresponding clean block-end state as its initial state, then runs Eq. (9) on the noisy tokens of that block only:

$$\tilde{\mathbf{S}}_{b,\text{init}} \;:=\; \mathbf{S}_{(b-1)B}, \qquad \tilde{\mathbf{S}}_\ell \;=\; \exp\left(\tilde{g}_\ell\right)\tilde{\mathbf{S}}_{\ell-1} \;+\; \tilde{\mathbf{k}}_\ell\,\tilde{\boldsymbol{\delta}}_\ell^\top, \quad (12)$$

for $\ell = (b-1)B+1, \ldots, bB$, where $\tilde{\mathbf{S}}_{b,\text{init}} \in \mathbb{R}^{d_k \times d_v}$ is block b's noisy-recurrence initial state (seeded from the clean state at the block boundary), $\tilde{\mathbf{S}}_\ell \in \mathbb{R}^{d_k \times d_v}$ is the noisy state after noisy position ℓ, and $(\tilde{g}_\ell, \tilde{\beta}_\ell, \tilde{\mathbf{q}}_\ell, \tilde{\mathbf{k}}_\ell, \tilde{\mathbf{v}}_\ell)$ are the noisy per-token inputs (produced by the same linear projections and ShortConv branches as their clean counterparts, but applied to the noisy stream). The noisy per-token delta update $\tilde{\boldsymbol{\delta}}_\ell = \tilde{\beta}_\ell\,\tilde{u}_\ell \in \mathbb{R}^{d_v}$ is defined via the noisy prediction error $\tilde{u}_\ell = \tilde{\mathbf{v}}_\ell - (\tilde{\mathbf{S}}_\ell^{\text{pre}})^\top\tilde{\mathbf{k}}_\ell$ with $\tilde{\mathbf{S}}_\ell^{\text{pre}} = \exp\left(\tilde{g}_\ell\right)\tilde{\mathbf{S}}_{\ell-1}$, reusing the five-line expansion of Eq. (9) on the noisy stream without restating it. The noisy output at every position in block b is read from the block-end state rather than the running state: $\tilde{\mathbf{o}}_\ell = s\,\tilde{\mathbf{S}}_{bB}^\top\tilde{\mathbf{q}}_\ell$ for all $\ell \in \{(b-1)B+1, \ldots, bB\}$. This single-shared readout state reproduces the bidirectional noisy-to-noisy region of the training mask (Eq. (8)). Between blocks the noisy state is reset; within a block it advances exactly B steps. Noisy blocks are mutually independent because each is seeded from its own clean state; all block-to-block information flow passes through the clean stream.

Block-level chain rule. Differentiating Eq. (11) yields the chain rule that both routes reuse. For a block b inside chunk c, let

$$\mathbf{dh}_b \;\equiv\; \partial\mathcal{L}/\partial\mathbf{S}_{bB}$$

denote the clean state gradient at the end of block b, accumulated so far from later blocks in the same chunk, and let $\mathbf{d}\tilde{\mathbf{S}}_{b,\text{init}}$ denote the gradient produced by reverse-sweeping the noisy recurrence of Eq. (12) on block b alone. The block-level recursion is then

$$\mathbf{dh}_{b-1} \;=\; \mathbf{d}\tilde{\mathbf{S}}_{b,\text{init}} \;+\; \exp\left(G_{bB} - G_{(b-1)B}\right)\mathbf{dh}_b. \quad (13)$$

The first term is block b's direct contribution to the gradient at its seed (the clean state at the end of block $b-1$). The second term is the gate-decayed gradient from blocks $b+1, b+2, \ldots$ propagated through f_b^{blk}'s scalar decay. Isolating the *within-chunk* part of this scan by seeding the recursion with $\mathbf{dh}_{cM+M} = \mathbf{0}$ (dropping external contributions to chunk c) and iterating M times yields $\mathbf{dh}_{cM}$, the *within-chunk contribution* $\mathbf{dh}_{[c]}^{\text{inject}}$. The chunk-level scan of Eq. (16) subsequently adds the two external contributions (from later chunks and from chunk c's clean-output gradient) on top of this injection.

D.4.3 Backward primitives

Three primitives appear repeatedly in both routes: a per-token backward inside one block (§D.4.3); a chunk-level backward across chunks on the clean side (§D.4.3); and a clean-transition backward that differentiates f_b^{blk} (§D.4.3). We derive each once and invoke them by name in Routes I and II.

Per-step backward. The per-step backward reverses a single token of Eq. (9); each reverse sweep over a noisy block consists of B consecutive invocations of this primitive. Let $\mathbf{dS}_t = \partial\mathcal{L}/\partial\mathbf{S}_t$, $\mathbf{do}_t = \partial\mathcal{L}/\partial\mathbf{o}_t$, $\mathbf{dS}_t^{\text{pre}} = \partial\mathcal{L}/\partial\mathbf{S}_t^{\text{pre}}$, and $\mathbf{d}u_t = \partial\mathcal{L}/\partial u_t$. Because $\mathbf{S}_t$ appears both in $\mathbf{o}_t$ and, through the subsequent token, in $\mathbf{S}_{t+1}$, its gradient is an *accumulator*: by the time $\mathbf{dS}_{t-1}$ is computed, $\mathbf{dS}_t$ has already received the contribution from $\mathbf{S}_{t+1}$ handed back by the reverse step on token $t{+}1$. The equations below therefore use '+=' (and in Eqs. (17)–(18) also '−=') in the programming sense: $x \mathrel{+}= y$ denotes $x \leftarrow x + y$. Differentiating the five lines of Eq. (9) in reverse yields

$$
\begin{aligned}
&\mathbf{dq}_t = s\,\mathbf{S}_t\mathbf{do}_t, && \text{(from } \mathbf{o}_t\text{)}\\
&\mathbf{dS}_t \mathrel{+}= s\,\mathbf{q}_t\,\mathbf{do}_t^\top, && \text{(from } \mathbf{o}_t\text{; add into } \mathbf{dS}_t\text{)}\\
&\mathbf{d}\boldsymbol{\delta}_t = \mathbf{dS}_t^\top\mathbf{k}_t,\ \ \mathbf{dk}_t^{\text{write}} = \mathbf{dS}_t\,\boldsymbol{\delta}_t, && \text{(from the outer-product write)}\\
&\mathbf{d}u_t = \beta_t\,\mathbf{d}\boldsymbol{\delta}_t,\ \ \mathbf{dv}_t = \mathbf{d}u_t,\ \ d\beta_t = \mathbf{d}\boldsymbol{\delta}_t^\top u_t, && \text{(from the delta update)}\\
&\mathbf{dk}_t^{\text{pred}} = -\mathbf{S}_t^{\text{pre}}\,\mathbf{d}u_t,\ \ \mathbf{dk}_t = \mathbf{dk}_t^{\text{write}} + \mathbf{dk}_t^{\text{pred}}, && \text{(from the prediction error)}\\
&\mathbf{dS}_t^{\text{pre}} = \mathbf{dS}_t - \mathbf{k}_t\,\mathbf{d}u_t^\top, && \text{(splits } \mathbf{S}_t = \mathbf{S}_t^{\text{pre}} + \mathbf{k}_t\boldsymbol{\delta}_t^\top\text{)}\\
&dg_t = \left\langle \mathbf{dS}_t^{\text{pre}},\ \mathbf{S}_t^{\text{pre}} \right\rangle_{\mathrm{F}}, && \text{(from } \mathbf{S}_t^{\text{pre}} = \exp(g_t)\,\mathbf{S}_{t-1}\text{)}\\
&\mathbf{dS}_{t-1} = \exp(g_t)\,\mathbf{dS}_t^{\text{pre}}, && \text{(gating reverse recurrence)}
\end{aligned}
\tag{14}
$$

where $\mathbf{d}\boldsymbol{\delta}_t = \partial\mathcal{L}/\partial\boldsymbol{\delta}_t \in \mathbb{R}^{d_v}$ is the gradient with respect to the delta update, $\mathbf{dk}_t^{\text{write}}, \mathbf{dk}_t^{\text{pred}} \in \mathbb{R}^{d_k}$ are the two contributions to $\mathbf{dk}_t$ from the outer-product write and the prediction error, respectively, and $\langle\cdot,\cdot\rangle_{\mathrm{F}}$ denotes the Frobenius inner product. The key gradient sums the two contributions because $\mathbf{k}_t$ enters both the outer-product write and the prediction error. Denote this primitive by PERSTEPBWD: it accepts $(\mathbf{dS}_t, \mathbf{do}_t, \mathbf{q}_t, \mathbf{k}_t, \mathbf{v}_t, g_t, \beta_t, \mathbf{S}_t^{\text{pre}}, \mathbf{S}_t)$ and returns $(\mathbf{dq}_t, \mathbf{dk}_t, \mathbf{dv}_t, dg_t, d\beta_t, \mathbf{dS}_{t-1})$. Eq. (14) assumes the token-causal readout $\mathbf{o}_t = s\,\mathbf{S}_t^\top\mathbf{q}_t$: each per-step state feeds exactly one output, so the output-side gradient enters the state accumulator at the same step. This is correct for the clean stream; the noisy stream uses a different output rule, which we cover next.

Noisy-stream backward under block-end readout. The noisy-block forward of Eq. (12) reads *every* noisy output in block b from the same end-of-block state $\tilde{\mathbf{S}}_{bB}$, so the mapping from outputs to states is many-to-one instead of one-to-one. Differentiating this readout gives a block-level output-side contribution that lives entirely on $\tilde{\mathbf{S}}_{bB}$'s gradient, plus a per-step query gradient that uses the same single state:

$$
\mathbf{d}\tilde{\mathbf{S}}_{bB} \;=\; s\sum_{\ell=(b-1)B+1}^{bB}\tilde{\mathbf{q}}_\ell\,\mathbf{d}\tilde{\mathbf{o}}_\ell^\top, \qquad \mathbf{d}\tilde{\mathbf{q}}_\ell \;=\; s\,\tilde{\mathbf{S}}_{bB}\,\mathbf{d}\tilde{\mathbf{o}}_\ell \quad \text{for all } \ell \in \{(b-1)B+1,\dots,bB\}. \tag{15}
$$

A reverse sweep over the noisy block then runs B consecutive PERSTEPBWD calls with $\mathbf{d}\tilde{\mathbf{o}}_\ell$ set to $\mathbf{0}$: the per-step output terms (the $\mathbf{dq}_t$ and $\mathbf{dS}_t \mathrel{+}= s\,\mathbf{q}_t\,\mathbf{do}_t^\top$ lines of Eq. (14)) are already supplied by Eq. (15), and the remaining lines (gating, outer-product write, prediction error, and gating reverse recurrence) are unchanged. Throughout this appendix, "reverse sweep over a noisy block of B tokens" refers to this procedure: initialize $\mathbf{d}\tilde{\mathbf{S}}_{bB}$ with Eq. (15), emit $\mathbf{d}\tilde{\mathbf{q}}_\ell$ directly from Eq. (15), then invoke PERSTEPBWD with $\mathbf{do} = \mathbf{0}$ for each of the B steps in reverse.

Chunk-level clean backward. The chunk-level backward differentiates Eq. (10) together with the clean-output kernel that emits $\mathbf{O}_{[c]}$. Its only cross-chunk serial step is a reverse matmul scan over chunk-boundary state gradients $\mathbf{dh}_{[c]} \equiv \partial\mathcal{L}/\partial\mathbf{S}_{[c]}$:

$$
\mathbf{dh}_{[c]} \;=\; \underbrace{\mathbf{dh}_{[c]}^{\text{inject}}}_{\text{external hand-off}} \;+\; \underbrace{\mathcal{C}_{[c]}}_{\text{in-chunk clean-output contribution}} \;+\; \exp\left(G_{(c+1)C} - G_{cC}\right)\mathbf{dh}_{[c+1]}, \tag{16}
$$

where $\mathbf{dh}_{[c]}^{\text{inject}} \in \mathbb{R}^{d_k\times d_v}$ is a per-chunk hand-off slot that accepts external state-gradients and adds them into the scan (written by Route I's fill-in backward, §D.4.5, or by Route II's fused kernel, §D.4.6), and $\mathcal{C}_{[c]} \in \mathbb{R}^{d_k\times d_v}$ is the in-chunk clean-output contribution obtained by differentiating $\mathbf{O}_{[c]}$ through $\mathbf{S}_{[c]}$ via a dense matmul over the C tokens of chunk c. $\mathcal{C}_{[c]}$ is identical to the AR baseline's clean-side computation and is not modified by either route; the two-stream-specific content is routed entirely through $\mathbf{dh}_{[c]}^{\text{inject}}$. Denote this primitive by CHUNKBWD.

Clean-transition backward. The clean-transition backward links the noisy-side reverse sweep to the clean-side cross-chunk scan: it converts the state gradient $\mathbf{dh}_b$ at the end of block b into per-token clean gradients and a propagated block-initial state gradient. It is the block-level analog of §D.4.3. We differentiate f_b^{blk} (Eq. (11)) with respect to its clean-side inputs $\big(\mathbf{k}_t, \hat{\mathbf{v}}_t, G_t\big)_{t\in\text{block}(b)}$ and the incoming state $\mathbf{S}$. Let $\rho_t \equiv \exp\left(G_{bB} - G_t\right)$ denote the per-token decay factor inside block b, and let $\mathbf{dh}_b \equiv \partial\mathcal{L}/\partial\mathbf{S}_{bB}$ denote the incoming state gradient.

Gradient through the scalar decay. f_b^{blk}'s leading term $\exp\left(G_{bB} - G_{(b-1)B}\right)\mathbf{S}$ propagates $\mathbf{dh}_b$ to $\mathbf{S}$ with factor $\exp\left(G_{bB} - G_{(b-1)B}\right)$; this factor is exactly the multiplicative factor in the block-level recursion of Eq. (13), and is how the recursion is derived. The same decay also yields gradients into G_{bB} and $G_{(b-1)B}$:

$$\begin{aligned} dG_{bB} \mathrel{+}= \exp\big(G_{bB} - G_{(b-1)B}\big)\,\big\langle \mathbf{dh}_b,\ \mathbf{S}\big\rangle_{\mathrm{F}}, &\qquad \text{(contribution to } dG_{bB}) \\ dG_{(b-1)B} \mathrel{-}= \exp\big(G_{bB} - G_{(b-1)B}\big)\,\big\langle \mathbf{dh}_b,\ \mathbf{S}\big\rangle_{\mathrm{F}}. &\qquad \text{(contribution to } dG_{(b-1)B}) \end{aligned} \tag{17}$$

The two writes have equal magnitude and opposite signs because $\partial(G_j - G_i)/\partial G_j = 1$ and $\partial(G_j - G_i)/\partial G_i = -1$. Since each dG accumulator receives contributions from several sources (this block's scalar-decay term of Eq. (17), this block's $\mathbf{\Delta}$ term below, and any later calls that hit the same G), we keep the '$+=$' / '$-=$' form throughout §D.4.3 to make the accumulator semantics explicit.

Gradient through the outer-product write. The write term $\mathbf{\Delta}_b = \sum_t \rho_t\,\mathbf{k}_t\,\hat{\mathbf{v}}_t^\top$ contributes, for each t in block b:

$$\begin{aligned} &\mathbf{dk}_t = \mathbf{dh}_b\big(\hat{\mathbf{v}}_t\,\rho_t\big), && \text{(clean key)} \\ &\mathbf{d}\hat{\mathbf{v}}_t = \rho_t\,\mathbf{dh}_b^\top\mathbf{k}_t, && \text{(chunk-corrected value)} \\ &\sigma_t := \rho_t\,\mathbf{k}_t^\top\mathbf{dh}_b\,\hat{\mathbf{v}}_t,\ \ dG_{bB} \mathrel{+}= \sigma_t,\ \ dG_t \mathrel{-}= \sigma_t, && \text{(cumulative gates)} \end{aligned} \tag{18}$$

where $\mathbf{dk}_t \in \mathbb{R}^{d_k}$ and $\mathbf{d}\hat{\mathbf{v}}_t \in \mathbb{R}^{d_v}$ are the per-token gradients of the block's outer-product write term with respect to the clean key and chunk-corrected clean value, and $\sigma_t \in \mathbb{R}$ is the scalar obtained by contracting $\mathbf{dh}_b$ with the rank-one update $\mathbf{k}_t\hat{\mathbf{v}}_t^\top$ and multiplying by ρ_t; it is the common magnitude of the two dG writes. The two G-writes have opposite signs because G_{bB} and G_t enter ρ_t with opposite signs. Summed over $t \in \text{block}(b)$ (and over all blocks and all contribution sources), these give the final per-token clean-side gradients $(\mathbf{dk}_t, \mathbf{d}\hat{\mathbf{v}}_t, dG_t)$ of block b.

Summary. Denote the combined sub-procedure by CLEANTRANSBWD: it accepts $(\mathbf{dh}_b, \mathbf{S}, \{(\mathbf{k}_t, \hat{\mathbf{v}}_t, G_t)\}_{t\in\text{block}(b)})$ and returns $(\mathbf{dk}_t, \mathbf{d}\hat{\mathbf{v}}_t, dG_t)$ for every token in block b. Its outputs are partial-gradient slots: the downstream chunk-level clean backward (CHUNKBWD, together with the WY backward that converts $\mathbf{d}\hat{\mathbf{v}}$ into $(\mathbf{dv}, \mathbf{dk}, d\beta)$) completes backpropagation into the final clean parameter gradients.

Gradient map. Figure 8 traces the composition of the three primitives on a single noisy token. Both routes produce the same tree and differ only in which kernel computes which subtree.

D.4.4 Overview of the two implementation routes

Both routes compute identical gradients and differ only in (i) how the block-boundary clean states $\mathbf{S}_{(b-1)B}$ are delivered on the forward and (ii) how block-level gradients are routed into the cross-chunk scan. Figure 9 contrasts their dataflow; §D.4.5 and §D.4.6 specify the algorithms and §D.4.7 quantifies the trade-off.

The block-boundary clean states $\mathbf{S}_{(b-1)B}$ follow §D.4.2; both routes ultimately route their gradients into the chunk-boundary slot consumed by CHUNKBWD.

D.4.5 Route I: Chunk-then-Refine

Route I materializes every $\mathbf{S}_{(b-1)B}$ in HBM prior to the noisy forward. The noisy stream then reduces to L/B independent block-local recurrences that run in parallel.

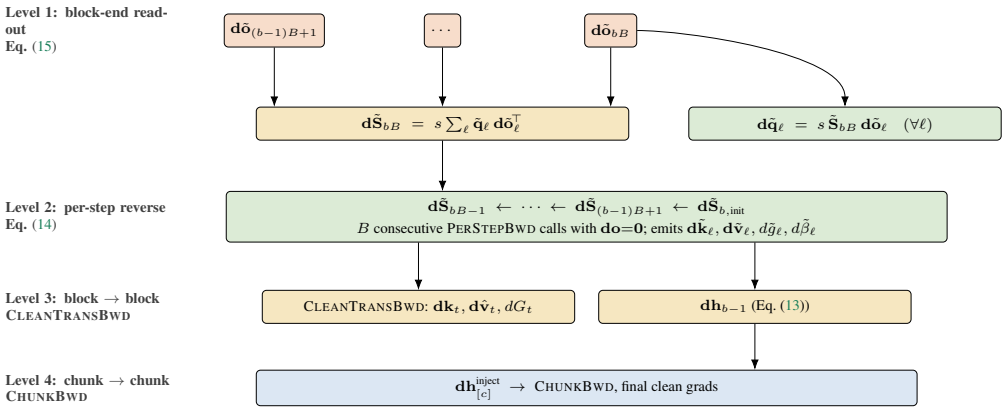
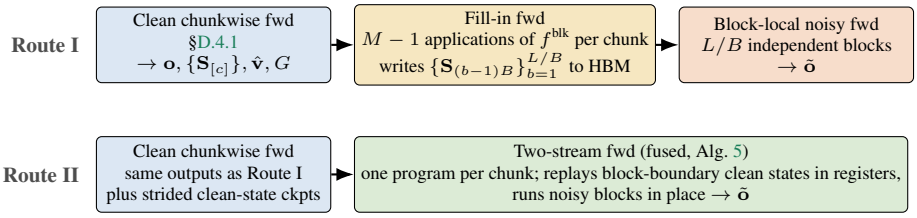
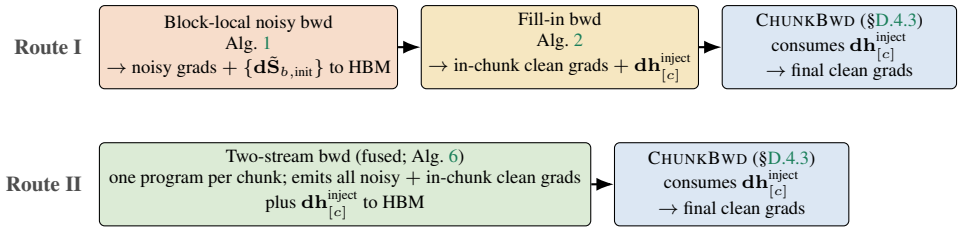


Figure 8: Backward dataflow for one noisy block b of chunk c under the block-end readout of Eq. (8). **Level 1** (output → state) folds the B per-token output gradients into the single block-end state accumulator $d\tilde{\mathbf{S}}_{bB}$ and produces $d\tilde{\mathbf{q}}_\ell$ for every ℓ directly (Eq. (15)). **Level 2** (per-step reverse) runs B PERSTEPBWD calls with $\mathbf{do}=\mathbf{0}$, propagating the block-end gradient to the block-initial state $d\tilde{\mathbf{S}}_{b,\text{init}}$ while emitting the remaining per-token gradients. **Level 3** (block → block) runs CLEANTRANSBWD on $d\mathbf{h}_b$ and combines $d\tilde{\mathbf{S}}_{b,\text{init}}$ with the gate-decayed $d\mathbf{h}_b$ (Eq. (13)) to produce $d\mathbf{h}_{b-1}$. **Level 4** (chunk → chunk) closes the block recursion into $d\mathbf{h}^{\text{inject}}_{[c]}$, which is consumed by CHUNKBWD. Both routes produce this dataflow; they differ only in which kernel hosts which level.



(a) Forward dataflow. Route I materializes every $\mathbf{S}_{(b-1)B}$ in HBM before the noisy blocks run; Route II replays each in registers from $\mathbf{S}_{[c]}$, writing only strided checkpoints for the backward.



(b) Backward dataflow. Route I chains three kernels through two HBM hand-off tensors; Route II collapses the first two into a single fused kernel and emits only $d\mathbf{h}^{\text{inject}}_{[c]}$ before CHUNKBWD.

Figure 9: Dataflow of the two implementation routes (forward, a; backward, b). Each box is a single Triton kernel launch. Both routes leave the clean-side kernels of §D.4.1 and §D.4.3 untouched.

1. **Clean chunkwise forward.** Run §D.4.1 on the clean stream; emit $\mathbf{o}$, $\{\mathbf{S}_{[c]}\}$, $\{\hat{\mathbf{v}}_t\}$, and $\{G_t\}$.

2. **Block-state fill-in (forward).** For each chunk c, apply $f^{\text{blk}}_{cM+1}, \ldots, f^{\text{blk}}_{cM+M-1}$ (Eq. (11)) starting from $\mathbf{S}_{[c]}$, and write every $\mathbf{S}_{(b-1)B}$ for b inside chunk c to HBM. The pass has no tensor-core work; it is a per-block recurrent scan whose only purpose is to populate the tensor $\{\mathbf{S}_{(b-1)B}\}_{b=1}^{L/B}$.

3. **Block-local noisy forward.** Treat the noisy stream as L/B independent length-B sequences; each starts from its fill-in state $\mathbf{S}_{(b-1)B}$, runs Eq. (12) for B steps, and emits $\tilde{\mathbf{o}}$ for those positions.

Backward. The backward mirrors the forward as three kernels connected by autograd through two HBM hand-off tensors (Figure 9b, Route I row). Using the primitives of §D.4.3:

Algorithm 1 Route I noisy backward (one program per block b).

Require: block b's noisy tokens $(\tilde{\mathbf{q}}, \tilde{\mathbf{k}}, \tilde{\mathbf{v}}, \tilde{g}, \tilde{\beta})$, seed $\mathbf{S}_{(b-1)B}$, noisy-output grads $\{\mathbf{d}\tilde{\mathbf{o}}_\ell\}_{\ell \in \text{block}(b)}$

run Eq. (12) forward for B steps, caching $(\tilde{\mathbf{S}}^{\text{pre}}_\ell, \tilde{\mathbf{S}}_\ell)$ per noisy token
$\mathbf{d}\tilde{\mathbf{S}}_{bB} \leftarrow s \sum_{\ell=(b-1)B+1}^{bB} \tilde{\mathbf{q}}_\ell \, \mathbf{d}\tilde{\mathbf{o}}_\ell^\top$ ▷ block-end readout init, Eq. (15)
$\mathbf{d}\tilde{\mathbf{q}}_\ell \leftarrow s \, \tilde{\mathbf{S}}_{bB} \, \mathbf{d}\tilde{\mathbf{o}}_\ell$ for each ℓ in block b
for $\ell = bB, bB-1, \ldots, (b-1)B+1$ **do**
    $(_, \mathbf{d}\tilde{\mathbf{k}}_\ell, \mathbf{d}\tilde{\mathbf{v}}_\ell, d\tilde{g}_\ell, d\tilde{\beta}_\ell, \mathbf{d}\tilde{\mathbf{S}}_{\ell-1}) \leftarrow \textsc{PerStepBwd}(\mathbf{d}\tilde{\mathbf{S}}_\ell, \mathbf{0}, \ldots)$ ▷ pass $\mathbf{do}=\mathbf{0}$
end for
write $\mathbf{d}\tilde{\mathbf{S}}_{b,\text{init}} \leftarrow \mathbf{d}\tilde{\mathbf{S}}_{(b-1)B}$ to HBM
return per-noisy-token gradients

Algorithm 2 Route I fill-in backward (one program per chunk c).

Require: chunk c's clean tokens, HBM-resident $\{\mathbf{S}_{(b-1)B}\}_{b \in \text{chunk}(c)}$ and $\{\mathbf{d}\tilde{\mathbf{S}}_{b,\text{init}}\}_{b \in \text{chunk}(c)}$

$\mathbf{dh} \leftarrow \mathbf{0}$ ▷ accumulator for Eq. (13)
for $j = M-1, \ldots, 0$ **do**
    $b \leftarrow cM + j + 1$; $\mathbf{S} \leftarrow \mathbf{S}_{(b-1)B}$
    $\textsc{CleanTransBwd}\big(\mathbf{dh}, \mathbf{S}, \{(\mathbf{k}_t, \hat{\mathbf{v}}_t, G_t)\}_{t \in \text{block}(b)}\big)$ ▷ writes $(\mathbf{dk}_t, \mathbf{d}\hat{\mathbf{v}}_t, dG_t)$
    $\mathbf{dh} \leftarrow \mathbf{d}\tilde{\mathbf{S}}_{b,\text{init}} + \exp\big(G_{bB} - G_{(b-1)B}\big) \, \mathbf{dh}$ ▷ Eq. (13)
end for
write $\mathbf{dh}^{\text{inject}}_{[c]} \leftarrow \mathbf{dh}$ to HBM

Algorithm 3 CHUNK-THEN-REFINE (pipeline).

fwd: $(\mathbf{o}, \{\mathbf{S}_{[c]}\}, \hat{\mathbf{v}}, G) \leftarrow$ Clean-fwd (§D.4.1)
fwd: $\{\mathbf{S}_{(b-1)B}\} \leftarrow$ Fill-in-fwd by repeated application of f^{blk} (Eq. (11)) ▷ L/B states to HBM
fwd: for each block b **in parallel**: run Eq. (12) for B steps from $\mathbf{S}_{(b-1)B}$
bwd: noisy-bwd (Alg. 1, for all b in parallel)
bwd: fill-in-bwd (Alg. 2, for all c in parallel)
bwd: final clean grads $\leftarrow \textsc{ChunkBwd}(\mathbf{do}, \mathbf{dh}^{\text{inject}}_{[c]})$

Route I's two costs are an L/B-sized tensor of block-boundary clean states in HBM and a three-kernel serial backward chain; Route II (§D.4.6) eliminates both, and §D.4.7 quantifies the difference.

D.4.6 Route II: Fused Two-Stream

Route II collapses forward stages 2–3 (and the corresponding backward stages) into a single fused kernel per direction. The forward kernel launches one program per chunk, loads $\mathbf{S}_{[c]}$ into registers, and interleaves noisy-block forward passes with applications of f^{blk} that advance the in-register clean state. No block-boundary clean state is written to HBM. The backward launches one program per (c, i_V) pair, where $i_V \in \{1, \ldots, \lceil d_v/b_v \rceil\}$ indexes the value-dimension tile, and sweeps the same structure in reverse, invoking the primitives of §D.4.3 as named sub-procedures.

Strided clean-state checkpoints. The backward requires $\mathbf{S}_{(b-1)B}$ for every block b in the chunk, but the forward keeps only $\mathbf{S}_{[c]}$ resident. A direct remedy is to store every block-end clean state to HBM (M snapshots per chunk, L/B snapshots per layer per sample), which reproduces Route I's fill-in tensor and its L/B footprint. We instead adopt a *strided-checkpoint* tensor parameterised by the *checkpoint stride* $S \in \{1, 2, \ldots, M\}$: within each chunk only every S-th block-boundary clean state is stored, giving $N_{\text{ckpt}} = M/S$ snapshots per chunk and $N_C \cdot M/S = L/(BS)$ snapshots

per layer per sample. The remaining $(S-1)/S$ fraction of block-end states are not persisted; the backward reconstructs each by loading the nearest prior snapshot and applying at most $S-1$ copies of f^{blk} in registers (Fig. 10). S parameterises a time/memory trade-off: the HBM cost scales as $1/S$ and the per-block replay work scales as $S-1$. Defaults are chosen by block size (§D.4.6); on the Qwen3.5-2B shape at $B{=}1, S{=}16$ the checkpoint tensor is approximately 128 MiB, roughly $16\times$ smaller than Route I's approximately 2 GiB state tensor.

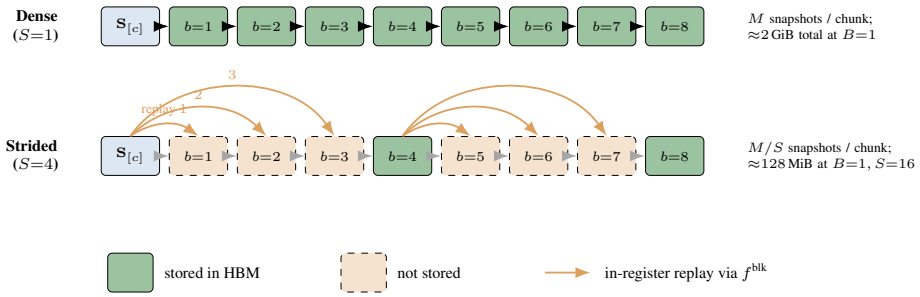


Figure 10: Dense ($S{=}1$, top) versus strided ($S{=}4$, bottom) checkpointing of the block-boundary clean states inside one chunk ($M{=}8$). Solid boxes are stored in HBM; dashed boxes are reconstructed in registers at backward time by applying $\leq S-1$ copies of f^{blk} from the nearest stored snapshot. Route II's HBM footprint scales as $L/(BS)$ versus Route I's L/B.

Algorithm 4 Strided clean-state checkpoint build (one program per chunk c).

Require: chunk-start state $\mathbf{S}_{[c]}$; clean tokens of chunk c; stride S
S $\leftarrow \mathbf{S}_{[c]}$
for $s = 0, 1, \ldots, M/S - 1$ **do**
  for $r = 0, 1, \ldots, S - 1$ **do**
    $b \leftarrow cM + sS + r + 1$; **S** $\leftarrow f_b^{\text{blk}}(\mathbf{S})$ ▷ Eq. (11), in registers
  end for
  write snapshot **S** to the s-th checkpoint slot of chunk c
end for

Choosing the stride. S is chosen so that Route II's extra HBM stays roughly constant independent of B: since the per-chunk snapshot count is $N_{\text{ckpt}} = M/S = C/(BS)$, we scale S with M so that N_{ckpt} stays small ($\mathcal{O}(1)$) per chunk. With $C{=}64$ fixed, the defaults are $S{=}16$ at $B{=}1$ ($N_{\text{ckpt}}{=}4$), $S{=}8$ at $B{=}2$ ($N_{\text{ckpt}}{=}4$), $S{=}2$ at $B{=}4$ ($N_{\text{ckpt}}{=}8$), and $S{=}\min(8, M)$ otherwise; across these configurations $N_{\text{ckpt}} \in \{1, 4, 8\}$ and the checkpoint tensor occupies between 32 MiB and 256 MiB on the Qwen3.5-2B shape. The per-block replay cost is at most $S-1$ evaluations of f^{blk} in registers and remains register-bound because each f^{blk} is a single $\mathbb{R}^{d_k \times d_v}$ update that fits in one tile.

Fused forward kernel. The forward traverses each chunk block-by-block in order: at block j the in-register clean state serves as the noisy-recurrence initial state; after the B noisy-token forward steps, the clean state is advanced by f^{blk} and the iteration proceeds. Only noisy outputs are written to HBM; the clean output is produced by the separate chunkwise kernel of §D.4.1.

Algorithm 5 Route II forward kernel (one program per chunk c).

Require: $\mathbf{S}_{[c]}$, clean tokens of chunk c, noisy tokens of chunk c
S $\leftarrow \mathbf{S}_{[c]}$ ▷ in registers
for $j = 0, 1, \ldots, M - 1$ **do**
  $b \leftarrow cM + j + 1$; $\tilde{\mathbf{S}}_{b,\text{init}} \leftarrow \mathbf{S}$ ▷ in registers; no HBM write
  run Eq. (12) forward on block b; emit $\tilde{\mathbf{o}}_\ell$ for $\ell \in \text{block}(b)$
  S $\leftarrow f_b^{\text{blk}}(\mathbf{S})$ ▷ advance clean state, Eq. (11)
end for

Fused backward kernel. The backward traverses each chunk block-by-block in reverse; the body is given in Alg. 6 and visualized in Fig. 11. All per-block replay, reverse sweep, and clean-transition

work is performed in registers; the kernel writes only the single chunk-boundary tensor $\mathbf{dh}_{[c]}^{\text{inject}}$ to HBM, which CHUNKBWD consumes exactly as in Route I.

Algorithm 6 Route II backward kernel (one program per chunk c, value tile i_V).

Require: $\mathbf{S}_{[c]}$, strided checkpoints, clean and noisy tokens of chunk c, $\mathbf{d\tilde{o}}$

$\mathbf{dh} \leftarrow \mathbf{0}$ ▷ in registers
for $j = M-1, \ldots, 0$ **do**
  $b \leftarrow cM + j + 1$
  REPLAY: $\mathbf{S}_{(b-1)B} \leftarrow$ load nearest checkpoint, apply $\leq S-1$ copies of f^{blk}
  run Eq. (12) forward for B steps (re-compute caches in registers)
  $\left(\mathbf{d\tilde{q}}, \mathbf{d\tilde{k}}, \mathbf{d\tilde{v}}, d\tilde{g}, d\tilde{\beta}, \mathbf{d\tilde{S}}_{b,\text{init}}\right) \leftarrow$ block-end-readout reverse sweep (Eq. (15) + PERSTEPBWD with $\mathbf{do}=\mathbf{0}$)
  $(\mathbf{dk}_t, \mathbf{d\hat{v}}_t, dG_t)_{t\in\text{block}(b)} \leftarrow \text{CLEANTRANSBWD}(\mathbf{dh}, \mathbf{S}_{(b-1)B}, \ldots)$
  $\mathbf{dh} \leftarrow \mathbf{d\tilde{S}}_{b,\text{init}} + \exp\left(G_{bB} - G_{(b-1)B}\right)\mathbf{dh}$ ▷ Eq. (13)
end for
write $\mathbf{dh}_{[c]}^{\text{inject}} \leftarrow \mathbf{dh}$ ▷ consumed by CHUNKBWD

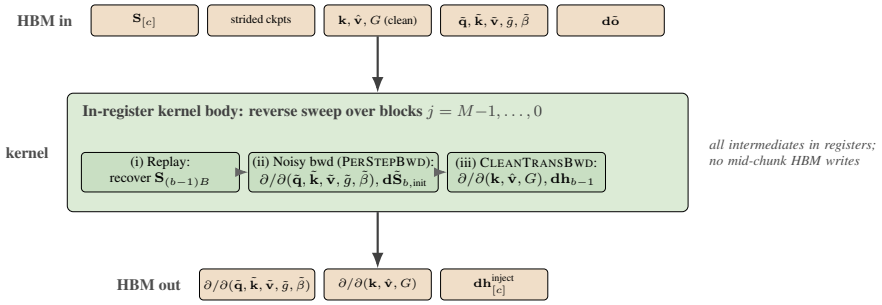


Figure 11: Route II backward kernel dataflow for a single program (one chunk c, one value-dimension tile). Five inputs are read from HBM (top); the reverse sweep of Alg. 6 runs entirely in registers (middle); three outputs are written back to HBM (bottom). CHUNKBWD (§D.4.3) consumes $\mathbf{dh}_{[c]}^{\text{inject}}$ afterwards.

Parallelism structure. The backward of Fig. 8 groups into three parallelism tiers once Levels 1 and 2 (within a block) are folded together: *tier A* performs one block-end output contraction plus B per-step reverse steps inside a block; *tier B* sweeps M such blocks inside a chunk via CLEANTRANSBWD and the block-level recursion; and *tier C* sweeps N_C chunks across the sequence via CHUNKBWD. Route II performs tiers A and B entirely in registers inside the fused kernel and delegates tier C to CHUNKBWD, the same chunk-level matmul scan used by the AR baseline. Since tier B is strictly within-chunk, chunks are independent at the fused-kernel level, which enables Route II's grid of $N_C \cdot B_{\text{batch}} \cdot H \cdot \lceil d_v/b_v \rceil$ programs; Route I combines tier B with tier C in the same kernel and is therefore restricted to a grid of $B_{\text{batch}} \cdot H \cdot \lceil d_v/b_v \rceil$ programs.

D.4.7 Implementation comparison

The two routes produce identical gradients, so the selection between them is governed by resource trade-offs. Route I has fewer moving parts: three small kernels per direction and no strided-checkpoint bookkeeping. Route II reduces peak memory and increases backward parallelism. In the deployed **FLARE** configuration ($B < 16$) Route II dominates on every axis considered below; at $B \geq 16$ Route I is faster on latency while Route II retains the peak-memory advantage. The three axes are listed in order of importance at small B.

Peak memory. Route I materializes an L/B-scaling tensor of block-boundary clean states: $\{\mathbf{S}_{(b-1)B}\}_{b=1}^{L/B}$ is written in forward stage 2 and kept alive until consumed by the block-local noisy forward (and, in general, by the fill-in backward). Its size is

$$\text{HBM}_{\text{Route I, states}} = \underbrace{L/B}_{\text{grows as } B \text{ shrinks}} \cdot H \cdot d_k \cdot d_v \cdot (\text{bytes per element}),$$

which at $B{=}1$ on the Qwen3.5-2B shape is approximately 2 GiB per layer per sample in bf16; across the backbone's 18 GDN layers this exhausts the available HBM at the small block sizes used in **FLARE** training. Route II does not form this tensor: $\mathbf{S}_{(b-1)B}$ is recomputed in registers from the nearest strided checkpoint and used immediately. The only extra HBM Route II allocates is the strided-checkpoint tensor of Alg. 4, which scales as $L/(BS)$ and at the $B{=}1$ default $S{=}16$ on the same shape is approximately 128 MiB, roughly $16\times$ smaller.

Per-chunk parallelism. Route I's backward combines cross-chunk propagation with the per-block reverse sweep in a single kernel, so its noisy-backward grid is $B_{\text{batch}}{\cdot}H{\cdot}\lceil d_v/b_v\rceil$ (where b_v is the value-dimension tile size). On the Qwen3.5-2B shape with $B_{\text{batch}}{=}1$, $H{=}16$, $b_v{=}32$ this yields 64 programs, well below the 108 SMs of an A100. Route II delegates the cross-chunk propagation to ChunkBwd, making each chunk's fused kernel independent; the grid becomes $N_C \cdot B_{\text{batch}} \cdot H \cdot \lceil d_v/b_v\rceil$ and at $N_C{=}64$ reaches 4096 programs, a $64\times$ increase that saturates SM occupancy. The shared cross-chunk ChunkBwd kernel itself runs the same 64-program ($B_{\text{batch}}{\cdot}H{\cdot}\lceil d_v/b_v\rceil$) grid in both routes and in the AR baseline.

HBM hand-off tensors. Route I's backward runs three kernels in strict sequence; each waits for the previous kernel's output tensor to be fully written to HBM before launching. The two internal hand-off tensors have sizes $L/B{\cdot}H{\cdot}d_k{\cdot}d_v$ (noisy-init gradients) and $N_C{\cdot}H{\cdot}d_k{\cdot}d_v$ (chunk-boundary gradients); the former again carries the L/B factor. Route II collapses the first two stages and emits only the chunk-boundary tensor $\mathbf{dh}^{\text{inject}}_{[c]}$ (N_C-sized) directly into ChunkBwd, halving the hand-off count and removing the L/B-sized intermediate.

Kernel latency and peak memory benchmark. Tables 4 and 5 report an empirical wall-clock and peak-memory comparison of Route I versus Route II for the GDR and the ShortConv kernels, respectively, across block sizes $B \in \{1, 2, 4, 8, 16, 32\}$ on the Qwen3.5-2B training shape. On the GDR, Route II cuts total wall-clock from 135.1 ms to 37.7 ms and peak memory from 18.14 GiB to 456 MiB at $B{=}1$; Route I overtakes Route II at $B{\geq}16$ once the chunk-level matmul saturates tensor cores. For the ShortConv, Route II is faster than Route I at every B in the sweep and holds peak memory at $\sim$294 MiB throughout. We therefore auto-dispatch Route II for the GDR at $B{<}16$ and Route I at $B{\geq}16$, and use Route II for the ShortConv at every block size.

Table 4: **GDR: Route I (Chunk-then-Refine) vs. Route II (Fused Two-Stream)** wall-clock and peak memory at varying diffusion-block size B. One A100-80GB, bf16, $L = 8192$ (half clean, half noisy) on the Qwen3.5-2B GDN shape ($H = 16$, $d_k = d_v = 128$). Wall-clock is mean $\pm$ std (ms) over three outer runs of ten iterations each (trimmed mean inside each outer run); peak is the maximum live-memory delta (MiB) across the combined forward and backward window. Bold marks the faster / smaller of the two routes at each block size.

Block size B	Route	fwd (ms)	bwd (ms)	total (ms)	peak (MiB)
1	Route I	26.36 ± 0.72	108.74 ± 15.27	135.10	18 576
1	Route II	$\mathbf{15.99 \pm 1.10}$	$\mathbf{21.70 \pm 0.19}$	**37.69**	**456**
2	Route I	$\mathbf{16.70 \pm 1.10}$	61.45 ± 0.37	78.14	9 360
2	Route II	19.19 ± 1.63	$\mathbf{21.22 \pm 0.29}$	**40.42**	**456**
4	Route I	$\mathbf{13.80 \pm 0.66}$	48.35 ± 3.35	62.14	4 752
4	Route II	17.64 ± 1.31	$\mathbf{20.61 \pm 0.02}$	**38.26**	**712**
8	Route I	$\mathbf{12.00 \pm 0.89}$	39.07 ± 1.29	51.07	2 448
8	Route II	15.65 ± 1.46	$\mathbf{23.19 \pm 0.36}$	**38.84**	**378**
16	Route I	$\mathbf{6.61 \pm 0.71}$	$\mathbf{12.07 \pm 0.50}$	**18.68**	1 296
16	Route II	20.00 ± 4.59	28.61 ± 2.16	48.61	**378**
32	Route I	$\mathbf{9.27 \pm 1.27}$	$\mathbf{11.76 \pm 2.29}$	**21.03**	720
32	Route II	16.08 ± 0.78	36.96 ± 0.34	53.04	**378**

D.5 1D Causal ShortConv: implementation routes

The 1D Causal ShortConv of width W (typically $W{=}4$) is the second sub-component of a GDN layer. It applies a depthwise causal 1D convolution to each channel of the input stream before the linear

projections that feed the GDR of §D.4, conditioning each token's $(\mathbf{q}_t, \mathbf{k}_t, \mathbf{v}_t)$ on its local temporal context. Under **FLARE**'s two-stream training, ShortConv is subject to the same class of constraints analyzed for the GDR in §D.3–§D.4.7: each noisy output position must read from the clean stream whenever its W-wide receptive field extends across the block boundary, while the clean stream is unaffected by the noisy stream. This subsection applies the GDR construction to ShortConv, identifies the two structural simplifications that reduce its cost, and states the two implementation routes in the same Route I / Route II terminology.

D.5.1 Two-stream contract for a width-W causal 1D convolution

Let $\mathbf{w} \in \mathbb{R}^{D\times W}$ be the depthwise filter applied by a standard single-stream causal convolution to a length-L input of D-dimensional tokens; we write w_i for its i-th lag slice (acting channel-wise). Under the two-stream contract, the clean-side output y^{c}_t at position t is computed from the clean stream in the standard form:

$$y^{\mathrm{c}}_t \;=\; \sum_{i=0}^{W-1} w_i \, x^{\mathrm{c}}_{t-i}, \tag{19}$$

where $x^{\mathrm{c}}_{t-i} \in \mathbb{R}^D$ is the clean-stream input at lag i (with out-of-range entries treated as zero) and $w_i \, x^{\mathrm{c}}_{t-i}$ denotes the depthwise (channel-wise) product. The noisy-side output $\tilde{y}_\ell$ at position ℓ inside block b (positions $(b{-}1)B+1, \ldots, bB$) reads from the *noisy* stream for lags that remain within block b and from the *clean* stream for lags that fall before block b's start:

$$\tilde{y}_\ell \;=\; \sum_{i=0}^{W-1} w_i \, z_{\ell,i}, \qquad z_{\ell,i} \;=\; \begin{cases} \tilde{x}_{\ell-i} & \text{if } \ell - i \geq (b{-}1)B + 1, \\ x^{\mathrm{c}}_{\ell-i} & \text{otherwise,} \end{cases} \tag{20}$$

where $z_{\ell,i} \in \mathbb{R}^D$ is the lag-i input read by the noisy output at position ℓ, selected from the noisy stream when the lag position falls inside block b and from the clean stream otherwise. The condition $\ell - i \geq (b{-}1)B + 1$ is the ShortConv analog of the clean-to-noisy visibility rule stated in Challenge 1 (§D.3): noisy tokens read from the clean stream only across the block boundary. Figure 12 illustrates the rule for $W{=}4$.

D.5.2 Analysis of implementation difficulty compared to Gated Delta Rule

Two structural simplifications reduce the cost of the ShortConv case relative to the GDR case.

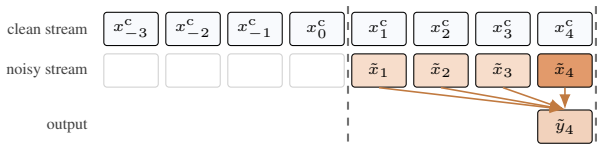
The noisy initial state is a raw tensor slice. The GDR's block-boundary clean state is a $d_k \times d_v$ matrix obtained by running B per-step recurrence updates (Eq. (9)) starting from $\mathbf{S}_{[c]}$, incurring a serial replay or checkpoint lookup (§D.4.5, §D.4.6). The ShortConv seed required by Eq. (20) is the last $W{-}1$ clean tokens preceding block b's start, $\big(x^{\mathrm{c}}_{(b-1)B-(W-1)+1}, \ldots, x^{\mathrm{c}}_{(b-1)B}\big)$, which is a slice of the clean-stream input tensor. No recurrence is replayed and no checkpoint tensor is constructed; the seed is read directly from the clean activations already consumed by the clean-side convolution.

The backward has no block-level state recursion. The GDR's backward (Fig. 8) composes four levels: a block-end readout (Level 1, Eq. (15)), a per-step reverse sweep within a block (Level 2, Eq. (14)), a block-level reverse recursion within a chunk through f^{blk}_b (Level 3, Eq. (13)), and a cross-chunk matmul scan on chunk-boundary states (Level 4, Eq. (16)). The ShortConv backward omits the block-level state recursion (Level 3): the gradient deposited by each noisy output into its clean-side lag positions is scattered directly into the corresponding positions of $\mathbf{dx}$, without a block-level state-gradient recursion, because the noisy-side seed is not a recurrence terminus. The cross-chunk Level 4 reduces to the standard clean-side causal-conv backward and requires no additional hand-off.

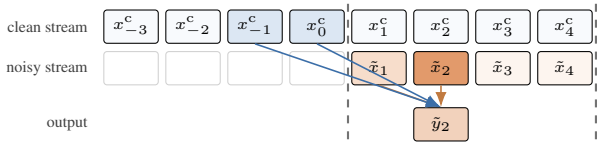
D.5.3 Route I: batched initial-state

The ShortConv analog of Route I materializes the noisy initial states in HBM and then invokes the stock single-stream causal-conv kernel in batch. Forward:

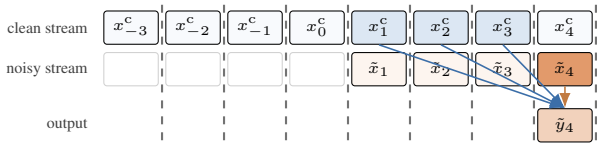
Backward mirrors this structure. The batched noisy backward emits $\mathbf{d\tilde{x}}$, the filter/bias gradients, and a per-block gradient $\mathbf{dH}^{\mathrm{init}}[b{-}1]$ with respect to each prepended context. A final scatter-add sends



(a) Interior of the block. Offset $j{=}3 \geq W{-}1$, so all four lags stay inside the active block (noisy reads).



(b) Near the block start. Offset $j{=}1$, so the first $j{+}1{=}2$ lags read noisy inside the active block while the remaining $W{-}1{-}j{=}2$ lags cross the boundary and read clean from the preceding block.



(c) Small B ($B{=}1$). Every noisy token is a length-1 block. Only lag 0 (noisy, self) stays inside the active block; the remaining $W{-}1{=}3$ lags (clean) each cross a distinct block boundary into a different preceding clean block.

Figure 12: Two-stream read pattern of the width-$W{=}4$ causal 1D ShortConv for a noisy output $\tilde{y}_\ell$, shown under three regimes: (a) interior ($j \geq W{-}1$); (b) straddle ($j < W{-}1$, clean lags in the single preceding block); (c) small-B ($B{=}1$, each non-self clean lag lands in a different preceding block). Per-panel streams: clean (top row), noisy (middle row), active output (bottom row). Orange arrows: noisy reads; blue arrows: clean reads; dashed lines: block boundaries; faded tokens: irrelevant to $\tilde{y}_\ell$.

Algorithm 7 ShortConv Route I forward.

Require: clean tokens $\{x_t^c\}$, noisy tokens $\{\tilde{x}_\ell\}$, filter $\mathbf{w}$, block size B

1: run the stock causal-conv kernel on $\{x_t^c\}$ to produce $\{y_t^c\}$
2: build $\mathbf{H}^{\text{init}} \in \mathbb{R}^{L/B \times D \times (W-1)}$ by slicing the last $W{-}1$ clean tokens before each noisy block (zero-pad at document starts)
3: reshape $\{\tilde{x}_\ell\}$ into L/B independent length-B sub-sequences
4: run the stock causal-conv kernel in batch over the L/B sub-sequences, each with $\mathbf{H}^{\text{init}}[b{-}1]$ as its prepended context
5: return $\{y_t^c\}$ and $\{\tilde{y}_\ell\}$

each $\mathbf{dH}^{\text{init}}[b{-}1]$ back to the last $W{-}1$ positions of $\mathbf{dx}$ that it was read from, and the clean-side causal-conv backward is run once on the full clean stream. The state tensor $\mathbf{H}^{\text{init}}$ lives in HBM throughout; its size is $L/B \cdot D \cdot (W-1)$ bytes (times precision), the ShortConv analog of Route I's L/B-scaling term for the GDR.

D.5.4 Route II: fused two-stream

Route II replaces both the init-state construction and the batched noisy conv with a single fused Triton kernel that implements Eq. (20) in place.

A companion fused kernel implements the backward: for each noisy output position ℓ and lag i, it computes the contribution $w_i\, \partial\mathcal{L}/\partial\tilde{y}_\ell$ and atomically adds it to $\mathbf{d\tilde{x}}_{\ell-i}$ if $\ell - i \geq \ell_{\text{blk}}$, or to $\mathbf{dx}_{\ell-i}$ otherwise; filter and bias gradients are accumulated into per-tile partial tensors and reduced. The clean-side output and its gradient are produced by the stock single-stream causal-conv forward and backward unchanged from the AR baseline. No L/B-scaling HBM tensor is materialized and no batched noisy-conv kernel is launched.

Algorithm 8 ShortConv Route II forward (one program tile per (ℓ-chunk, D-tile) pair).

Require: clean tokens $\{x_t^c\}$, noisy tokens $\{\tilde{x}_\ell\}$, filter $\mathbf{w}$, block size B
for each noisy output position ℓ in the tile **do**
  compute ℓ's block start $\ell_{\text{blk}} = \lfloor (\ell - 1)/B \rfloor \cdot B + 1$
  for lag $i = 0, 1, \ldots, W - 1$ **do**
    $z_{\ell,i} \leftarrow \tilde{x}_{\ell-i}$ if $\ell - i \geq \ell_{\text{blk}}$, else $x^c_{\ell-i}$
  end for
  $\tilde{y}_\ell \leftarrow \sum_i w_i \, z_{\ell,i}$
end for

D.5.5 Implementation comparison

The three axes of §D.4.7 reapply in a reordered ranking, since the ShortConv state is small relative to the GDR state. In the deployed **FLARE** configuration (W=4 with silu on CUDA, small B), Route II is preferable on all three axes; outside this configuration Route I remains a fallback.

Per-chunk parallelism. Route I's batched noisy conv has grid $(L/B) \cdot \lceil D/b_d \rceil$, where b_d is the channel-dimension tile size; Route II's fused kernel instead tiles along the t-dimension, with grid $N_t \cdot \lceil D/b_d \rceil$ for N_t the t-dimension chunk count, decoupling the launchable-program count from the block size B.

Kernel-fusion overhead. Route I issues three kernel calls per direction (forward: clean conv, init-state build, batched noisy conv; backward: batched noisy backward, init-state scatter, clean conv backward); Route II issues two (clean conv and fused two-stream conv, per direction). The launch-overhead savings are material at the short sequence lengths typical of ShortConv layers.

Peak memory. Route I's $\mathbf{H}^{\text{init}}$ has size $L/B \cdot D \cdot (W - 1)$, a few hundred MB per layer per sample at the smallest block sizes and shrinking with B. This is much smaller than the analogous L/B-scaling tensor in the GDR case (gigabytes), so memory is not the dominant factor in selecting Route II for the ShortConv.

Kernel latency and peak memory benchmark. Table 5 reports the empirical wall-clock and peak-memory comparison of ShortConv Route I versus Route II at $B \in \{1, 2, 4, 8, 16, 32\}$ on the Qwen3.5-2B ShortConv shape. Route II is faster than Route I at every block size in the sweep and keeps peak memory within ∼294 MiB throughout, so **FLARE** uses Route II for the ShortConv at every block size.

Table 5: **1D Causal ShortConv: Route I (batched initial-state) vs. Route II (fused two-stream)** wall-clock and peak memory at varying B. Same hardware and setup as Table 4, on the Qwen3.5-2B ShortConv shape ($D = 6144$, $W = 4$, t-dimension tile size $BT = 64$). Bold marks the faster / smaller of the two routes at each block size.

Block size B	Route	fwd (ms)	bwd (ms)	total (ms)	peak (MiB)
1	Route I	13.41 ± 0.86	42.65 ± 0.66	56.06	1 452
1	Route II	$\mathbf{6.26 \pm 0.60}$	$\mathbf{3.40 \pm 0.11}$	**9.65**	**294**
2	Route I	$\mathbf{11.99 \pm 1.12}$	21.07 ± 0.22	33.06	816
2	Route II	13.22 ± 10.73	$\mathbf{4.25 \pm 1.09}$	**17.47**	**294**
4	Route I	12.38 ± 7.43	11.06 ± 0.08	23.44	372
4	Route II	$\mathbf{7.00 \pm 0.98}$	$\mathbf{3.56 \pm 0.34}$	**10.55**	**294**
8	Route I	11.44 ± 0.54	6.30 ± 0.08	17.74	330
8	Route II	$\mathbf{6.86 \pm 0.51}$	$\mathbf{3.09 \pm 0.15}$	**9.95**	**294**
16	Route I	16.72 ± 10.98	3.64 ± 0.01	20.36	309
16	Route II	$\mathbf{6.71 \pm 0.14}$	$\mathbf{2.93 \pm 0.27}$	**9.64**	**294**
32	Route I	9.11 ± 1.39	$\mathbf{2.43 \pm 0.01}$	11.54	299
32	Route II	$\mathbf{6.86 \pm 0.20}$	2.86 ± 0.12	**9.72**	**294**

Summary. We auto-select Route II for every benchmarked ShortConv configuration supported by the fused kernel ($W \leq 4$ with silu/swish or no activation on CUDA), and retain Route I as a fallback for configurations outside it (e.g. alternative activations or larger filter widths).

D.6 Document-packed training

This subsection specifies the document-packing guarantees required by Challenge 2 of §D.3. The kernels in §D.4–§D.5 are described as operating on a single L-token document; for training throughput, multiple shorter documents are packed into each training sequence with per-document start offsets $\mathbf{cu} = (c_0, c_1, \ldots, c_{N_{\text{doc}}})$, $c_0 = 0$, $c_{N_{\text{doc}}} = L$ (the standard `cu_seqlens` layout). The two-stream mask is defined per-document; under packing, the block-diffusion visibility rule is composed with a document-level mask that forbids cross-document attention, so each document is trained as if in isolation. This subsection states how the GDR and 1D Causal ShortConv kernels of §D.4–§D.5 preserve single-document semantics under packing.

Constraint: document starts are block-aligned. Every entry of $\mathbf{cu}[:-1]$ must be a multiple of the diffusion-block size B, i.e. every document's first token must also be a block start. This is a hard precondition: the two-stream block mask of §2.2 is defined on block boundaries, so a document starting in the interior of a block would cause its clean-to-noisy visibility to straddle a document boundary, violating the doc-level visibility contract. The final packed tail ($c_{N_{\text{doc}}}$) is permitted to be a partial block; the implementation zero-pads the trailing tokens up to a block boundary. A single helper enforces the check at every two-stream entry point; violations abort the training step with an explicit error.

Three kernel-level guards for cross-document correctness. Under packing the three two-stream kernels of §D.4 and §D.5 each carry one document-boundary guard that the single-document versions do not need.

- **Noisy recurrence's initial state at document starts** (Route II forward kernel of §D.4.6; same boundary also handled by Route I's fill-in kernel). The fused two-stream kernel normally seeds the noisy recurrence of block b from the clean state $\mathbf{S}_{(b-1)B}$ (Eq. (12)). When block b is the first block of a document, $\mathbf{S}_{(b-1)B}$ would be read from the previous document's tail chunk and carry its clean-prefix information across the document boundary. The kernel detects this case (via the per-program `is_doc_first_chunk` flag built from $\mathbf{cu}$) and zeroes the noisy initial state instead. The forward's noisy block b=1 of every document therefore starts from $\mathbf{0}$, matching the single-document semantics where each document is trained independently.
- **Cross-chunk gradient shift at document boundaries** (Route I fill-in backward of §D.4.5; also analogous code path inside the clean-side cross-chunk scan of §D.4.3). The normal cross-chunk shift turns $\partial\mathcal{L}/\partial\mathbf{S}_{[c]}$ (a chunk-boundary gradient) into the previous chunk's boundary gradient $\mathbf{dh}_{[c-1]}$ for the scan. At a document boundary this shift must not cross: the boundary gradient of a new document's first chunk has no corresponding end of a preceding chunk *in the same document* to flow into. The document-aware shift implements this by masking the cross-document assignment to zero; concretely, the contribution $\mathbf{dh}_{[c-1]} \mathrel{+}= \mathbf{dh}_{[c]}$ is taken if chunks $c-1$ and c are in the same document, and zeroed otherwise. Per-document chunk counts and offsets are computed once from $\mathbf{cu}$ at the start of the backward step.
- **ShortConv boundary read of the first noisy block of each document** (Route II fused ShortConv kernel of §D.5.4). The lag-read rule of Eq. (20) lets a noisy output position read the preceding clean block when its receptive field straddles the block boundary. At a document boundary this read would cross into the previous document's clean tokens. The kernel avoids this by bounds-checking every read with `src >= 0` and `src < doc_T` after offsetting by the document's `bos`; reads outside the current document contribute zero. For Route I the analogous behavior is enforced by zeroing the `initial_states` tensor at every document start, and by masking the corresponding $\mathbf{dh}_0$ gradient in the backward so that no gradient flows back into the preceding document's trailing clean tokens.

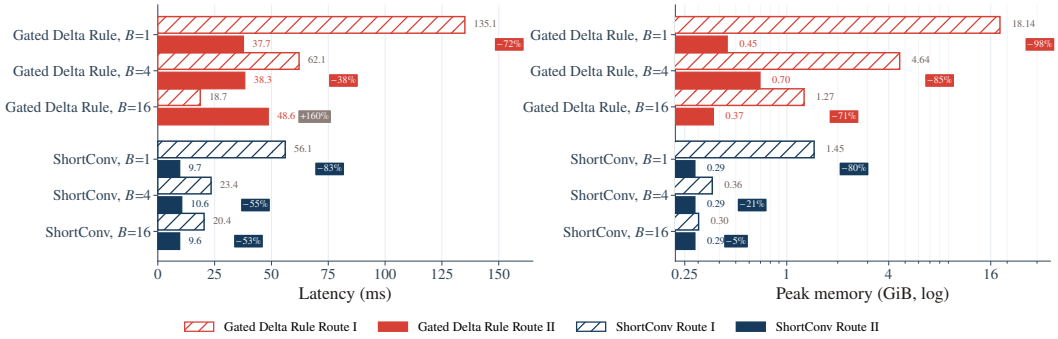
Variable-length kernel support. In addition to the three guards above, every kernel in §D.4–§D.5 accepts a `cu_seqlens` argument with a compile-time `IS_VARLEN` switch. When `cu_seqlens` is

provided, each Triton program loads its per-document `bos/eos` offsets from `cu` and uses them to translate an in-program position into the correct global token index, replacing the default $i_n \cdot T$ layout. This is standard variable-length kernel infrastructure inherited from the FLA reference implementation and is not FLARE-specific; packed and unpacked inputs share the same kernel code path.

Consequence for the cost model. The quantitative claims of §D.4.7 and §D.5.5 (HBM scaling, backward-grid size, hand-off count) are stated in terms of a packed sequence length $L = c_{N_{\text{doc}}}$; document count does not appear because the document-level guards above add $\mathcal{O}(1)$ work per document and no additional per-layer tensors. In particular, Route II's strided-checkpoint tensor has size $N_C \cdot (M/S) \cdot H \cdot d_k \cdot d_v$ (bytes $\times$ precision) regardless of how many documents are packed into the N_C chunks.

D.7 Route-level kernel microbenchmark

We first evaluate the schedules at the kernel level. Figure 13 shows Route II is most effective in the small-block regime diffusion training requires. For the GDR at B=1, it cuts total latency from 135.10 ms to 37.69 ms and peak memory from 18.14 GiB to 0.45 GiB. At larger B, Route I can win because its dense chunk-level matmul better saturates tensor cores. Since diffusion training requires small block sizes (FLARE uses $B = 4$), Route II is a structural requirement rather than an optimization: without it the small-B regime is infeasible in memory. We therefore use Route II for FLARE's small diffusion blocks and retain Route I as the large-block path. For ShortConv, Route II stays faster across all measured block sizes by avoiding many small block-level launches.



(a) Forward+backward latency (b) Peak memory

Figure 13: **Route-level kernel microbenchmark on Qwen3.5-2B shapes.** Horizontal paired bars compare Route I and Route II for the Gated Delta Rule and ShortConv kernels across diffusion block sizes $B \in \{1, 4, 16\}$. Labels on Route II report the relative change from Route I. Route II sharply reduces latency and memory in the small-block regime, while Route I overtakes the Gated Delta Rule at B=16 as dense chunk-level matmul better saturates tensor cores. Full sweeps are in Appendix D.

D.8 End-to-end MFU ablation

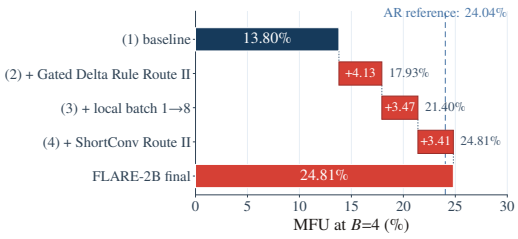


Figure 14: Training-time MFU of FLARE-2B at B=4 on 8×A100-80GB (bf16), as a horizontal waterfall. Bar (1) is the unoptimized baseline (Route I for both kernels, local batch 1); bars (2)–(4) show the cumulative MFU lift after each kernel-stack change, with per-step gains in percentage points. The final bar is the FLARE setting, closing the gap to the AR Qwen3.5-2B reference (24.04%). Full sweep and per-block HBM in Tab. 6 of Appendix D.8.

§4.2 reports the aggregate MFU lift produced by our kernel stack on FLARE-2B. This subsection breaks that number into its four incremental contributors and reports peak HBM alongside MFU at four block sizes, so that each MFU gain is attributable to a specific kernel change.

Setup. All runs train FLARE-2B on a single node of 8×A100 80 GB in bf16, using fully-sharded data parallelism (FSDP) and no tensor-, pipeline-, context-, or expert-parallelism. Attention runs under FlexAttention; to fit A100 shared memory at head dimension 256, we instruct it to reduce the kernel's pipeline depth to two stages. All runs use per-operator activation checkpointing (the PyTorch selective-AC policy with granularity "op"), a fused cross-entropy kernel, and the two training-side variance reduction options of §2.2: complementary-mask block pairs and antithetic mask sampling. The sequence length is L=4096, the global batch is 8× the local batch (8 for the local-batch-1 rows, 64 for the local-batch-8 rows), the training corpus is the Nemotron post-training v2 SFT mix (see Appendix F), and the GDR chunk size (§D.4.1) is C=64. Each row reports MFU and peak per-GPU HBM as the mean over measurement steps 11–30 of a 30-step warm-up-then-measure run. *Local batch* is the per-GPU batch B_{batch}; *block size* is the diffusion-block size B of §2.2. The AR reference is a plain Qwen3.5-2B run on the same tokens under a pure-AR single-stream model spec; it uses local batch 1 because that spec computes cross-entropy over the full logits tensor (the fused cross-entropy kernel above is only wired into the FLARE two-stream specs).

Table 6: **Incremental MFU ablation for FLARE-2B on 8×A100-80GB (bf16).** Each row adds one kernel-path change on top of the row above. **GDR** is the GDR kernel; **ShortConv** is the 1D causal ShortConv kernel (§D.5); **local batch** is the per-GPU batch; B is the diffusion-block size. **Bold** marks the best cell at each block size. **—** in the B=1 baseline row marks an out-of-memory run: Route I's L/B-scaling block-boundary state tensor, multiplied across the 18 GDN layers of the backbone, exceeds the 80 GiB HBM budget. The AR reference is a plain Qwen3.5-2B run on the same tokens.

Setting	GDR	ShortConv	`local batch`	MFU (%) at block size B				Peak HBM (GiB) at block size B			
				1	4	8	16	1	4	8	16
AR reference (Qwen3.5-2B)	—	—	1	24.04 ± 0.06 (single-stream)				20.5			
(1) baseline	Route I	Route I	1	—	13.80	16.73	22.06	—	18.3	13.8	12.7
(2) + GDR Route II	Route II	Route I	1	12.56	17.93	18.51	18.36	12.7	12.7	12.7	12.7
(3) + local-batch 1 → 8	Route II	Route I	8	14.20	21.40	22.26	22.14	49.0	45.8	45.9	45.6
(4) + ShortConv Route II (FLARE)	Route II	Route II	8	**24.20**	**24.81**	**23.69**	**22.55**	45.0	45.0	45.0	45.0

Per-row interpretation. *(1) Baseline.* Both linear-attention sub-components run the Chunk-then-Refine path of §D.4.5. At B=1 the row is out of memory; at B=16 the baseline reaches 22.06% because the L/B-scaling state tensor is 16× smaller and the chunkwise matmul achieves high tensor-core utilization. *(2) GDR Route II.* Switching the GDR to the fused two-stream kernel of §D.4.6 unblocks B=1 (12.56%) and caps peak memory at every block size to 12.7 GiB, because the L/B state tensor is no longer materialized in HBM. At B=4 the fused path is faster than Route I and lifts MFU by roughly +4 points (and by ∼ +1.8 at B=8); at B=16 it is slower, matching the crossover of §D.4.7 where the dense chunk-level matmul overtakes the fused kernel. *(3) Raising the per-GPU batch from 1 to 8.* The ∼ 6 GiB saved at B=4 by setting (2) is spent on a larger per-GPU batch. No kernel code changes here; the memory headroom is converted into tensor-core utilization. The step is worth ∼ +3.5–3.8 MFU points at $B \in \{4, 8, 16\}$ and pushes peak HBM to ∼ 46 GiB, still well within the 80 GiB budget. *(4) ShortConv Route II (FLARE).* The ShortConv fused kernel replaces L/B small per-block launches by a single fused kernel per sequence. The gain is largest at small B (+10 points at B=1), since launch overhead scales as L/B; it tapers off as B grows. The final row reaches 24.81% MFU at B=4, matching and slightly exceeding the pure-AR reference of 24.04% on identical tokens. This is the ceiling one expects for a hybrid-backbone dLLM: FLARE processes both the clean and the noisy streams, and the complementary-mask training of §2.2 trains each block under both the masked and the unmasked partition in a single forward, so FLARE performs roughly 4× the per-input-token attention work of the AR baseline; matching AR MFU therefore means the kernel stack has absorbed that extra work without dropping hardware efficiency.

The B=16 crossover is consistent with the microbenchmark. Reading Tab. 6 at fixed B=16, Route II for the GDR (row 2) underperforms Route I (row 1), for the same reason quantified in Tab. 4: at larger block sizes Route I's dense chunk-level matmul overtakes Route II's smaller launch footprint.

Setting (4) still matches or beats setting (1) at B=16 because the ShortConv Route II gain is additive and independent of the GDR choice.

Caveats. Two properties of the measurement window should be kept in mind when comparing to single-stream MFU numbers reported elsewhere. *(i)* The complementary-mask training of §2.2 trains each block under both the masked and the unmasked partition in a single forward, doubling the effective batch seen by attention; turning this off would approximately double all **FLARE** MFU numbers in the table but would no longer reflect the production training regime of the released **FLARE** checkpoints. *(ii)* The FlexAttention pipeline-depth reduction above costs roughly 1–2% of attention throughput uniformly across all four settings and the AR reference, so it is already folded into every row. Absolute numbers on H100 or in FP8 will differ; the relative gains between settings should nonetheless be representative of the kernel stack's contribution.

E Efficient Inference and Implementation

E.1 Overview

Section 2.3 defines the two decoding interfaces used by **FLARE**, summarized in Figure 16. This appendix does not redefine those interfaces; instead, it records the serving-side details needed to implement them on a hybrid-attention backbone. We first show the path-specific masks and block layouts, then describe the proposal policies, fused verification kernels, denoising-loop controls, and recurrent-state machinery used by our SGLang-based serving stack (Zheng et al., 2024).

Algorithms 9 and 10 give the operational view of the two interfaces, whose token-emission rules are Eqs. (4) and (5) in Section 2.3.

Algorithm 9. Diffusion-Trust Decoding.

Input: x, prefix $y_{<t}$, block size B, steps S
Initialize Active Block
$y_{t:t+B-1} \leftarrow \texttt{[MASK]}^B$, $R_1 \leftarrow \{t, \dots, t+B-1\}$
Parallel Block Denoising
for $s = 1, \dots, S$ until $R_s = \varnothing$ **do**
  $\tilde{z} \leftarrow f_\theta(x, y_{<t}, y_{t:t+B-1})$
  $\pi_i \leftarrow \sigma_{k,p,T}(\tilde{z}_i^{\text{shift}})$; $\hat{y}_i \sim \pi_i$
  $c_i \leftarrow \text{score}(\pi_i, \hat{y}_i)$
  $A_s \leftarrow \{i \in R_s : c_i \geq \gamma_s\}$; if $s = S$, set $A_s \leftarrow R_s$
  Update y and R_{s+1} by Eq. (4)
end for
Commit State
Run token-causal replay on finalized block
Append block to prefix and advance the live state

Algorithm 10. AR-Trust Decoding.

Input: x, prefix $y_{<t}$, held proposals $(d_i, q_i)_{i=1}^K$
Active Window Forward
$w \leftarrow [y_{<t}; d_{1:K}; \texttt{[MASK]}^{N-1}]$
$(z^c, \tilde{z}) \leftarrow f_\theta(x, w)$, $p_i \leftarrow \sigma_{k,p,T}(z_i^c)$
Clean-Stream Verification
for $i = 1, \dots, K$ **do**
  $a_i \sim \text{Bern}(\min\{1, p_i(d_i)/q_i(d_i)\})$
  if $a_i = 0$ **then**
    Emit correction from Eq. (5); **break**
  end if
  Emit d_i
end for
If all drafts are accepted, emit $y^\star \sim p_{K+1}$
Proposal Refill
$(d'_{1:K}, q'_{1:K}) \leftarrow \text{Draft}(\tilde{z})$

Eqs. (4) and (5) define the two regimes used by Algorithms 9 and 10. In AR-Trust, Eq. (5) accepts each held draft with probability $\min\{1, p_i(d_i)/q_i(d_i)\}$; following the acceptance-rate analysis of I-DLM (Yu et al., 2026), the plotted α abstracts the average success probability across horizon K, so speed improves only when noisy proposals align with the clean target. In Diffusion-Trust, Eq. (4) commits an active block over at most S denoising steps, so finalizing B tokens in S forwards yields the ideal ratio B/S. Figure 15 summarizes these analytical regimes and isolates the algorithmic control variables: α and K for AR-Trust, and B and S for Diffusion-Trust. System factors such as state replay, cache updates, and kernel scheduling are deliberately excluded here; measured throughput is reported in Section 4.3.

Notation. We write the prompt as $x_{1:n}$, the generation buffer as $y \in \mathcal{V}^L$ over vocabulary $\mathcal{V}$ (initially $y_i = \texttt{MASK}$ for $i > n$), the diffusion block size as B, the per-block denoise step budget as S_{step}, and the top-1 confidence $c_i = \max_{v \in \mathcal{V}} p_\theta(y_i = v \mid \cdot)$ under the checkpoint's output distribution

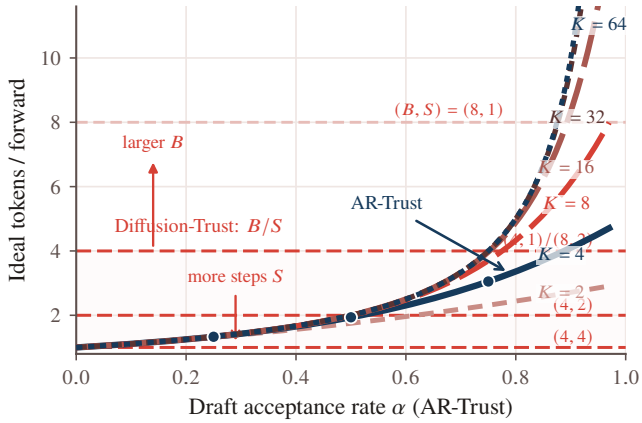


Figure 15: **Analytical decoding-speed regimes.** AR-Trust varies with acceptance rate α and horizon K; Diffusion-Trust scales with the block-to-step ratio B/S.

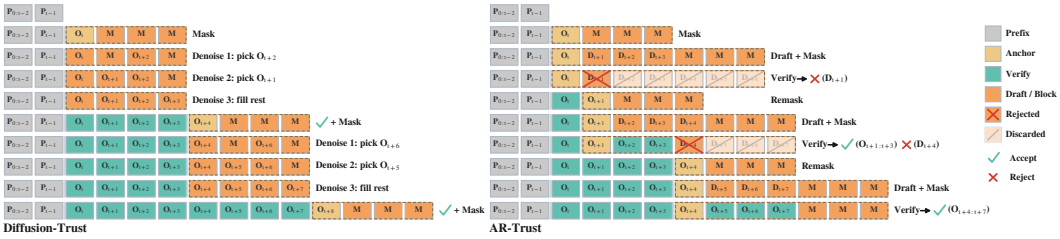


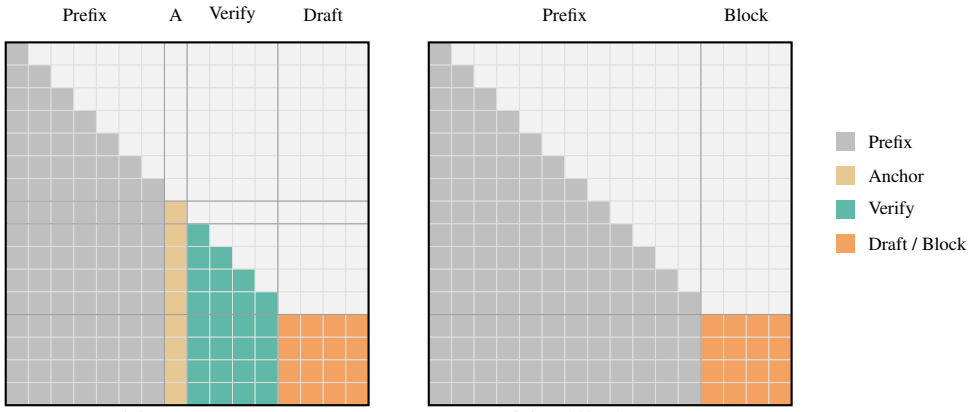
Figure 16: Two decoding procedures supported by a single trained FLARE checkpoint. **Left: Diffusion-Trust.** The noisy stream is trusted: a masked block is denoised in parallel over one or more refinement steps and then committed once finalized. **Right: AR-Trust.** The clean stream is trusted: noisy-stream logits propose multiple draft tokens, and clean-stream logits verify them left-to-right within the same checkpoint.

p_θ. A confidence threshold $\gamma \in (0, 1)$, temperature T, and truncation parameters (k, p) parametrize sampling. On the AR-Trust side N is the speculation horizon, $\mathbf{d} = (d_0, \ldots, d_{N-2})$ are the held draft tokens, and z, $\tilde{z}$ denote the clean-stream and noisy-stream logits emitted by one forward over the block (with z_i, $\tilde{z}_i$ their rows at position i); p_i, q_i are the target (clean-stream) and draft (noisy-stream) distributions at the i-th verify position, formed from z_i, $\tilde{z}_i$ under the policy in use. We write $\mathcal{B}_b = \{n + bB + 1, \ldots, n + (b{+}1)B\}$ for the position set of the b-th generation block. On the recurrent side, $\mathbf{S}$ is the GDN state (the same $\mathbf{S}$ as in Appendix D), L_{gdn} the number of GDN layers in the backbone, d_{state} the per-layer state dimension, B_{batch} the scheduler batch size, and $\mathbf{S}^{(r)}$ the state after r accepted updates inside a verify round. $\mathcal{K}$ denotes a request's KV cache slot set; writing a token y_i to $\mathcal{K}$ means appending its key and value tensors as new entries. The set of GDN states indexed by $\mathcal{K}$ is denoted $\mathbf{S}[\mathcal{K}]$.

E.2 AR-Trust Sampling

Algorithm 10 gives the method-level interface. Here we specify the serving block layout and proposal policies used to realize that interface. Figure 17(a) shows the attention pattern: the active block contains one anchor position (the last accepted token y_{anc}), $K \leq N{-}1$ verify positions that hold the draft tokens to be accepted or rejected, and $N{-}1$ MASK positions that produce the next round's drafts. The anchor and verify positions are token-causal to the prefix and to each other; the draft positions attend bidirectionally over the whole $2N{-}1$-wide active block and causally to the prefix.

Block layout. The active block has length $2N{-}1$ (the maximum-width, steady-state shape); shorter rounds left-pad with `MASK`. In a *cold-start* round the block is $[t_0, \texttt{MASK}, \ldots, \texttt{MASK}]$ with $1{+}2(N{-}1) = 2N{-}1$ positions: position 0 is a fresh clean token and the $2(N{-}1)$ masked positions produce the next round's $N{-}1$ draft tokens (the first $N{-}1$ MASK positions are unused padding on the cold start).



(a) AR-Trust **(b)** Diffusion-Trust

Figure 17: Attention patterns presented to the hybrid backbone during decoding. **AR-Trust** uses an anchor token and token-causal verify rows for clean-stream verification, followed by bidirectional draft rows whose noisy-stream logits propose the next speculative group. **Diffusion-Trust** uses a bidirectional active block behind a causal prefix; active positions attend to the prefix and to one another. Gray denotes prefix/context, tan denotes the anchor, teal denotes verify rows, and orange denotes bidirectional draft or active-block rows.

Algorithm 11 AR-Trust serving round (implementation details).

Require: draft policy $\pi \in \{\text{Exact-Truncated}, \text{Softmax-Argmax}, \text{Truncated-Argmax}\}$, y_{anc}, $d_0, \ldots, d_{K-1}$, $q_0, \ldots, q_{K-1}$ (or $\varnothing$ on cold start); horizon N;
if cold start **then** $y_{\text{blk}} \leftarrow [t_0, \texttt{MASK}^{2(N-1)}]$ ▷ length $2N-1$; last $N-1$ MASKs hold the next-round drafts
else $y_{\text{blk}} \leftarrow [y_{\text{anc}}, d_0, \ldots, d_{K-1}, \texttt{MASK}^{N-1}]$ ▷ length $1 + K + (N-1) \le 2N-1$
end if
run one forward over y_{blk} under the hybrid block mask; obtain clean logits z and noisy logits $\tilde{z}$
$r \leftarrow 0$; $\mathit{accepted} \leftarrow \varnothing$; $\mathcal{D}_{\text{tail}} \leftarrow \varnothing$
for $i = 0, 1, \ldots, K-1$ **do** ▷ verify pass
    form p_i from the clean logit z_i; retrieve or approximate q_i according to policy π; draw $u \sim \mathcal{U}(0, 1)$
    evaluate the modified rejection rule of policy π (Eqs. (5), (21), (22))
    if rejected **then**
        set $\mathcal{D}_{\text{tail}}$ to the policy's correction distribution at position i; **break**
    end if
    $\mathit{accepted} \leftarrow \mathit{accepted} \cup \{d_i\}$; $r \leftarrow r + 1$
end for
if $r = K$ **then**
    set $\mathcal{D}_{\text{tail}}$ to the clean-stream target distribution at the first masked position
end if
retain the first r new KV entries in $\mathcal{K}$ and release the remainder; update the recurrent state pool to $\mathbf{S}^{(r)}$ via the mechanism of §E.4
sample $y^\star \sim \mathcal{D}_{\text{tail}}$; draft $\mathbf{d}' = (d'_0, \ldots, d'_{N-2})$ from $\tilde{z}$ at the $N-1$ masked positions under policy π
emit $\mathit{accepted} \cup \{y^\star\}$; hold $\mathbf{d}'$ (and $\mathbf{q}'$ for Exact-Truncated) for the next round

In a *verify* round the block is $[y_{\text{anc}}, d_0, d_1, \ldots, d_{K-1}, \texttt{MASK}, \ldots, \texttt{MASK}]$ of length $1 + K + (N-1)$ where y_{anc} is the last accepted token and $d_0, \ldots, d_{K-1}$ are the $K \le N-1$ draft tokens held from the previous round; at steady state $K = N-1$ and the block reaches its full width $2N-1$. The same forward returns clean logits at positions $1, \ldots, K$, used to accept or reject each draft token, and fresh clean-plus-draft proposals from the $N-1$ masked tail positions. Algorithm 11 expands the compact interface of Algorithm 10 with serving state, KV-cache bookkeeping, and recurrent-state updates.

Draft policies. We consider three draft policies: *Exact-Truncated*, *Softmax-Argmax*, and *Truncated-Argmax*. They differ in how they form the proposal distribution q_i and, consequently, in the modified rejection rule and correction distribution they induce. The key subtlety is proposal consistency. Classical speculative decoding is distribution preserving only when q_i is the conditional law that actually produced d_i under the same proposal factorization assumed by verification. AR-Trust violates

this condition if one simply recomputes q_i from current noisy-stream logits, because the noisy stream drafts multiple positions in parallel under a masked block context rather than an autoregressive proposal context. Exact-Truncated therefore stores the proposal probabilities used at draft time, while the two argmax policies intentionally approximate this condition to reduce serving overhead. Empirically, we do not observe significant quality degradation, but we separate these policies to make the exactness–efficiency trade-off explicit.

Exact-Truncated. Draft tokens are sampled from the same truncated distribution q_i (top-k and top-p applied to $\tilde{z}_i/T$) that verification will consult. The compact support $(\text{top-}k \text{ ids}, \text{probs})$ of q_i is stored alongside each d_i until the next round. The corresponding modified rejection-sampling rule is given in Eq. (5); the stored q_i is the proposal distribution used when d_i was sampled, which is the closest match to the classical speculative-decoding condition.

Softmax-Argmax. Draft tokens are the argmax of $\tilde{z}_i$, so q_i is a point mass at $\arg\max \tilde{z}_i$. The ratio $p_i(d_i)/q_i(d_i)$ degenerates and the accept rule reduces to a target-only check, with p_i^{full} the full-softmax target over z_i/T:

$$\text{accept } d_i \iff u \le \min\Big(1,\, p_i^{\text{full}}(d_i)\Big), \quad y^\star \sim p_i^{\text{full}} \setminus \{d_i\} \text{ (via Gumbel-max on } z_i/T). \tag{21}$$

No draft distribution is stored.

Truncated-Argmax. Draft tokens are the argmax of $\tilde{z}_i$, but the accept rule uses the truncated p_i^{trunc} and q_i^{trunc} obtained by applying (k, p) truncation to both z_i and $\tilde{z}_i$:

$$\text{accept } d_i \iff u \le \min\Big(1,\, p_i^{\text{trunc}}(d_i)/q_i^{\text{trunc}}(d_i)\Big), \quad y^\star \sim \text{normalize}\big(\max(p_i^{\text{trunc}} - q_i^{\text{trunc}},\, 0)\big). \tag{22}$$

Because d_i is obtained by argmax rather than by sampling from q_i^{trunc}, the induced distribution over accepted tokens does not match the target distribution exactly. This policy does not require storing q_i across rounds, and the verify kernel operates on a k-wide support.

Fused full-softmax verify kernel. For the Softmax-Argmax policy, verification requires $p_i^{\text{full}}(d_i)$ on every verify row. A baseline implementation launches four or more kernels per row (softmax of z_i, gather of $p_i(d_i)$, the accept check, and, on rejection, construction and sampling of the correction distribution), and it materializes an $\mathcal{O}(KV)$ probability tensor for the correction step at vocabularies of $V \approx 1.5 \times 10^5$. This tensor dominates the AR-Trust decode step's memory traffic: without fusion, verify becomes I/O-bound. We implement the verify step as a single Triton kernel that does not materialize this tensor. The kernel streams over z_i in tiles of $\mathcal{O}(10^3)$ tokens, maintaining a numerically-stable running $(\max, \text{sum-exp})$ pair in the style of FlashAttention (Dao, 2023), and computes $p_i^{\text{full}}(d_i)$ directly from the final pair and the logit at d_i. Accepted rows exit the kernel after the accept check without performing correction-side work. Rejected rows take a second streaming pass and select the correction token by Gumbel-max over $\log \max(p_i - q_i, 0) + \text{Gumbel}(0, 1)$, tracking a running argmax across tiles. Gumbel-max samples the correction token without normalizing or CDF-inverting the correction distribution, removing two kernel launches that a separate CDF-inverse sampler would require.

Fused sparse verify kernel. For the Exact-Truncated and Truncated-Argmax policies, p_i and q_i both have compact supports of size k (the top-k of each side's truncation). The sparse verify kernel operates on $[k, k]$ tiles: loading the k token-id / probability pairs for target and draft takes $4k$ floats plus $2k$ integer indices per row. The correction $\max(p_i - q_i, 0)$ is evaluated on the k token ids where p_i has mass via a $k \times k$ lookup that reads $q_i(v)$ for each $v \in \text{supp}(p_i)$, and inverse-CDF sampling over k entries selects the correction token. The per-row memory traffic is $\mathcal{O}(k)$ instead of $\mathcal{O}(V)$, which reduces the verify-kernel memory footprint by roughly three orders of magnitude at $k \sim 50$ and $V \sim 1.5 \times 10^5$; the kernel contains no runtime branches on the compile-time-known k.

Tiled top-k logits over the vocabulary projection. The draft policies require, at each masked position, the top-k ids and probabilities of the noisy-stream distribution. Let $\mathbf{H} \in \mathbb{R}^{M \times d_{\text{model}}}$ denote the final-layer hidden states at the M positions for which logits are needed, and $W_{\text{lm}} \in \mathbb{R}^{V \times d_{\text{model}}}$ the LM-head projection. A baseline implementation materializes the full $\mathbf{H}W_{\text{lm}}^\top$ logits tensor of shape $[M, V]$ and then applies a top-k. We tile the vocabulary-projection matrix multiplication: for vocabulary chunks of $\mathcal{O}(3 \times 10^4)$ rows, we compute the partial $\mathbf{H}W_{\text{lm}}^\top|_{\text{chunk}}$ matrix, take its top-k

per row, and merge the per-chunk top-k sets into a running top-k across chunks by concatenation and re-selection. Peak auxiliary memory is bounded by the size of the per-chunk partial matrix multiplication plus the running top-k, and the full $[M, V]$ logits tensor is never instantiated. This is a chunked matrix multiplication with top-k merging rather than a fully fused Triton LM-head kernel; it avoids the dominant memory term in diffusion decoding and produces the compact logits that the sparse verify kernel consumes.

E.3 Diffusion-Trust Sampling

Algorithm 9 gives the method-level denoising interface. Here we describe the serving controls that make the same interface compatible with the hybrid state cache. Figure 17(b) shows the attention pattern used during each denoising forward.

Per-block denoise loop. A block begins with a clean seed at its first position followed by $B-1$ masked positions. The seed is sampled from the final logit row of the previous block (or from the prompt prefill for block 0). Each denoise iteration forms the unresolved set $\mathcal{U}^{(s)}$, runs one bidirectional forward over the block, samples a candidate token at every unresolved position from the shifted noisy-stream logits under (T, k, p), and commits the selected set $\mathcal{C}^{(s)}$ according to the threshold schedule γ_s. When all positions are set or the step budget is reached, a token-causal forward writes $\mathbf{S}$ back to the state pool, appends the block to $\mathcal{K}$, and produces the final logit row from which the next block's seed is sampled. Algorithm 12 expands the compact interface of Algorithm 9 with seed sampling, state-pool updates, and per-forward control flags.

Differences between the denoise pass and the state-update pass. Each denoise iteration operates on speculative (masked) rows whose committed values are not yet known, so writing the resulting recurrent state back to the live pool would contaminate $\mathbf{S}$ with intermediate speculative updates. We therefore process every block in two forwards with different control flow. (i) The denoise forward uses the bidirectional block mask of Figure 17(b); the state-update forward uses a token-causal mask over the block. (ii) The denoise forward reads $\mathbf{S}$ from the recurrent pool but does not write it back; the state-update forward writes the final per-layer state back to the pool. (iii) The denoise forward returns the full vocabulary distribution at every block position because multiple positions are sampled per iteration, while the state-update forward returns only the final logit row for seed sampling. Our implementation exposes these three axes as flags on the forward-batch descriptor that the hybrid-attention kernels read to select mask type, state-update rule, and logits granularity.

Algorithm 12 Diffusion-Trust serving loop (implementation details).

Require: prompt $x_{1:n}$, max generation length L, block size B, step budget S_{step}, thresholds $\gamma_1, \ldots, \gamma_{S_{\text{step}}}$, sampling parameters (T, k, p)

$y_{n+1:n+L} \leftarrow$ `MASK`; prefill $x_{1:n}$ under a causal mask, write $\mathbf{S}$ back to the state pool, and sample seed s_0 from the final prompt logit
for $b = 0, 1, \ldots, \lceil L/B \rceil - 1$ **do**
  $\mathcal{B}_b \leftarrow \{\, n + bB + 1, \ldots, n + (b{+}1)B \,\}$
  set $y_{n+bB+1} \leftarrow s_b$
  for $s = 1, \ldots, S_{\text{step}}$ **do**
    $\mathcal{U}^{(s)} \leftarrow \{i \in \mathcal{B}_b : y_i = \texttt{MASK}\}$
    if $\mathcal{U}^{(s)} = \varnothing$ **then break**
    end if
    run a bidirectional forward over $\mathcal{B}_b$ without writing $\mathbf{S}$ back to the state pool
    for each position $i \in \mathcal{U}^{(s)}$ **do**
      sample $\hat{y}_i$ from the shifted noisy-stream logit under (T, k, p); record $c_i = p_\theta(\hat{y}_i \mid \cdot)$
    end for
    $\mathcal{C}^{(s)} \leftarrow \{i \in \mathcal{U}^{(s)} : c_i \geq \gamma_s\}$ ▷ $\gamma_{S_{\text{step}}}$ commits all remaining positions
    set $y_i \leftarrow \hat{y}_i$ for all $i \in \mathcal{C}^{(s)}$; $\mathcal{U}^{(s+1)} \leftarrow \mathcal{U}^{(s)} \setminus \mathcal{C}^{(s)}$
  end for
  run a token-causal forward over $\mathcal{B}_b$; write $\mathbf{S}$ back to the state pool; append the keys and values of $\mathcal{B}_b$ to $\mathcal{K}$
  sample seed s_{b+1} from the final logit row
  if `EOS` $\in y_{\mathcal{B}_b}$ **then break**
  end if
end for
return y up to the first `EOS` (or the length budget)

E.4 Serving-system machinery for the hybrid backbone

The serving stack is built around four mechanisms beyond a standard speculative-decoding engine. (1) *Recurrent-state commit*: accepting only r of K speculative tokens is not just a KV tail trim, since the GDN state after the verify round must also be rewound to $\mathbf{S}^{(r)}$. (2) *Native dLLM mask modes*: dense custom masks impose unnecessary per-score overhead, so **FLARE** encodes the clean/noisy row types compactly and uses a prefix-tile fast path whenever a KV tile lies entirely in the causal prefix. (3) *Fused verification and top-k kernels*: AR-Trust verification avoids materializing full $[M, V]$ logits or probability tensors, which is important at large vocabularies. (4) *CUDA-graph replay safety*: graph reuse must match not only tensor shapes but also block size, mask mode, recurrent-state update mode, and logits-output mode; otherwise a graph captured for one decoding path can be incorrectly replayed for another.

Recovering the recurrent state at the accept boundary. On a pure-softmax Transformer backbone, accepting r of K speculative tokens is a KV tail-trim: retain the first r new entries of $\mathcal{K}$ and release the rest. The Gated DeltaNet recurrent state $\mathbf{S}$ has no such per-position structure: the verify forward advances $\mathbf{S}$ through K rank-one updates, and only the state after the K-th update is retained in the live pool. Two mechanisms can recover the required post-accept state $\mathbf{S}^{(r)}$ (Figure 18). Path (a) – *replay* – re-executes the first r accepted updates through every GDN layer after the verify decision, at a cost of $L_{\text{gdn}} \cdot r$ additional recurrent updates (matrix multiplications). Path (b) – *cache-and-scatter* – records $\mathbf{S}^{(t)}$ for every verify position $t \in \{0, \ldots, K{-}1\}$ during the verify forward itself, at a cost of $L_{\text{gdn}} \cdot K$ additional stores (memory writes, no floating-point work). Draft-position intermediates are not stored, since the accept offset $r \leq K$ can never land on a draft position. **We adopt path (b)**, because the store cost is dominated by the recurrent kernel's existing matrix multiplication and is substantially cheaper than path (a)'s extra matrix multiplication. Concretely, the cache-and-scatter writes are produced inside the same per-step loop that already holds $\mathbf{S}$ in registers: at the start of step $t < K$, the kernel writes the current state $\mathbf{S} = \mathbf{S}^{(t)}$ to a buffer of shape $[L_{\text{gdn}}, B_{\text{batch}}, K, d_{\text{state}}]$ and then performs the step-t update, with no additional kernel launch. Once the verify decision is known, a single fused Triton kernel reads the accepted offset r per request, gathers the corresponding state slice from the buffer, and writes it to $\mathbf{S}[\mathcal{K}]$ for every GDN layer. The kernel is launched with one program per $(\text{request},\ \text{layer},\ \text{state-tile})$ triple, so the $B_{\text{batch}} \cdot L_{\text{gdn}}$ copies execute in one launch with per-tile parallelism over the state dimension, and requests with $r = 0$ early-exit without memory

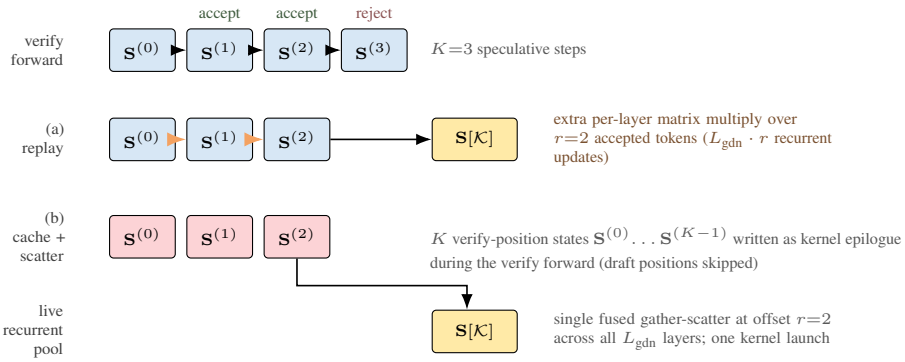


Figure 18: Two candidate mechanisms for producing the recurrent state $\mathbf{S}[\mathcal{K}]$ after a verify round that accepts $r=2$ of $K=3$ speculative steps; we adopt path **(b)**. **(a) Replay.** Re-execute the recurrence on the r accepted tokens after the verify result is known, producing $\mathbf{S}^{(r)}$ at the cost of an extra per-layer matrix-multiply pass. **(b) Cache-and-scatter (ours).** The verify forward writes the K verify-position intermediate states into an $[L_{\text{gdn}}, B_{\text{batch}}, K, d_{\text{state}}]$ buffer as the recurrent kernel's epilogue (one store per verify step, no extra launch), and a single fused Triton kernel selects the slice at offset r and writes it to the live pool for every GDN layer in one launch. Draft-position states are skipped because the accept offset r can never land on a draft position.

traffic. A baseline PyTorch implementation of the same scatter step requires five operations (validity mask, nonzero on valid request indices, two advanced-index selects, and a per-layer scatter), each a separate launch serialized over the layer dimension; our single-launch design removes this overhead.

Native dLLM mask modes and the prefix-tile fast path. The mask required by both sampling paths is causal over the prefix, causal for the clean rows of the active block, and bidirectional for the masked rows. FlashInfer's generic custom mask interface expresses this mask as a dense boolean tensor of size $\mathcal{O}(B(B+n))$ per request. Our implementation replaces this encoding and the per-score predicate it triggers with three mechanisms in FlashInfer's prefill kernel. First, two new mask modes accept compact metadata $(B, \text{row_type}_{1..B})$ in place of the dense boolean tensor: one mode assumes a uniform block layout across the batch, the other admits per-request row-type variation. Second, the kernel exploits the following *prefix-tile visibility invariant*: for any KV tile whose indices lie entirely in the prefix, every score is unmasked regardless of query row type, because a causal clean row sees every prefix token and a bidirectional masked row also sees every prefix token. Third, when the kernel detects that a KV tile ends before the prefix boundary (a single integer comparison per tile), it skips the per-score mask predicate and takes the causal fast path used by non-dLLM decode. The compact encoding plus the prefix-tile fast path reduce the dLLM mask's per-score overhead to zero on prefix-only tiles, which constitute the majority of attention compute at long context.

Graph-replay eligibility under diffusion-LLM invariants. Fixed-shape decode blocks are captured into CUDA graphs to amortize launch overhead. A standard shape-based eligibility check is, however, either too strict for the two sampling paths (disabling graph capture whenever any dLLM flag differs from the captured state) or too lax (silently replaying the wrong graph under a mask-mode or logits-mode mismatch); the latter case corrupts AR-Trust verification. Our replay-eligibility predicate matches four invariants at replay time: (i) block size, (ii) mask-mode metadata, (iii) recurrent-state update mode, and (iv) logits-output mode. The draft policies of §E.2 differ on invariant (iv): Softmax-Argmax consumes dense logits whereas Exact-Truncated and Truncated-Argmax consume top-k logits. We therefore capture one graph per draft policy and dispatch through the four-invariant predicate. This preserves graph-capture speedups on the steady-state decode loop and eliminates the silent correctness failure mode.

F Transfer-data mixes: sources and curation pipeline

This appendix documents every public dataset used to build the four transfer-data mixes of §3 (`Long-CoT`, `Short-CoT+Math`, `Long-CoT+Math`, `Long-CoT+Math+IF`), the common curation

pipeline we apply to those sources, and the per-mix filtering and weighting decisions. §F.1 catalogs the source datasets with their size, sample format, prompt origin, and generator. §F.2 describes the shared preprocessing steps (message-schema unification, reasoning-trace handling, token estimation, filtering). §F.3 walks through each of the four paper-facing mixes and shows which sources it is built from and how its components are weighted at training time.

F.1 Source datasets

All sources are public SFT corpora with a shared `[{role, content}]` chat schema (some carry an auxiliary `reasoning_content` field that we merge into the assistant message, see §F.2). Each paragraph below gives a one-sentence summary of what the dataset contains, the sample count, the prompt source, and the generator that produced the assistant responses; the Hugging Face URL is in a footnote. Sample counts, storage sizes, and token statistics in this subsection are as reported on the dataset cards at the time of download.

Llama-Nemotron-Post-Training-Dataset (SFT splits)[1]. A post-training SFT mix of long-chain-of-thought traces across four splits, approximately 4.2M samples total (math, 2.22M; code, 953K; science, 708K; chat, 349K). Each sample carries a `reasoning` field set to `on` or `off`; `on` samples contain an assistant response with an explicit `<think>...</think>` trace baked into the content. Prompts are drawn from NVIDIA's curated post-training prompt pool spanning math competitions, coding benchmarks, science question-answering, and chat. Assistant responses are generated by the Llama-Nemotron synthesis pipeline (primarily DeepSeek-R1 and variants). Storage: 114 GB.

Nemotron-Post-Training-Dataset-v2[2]. A general-purpose post-training SFT corpus spanning four English categories (stem, chat, math, code) and five multilingual categories (Japanese, German, French, Spanish, Italian); only the English splits (1,397,187 samples) are used in this paper. Average CoT length is shorter than Llama-Nemotron's long-CoT traces (roughly 800–1,350 tokens per assistant message, depending on category). Assistant responses have `<think>` reasoning baked into `content` directly rather than carried in a separate field. Synthetic data is generated with Qwen3 and DeepSeek-R1-0528. Storage: 92 GB.

Nemotron-Instruction-Following-Chat-v1[3]. Instruction-following chat conversations (320K samples) with an explicit `capability_target` label that distinguishes instruction-following from structured-output samples. Average assistant message is around 3.9K tokens. Reasoning is stored in a separate `reasoning_content` field. Prompts come from NVIDIA's instruction-following prompt pool; responses are generated with DeepSeek-R1-family models. Storage: 6.4 GB.

Nemotron-SFT-Instruction-Following-Chat-v2[4]. The v2 refresh of the instruction-following chat set, with broader prompt coverage and 1.99M samples. Average assistant message is around 1.2K tokens (shorter than v1). Same `reasoning_content` schema as v1. Storage: 14.4 GB.

Nemotron-Science-v1[5]. Science / STEM reasoning conversations (226K samples, average assistant message around 2.5K tokens). Topics include physics, chemistry, biology, and STEM-style problem solving. Reasoning is stored in `reasoning_content`. Generators include DeepSeek-R1 and Qwen3-family models. Storage: 2.3 GB.

Nemotron-Math-Proofs-v1[6]. Math-proof conversations (925K samples, average assistant message around 4.3K tokens) emphasising symbolic derivations and proof-style chain-of-thought. Reasoning is stored in `reasoning_content`. Storage: 27 GB.

Nemotron-SFT-Competitive-Programming-v2[7]. Competitive programming problem/solution pairs. The schema varies across its four splits: an `exercism` split (79K samples) with no reasoning

[1] https://huggingface.co/datasets/nvidia/Llama-Nemotron-Post-Training-Dataset
[2] https://huggingface.co/datasets/nvidia/Nemotron-Post-Training-Dataset-v2
[3] https://huggingface.co/datasets/nvidia/Nemotron-Instruction-Following-Chat-v1
[4] https://huggingface.co/datasets/nvidia/Nemotron-SFT-Instruction-Following-Chat-v2
[5] https://huggingface.co/datasets/nvidia/Nemotron-Science-v1
[6] https://huggingface.co/datasets/nvidia/Nemotron-Math-Proofs-v1
[7] https://huggingface.co/datasets/nvidia/Nemotron-SFT-Competitive-Programming-v2

traces; a `text_to_sql` split (97K); and two competitive-coding splits in C++ (333K) and Python (337K), both with `reasoning_content` traces averaging 30K–56K tokens. Weighted total: 845K samples. Prompt sources span Exercism exercises, text-to-SQL benchmarks, and CodeForces-style competitive problems. Responses are generated by DeepSeek-R1 variants. Storage: 90 GB.

Nemotron-Cascade-1-SFT-Data (Stage-2 instruction-following only)[8]**.** NVIDIA's first-generation "cascade" SFT dataset; we only use its instruction-following subset (146K samples). Prompts are taken from the Tulu-3 SFT mixture[9]; assistant responses are generated by DeepSeek-R1-0528.

Nemotron-Cascade-2-SFT-Data (instruction-following subset)[10]**.** NVIDIA's second-generation cascade SFT dataset. We use its instruction-following split after applying NVIDIA's own constraint-satisfaction filter, which selects 362K samples from a raw pool of about 820K. Assistant responses are generated by the Cascade-2 pipeline (GPT-OSS-120B).

F.2 Curation pipeline

Every source goes through the same preprocessing pipeline before it is combined into any of the four transfer mixes. The pipeline has four stages, implemented in our data-preparation scripts and applied uniformly across sources.

(1) Message-schema unification. All samples are rewritten into a single `[{role, content}]` chat schema. Three conversion variants handle the three schema families we encounter in the sources above:

- Sources with a separate `reasoning_content` field (Chat-v1, Chat-v2, Science-v1, Math-Proofs-v1, Competitive-Programming-v2) have it merged into the assistant `content` wrapped in a `<think>{reasoning}</think>` block.
- Llama-Nemotron's `input/output` schema is flattened into a messages list; the `<think>` trace is already baked into the output string, so no reasoning merge is needed.
- Post-Training-v2 already ships with `messages` and `<think>` tags inline; we only filter to the English categories and pass the rest through unchanged.

Samples without any reasoning trace are padded with an empty `<think></think>` block so that downstream prompt-template handling is uniform across sources. The goal of this unification is that a plain string-concatenation of the resulting `[{role, content}]` messages under Qwen3's ChatML template yields the same token sequence that a full Qwen3/Qwen3.5 `apply_chat_template` call would produce on the upstream raw sample. This lets our data pipeline avoid instantiating a tokenizer: every source is rewritten once into the canonical messages form, and the trainer handles tokenisation.

(2) Category and source tagging. Each sample is assigned a `category` label (one of *math*, *code*, *stem*, *chat*) and a `source` tag identifying the upstream dataset. These labels are preserved through all subsequent stages and let us compute per-category statistics and apply category-specific filtering.

(3) Token-count estimation. Because the combined source pool has tens of millions of samples, tokenizing the full corpus with the Qwen3-1.7B tokenizer is prohibitively slow. We instead estimate per-sample token counts from character counts plus a per-source char-to-token ratio: we tokenize 3,000 randomly-sampled messages per source to compute that source's median character-to-token ratio, and divide the per-sample character count by this ratio (with a fixed 15-token template overhead). Empirical spot checks against full tokenisation put the estimation error under 5%. We record both a total-token and an assistant-token estimate for each sample.

(4) Filters applied uniformly. Two filters run on every mix regardless of its target length or domain. First, samples whose assistant message contains an `<think>` tag without a matching `</think>` (indicating a truncated reasoning trace in the upstream source) are dropped; this removes

[8] https://huggingface.co/datasets/nvidia/Nemotron-Cascade-1-SFT-Data
[9] https://huggingface.co/datasets/allenai/tulu-3-sft-mixture
[10] https://huggingface.co/datasets/nvidia/Nemotron-Cascade-2-SFT-Data

approximately 45K samples from the `Long-CoT` source pool and 6K from the `Math` source pool. Second, when multiple sources are concatenated, we deduplicate across the combined pool using a hash of the serialized message list; this removes approximately 11K duplicates when the `IF` sources are merged.

Output format. After preprocessing each source yields a curated table where every row carries: the serialized messages list, the category and source tags, the estimated total and assistant token counts, and the sample's origin metadata. The four transfer mixes of §F.3 are built by selecting, filtering, and concatenating rows from these curated source tables.

F.3 The four transfer-data mixes

Concretely, the four mixes are built from three reusable source pools, each drawn from publicly released SFT corpora: a long chain-of-thought pool (**Long-CoT**, primarily Llama-Nemotron-Post-Training (Bercovich et al., 2025)), a math pool (**Math**, the math splits of Llama-Nemotron-Post-Training plus Nemotron-Math-Proofs-v1), and an instruction-following pool (**IF**, the IF-tagged subsets of Nemotron-Cascade-1/-2-SFT-Data and Nemotron-Instruction-Following-Chat-v1/v2 (Basant et al., 2025)); a short-reasoning pool (**Short-CoT**, the English split of Nemotron-Post-Training-Dataset-v2 (Basant et al., 2025)) is used only by Mix 2. Specifically, (1) **Mix 1** (`Long-CoT`) uses the Long-CoT pool alone and serves as a reasoning-heavy baseline; (2) **Mix 2** (`Short-CoT+Math`) combines the Short-CoT pool with the Math pool to separate the effect of trace length from math coverage; (3) **Mix 3** (`Long-CoT+Math`) replaces Short-CoT with the Long-CoT pool to test whether stronger math coverage helps without shortening traces; and (4) **Mix 4** (`Long-CoT+Math+IF`) further adds the IF pool on top of Long-CoT and Math to test whether broader instruction coverage improves transfer beyond reasoning-centric domains. The four pools span roughly 0.5–17B assistant tokens, and components are combined under per-mix sampling weights that we arrived at empirically through the comparisons reported below; the resulting weights are visualized in Figure 3(a) and detailed in Table 7 and below, while Appendix F.5 additionally documents an automatic, instance-level data-selection pipeline that we explored as a complementary route.

The four mixes combine the curated sources of §F.1 under per-mix filtering and weighting rules. Each mix is trained at the same 4,096-token packed context with global batch size 256 for 9,000 optimiser steps on the Qwen3-1.7B seed (§3).

F.3.1 Long-CoT

`Long-CoT` targets the high-reasoning-depth regime by keeping only samples whose estimated total token count is at least 3,000. It draws from seven sources of §F.1: Llama-Nemotron-Post-Training (all four SFT splits), Post-Training-v2 (English splits only), Instruction-Following-Chat-v1, Instruction-Following-Chat-v2, Science-v1, Math-Proofs-v1, and Competitive-Programming-v2. The construction proceeds as follows:

1. Apply the common preprocessing pipeline (§F.2) to each source.
2. For efficiency, apply a cheap character-count prefilter at twice the target token threshold (since the character/token ratio is bounded below by approximately 2).
3. Apply the final filter using the calibrated total-token estimate, keeping samples with at least 3,000 tokens.
4. Concatenate the filtered per-source tables and shuffle with a fixed seed.

Final stats: 4,588,156 samples total, broken down as math 2.55M, code 1.23M, chat 635K, stem 179K. Storage: roughly 186 GB. At the 4,096-token packing cap used during training, this corresponds to approximately 18.3B total tokens and 16.8B assistant (supervised) tokens.

F.3.2 Short-CoT+Math

`Short-CoT+Math` pairs a short-reasoning general mix with a dedicated `Math` source pool, giving a contrast point to `Long-CoT`. The short-reasoning source is Post-Training-v2 (English splits) used without any length filter; average assistant traces in Post-Training-v2 are roughly 1K tokens, an order of magnitude shorter than Llama-Nemotron's long-CoT output. The `Math` source pool is

built from the math split of Llama-Nemotron (restricted to `reasoning=on`, roughly 2.22M samples) concatenated with all of Math-Proofs-v1 (roughly 920K samples after the unclosed-`<think>` filter drops 6K). No length filter is applied to either component. The two components are passed to the trainer with equal weights (see Table 7). The `Math` source pool alone contains 3,140,111 samples; at the 4,096-token packing cap it contributes approximately 11.4B total tokens.

F.3.3 Long-CoT+Math

`Long-CoT+Math` is a direct intersection of the previous two mixes: the `Long-CoT` source pool from §F.3.1 (all 4.6M length-filtered samples) and the `Math` source pool from §F.3.2 (Llama-Nemotron math plus Math-Proofs-v1, 3.1M samples total). No length filter is applied to the `Math` source pool in this mix; the two pools are passed to the trainer with equal weights. This isolates the effect of adding a math-only anchor on top of the `Long-CoT` source pool without changing the short-CoT baseline.

F.3.4 Long-CoT+Math+IF

`Long-CoT+Math+IF` adds a dedicated instruction-following `IF` source pool to the previous mix. The `IF` source pool is built from four sources and deduplicated across the merged pool:

1. Nemotron-Cascade-1-SFT-Data Stage-2 IF (146K samples), which uses Tulu-3 SFT prompts and DeepSeek-R1-0528 responses.
2. Nemotron-Cascade-2-SFT-Data IF (362K samples, selected by NVIDIA's constraint filter from a raw pool of roughly 820K); responses are generated by GPT-OSS-120B.
3. Instruction-Following-Chat-v1 samples tagged with the `instruction_following` capability target (78K), giving a direct IF-labeled subset of Chat-v1.
4. Instruction-Following-Chat-v1 structured-outputs samples (5K), which ask the model to generate JSON or XML conforming to a given schema.

After preprocessing and deduplicating across the four parts on a hash of the serialized message list (which removes about 11K duplicates), the `IF` source pool contains 591,718 samples, approximately 0.66B total tokens and 0.53B assistant tokens. It is trained alongside the `Long-CoT` and `Math` source pools with the weights of Table 7.

F.4 Mixing weights, sampling, and packing

Table 7 lists the trainer-level mixing weights used for the four mixes. The rest of this subsection describes how those weights are applied during data loading and how samples are packed into fixed-length training sequences.

Source-local packing. Each source dataset in the mix is wrapped in its own SFT data-loading iterator that produces *already-packed* 4,096-token training sequences. Raw samples from that source are concatenated into a running buffer via greedy bin-packing: for each raw sample we apply Qwen3's ChatML template to produce a pair $(\text{tokens}, \text{labels})$ with -100 on user and system positions (so only assistant tokens contribute to the loss); a sample whose tokenized length exceeds 4,096 is truncated, while a sample that would merely overflow the current buffer is deferred (the current buffer is flushed with padding up to 4,096, and the deferred sample starts the next packed sequence). Each packed sequence carries per-token document IDs; the trainer uses these with a document-causal attention mask to prevent cross-document attention within a packed sequence. For the AR training variant, the first label of each post-boundary document in a packed sequence is additionally set to -100, so the shifted-label AR loss never asks the last token of one document to predict the first token of the next.

Per-sequence multinomial mixing. Because packing is source-local, the unit of draw from the mixture is one fully-packed 4,096-token sequence. On each `next` call, a mixer component is selected from the multinomial distribution $(w_1, \ldots, w_k)$ of the current mix, and that component's next packed sequence is yielded. The multinomial draw is with replacement over components, while each component iterates its own packed sequences without replacement within its own epoch; the realized per-batch composition therefore matches $(w_1, \ldots, w_k)$ only in expectation. If a component is exhausted mid-epoch its weight is zeroed and the remaining weights are re-normalized. One consequence of this design matters for interpretation: raw samples from different sources never

co-occur in the same packed sequence, so no cross-source attention can happen even under the document-causal mask. The sampling RNG is seeded deterministically and persisted in the training checkpoint, so a resumed run produces the same sample stream as an uninterrupted run.

Table 7: Mixing weights for the four transfer mixes. Each row is one paper-facing mix; the Components column lists, for each component, the source pool and the sampling weight passed to the trainer. All four mixes are trained with a 4096-token context, global batch size 256, and 9000 optimiser steps on the Qwen3-1.7B seed.

Mix	Components (source pool → weight)
`Long-CoT`	`Long-CoT` pool (§F.3.1) → 1.0
`Short-CoT+Math`	Post-Training-v2 (English) → 0.5; `Math` pool (§F.3.2) → 0.5
`Long-CoT+Math`	`Long-CoT` pool → 0.5; `Math` pool → 0.5
`Long-CoT+Math+IF`	`Long-CoT` pool → 0.4; `Math` pool → 0.4; `IF` pool (§F.3.4) → 0.2

F.5 Automatic SFT data selection for AR-to-dLLM transfer

The four transfer mixes of §F.3 select data primarily by length and per-source weighting. An orthogonal question is whether automatic, instance-level quality selection on the same raw SFT pool improves AR-to-dLLM transfer quality. This subsection documents such a pipeline: it scores individual examples with an instruction-following-difficulty signal, applies per-source quality gates, and rebalances the result by domain via cluster-aware sampling, all without manual curation. The resulting mixture (which we call the `Auto-selected` mix below) is not used by any of the paper-facing **FLARE** runs, but its controlled comparison against an unfiltered baseline (under the same Qwen3-1.7B seed and training budget) informed our final decision to keep the `Long-CoT+Math+IF` mix lightly filtered.

Source pools. The automatic selection pipeline starts from two public SFT pools that together approximate the long-CoT plus instruction-following coverage of §F.3: `Dolci-Think-SFT-32B` (2.25M raw rows, 23.58B raw tokens; 1.17M rows and 3.02B tokens at $\leq$ 8K context) and `Nemotron-Cascade-2-SFT-Data` (24.49M rows, 162.94B tokens; 19.06M rows and 54.04B tokens at $\leq$ 8K context). Together the union has 26.74M rows and 186.52B tokens raw, dropping to 20.23M rows and approximately 57B tokens after the 8,192-token Qwen3 ChatML length filter. Cascade-2 overlaps in part with the Nemotron-Cascade-2-SFT-Data instruction-following subset already described in §F.1; here we use the un-filtered $\leq$ 8K pool rather than NVIDIA's constraint-satisfaction selection.

Six-step curation pipeline. The pipeline applies the following six steps in order. Each step only sees examples that survived the previous one.

1. **Structural validity.** Drop malformed conversations: empty `messages`, missing roles, zero-length content, unmatched `<think>` tags, assistant turns that repeat the prompt verbatim, and dangling final user turns. Known Cascade-2 tool-message leaks are normalized in place.
2. **Length filter.** Tokenize with the Qwen3 ChatML template and drop examples with more than 8,192 tokens. Long reasoning traces are dropped rather than truncated.
3. **Deduplication and evaluation decontamination.** Remove near-duplicates with MinHash LSH over 5-gram prompt shingles at Jaccard ≥ 0.8 for intra-source dedup, and drop rows with Jaccard ≥ 0.5 against the prompts of GSM8K, MATH-500, IFEval, HumanEval, MBPP, ARC-C, and GPQA. This step follows standard large-corpus deduplication and decontamination practice.
4. **Quality scoring.** Score each instruction-response pair (x, y) with the Instruction-Following Difficulty (IFD) score, defined as $\mathrm{IFD}(x, y) = \mathrm{PPL}(y \mid x)/\mathrm{PPL}(y)$, where $\mathrm{PPL}(y \mid x)$ is the response perplexity under the scorer when conditioned on the prompt and $\mathrm{PPL}(y)$ is the unconditional response perplexity. Lower IFD indicates that conditioning on the instruction substantially reduces response perplexity, which we interpret as stronger instruction–response alignment. We use Qwen3-1.7B-Base as a cheap weak scorer. We additionally record rank-normalized IFD, response PPL, and response length, drop extreme response-PPL outliers, and do not score very short responses.

5. **Composite ranking and source-specific gating.** Rank rows within each source by a composite of normalized IFD, length, and a diversity weight: $0.5 \cdot (1 - \text{IFD}_{\text{norm}}) + 0.3 \cdot \text{length}_z + 0.2 \cdot \text{diversity}_{\text{wt}}$. Apply source-specific keep rates (top 70% for cleaner math/science/IF sources, top 50% for Dolci Python algorithms, and top 40–60% for noisier chat/tool sources).
6. **Cluster-balanced sampling.** Embed instruction text with `BAAI/bge-small-en-v1.5`, run k-means within each domain bucket, and round-robin sample from clusters by composite score until each bucket budget is met. Each curated row carries a `cluster_id` and an inverse cluster-frequency weight.

Target and realized domain mixture. The bucket budgets were chosen to be evaluation-driven rather than proportional to raw source size, biased toward math, code, science, and instruction following. Table 8 lists the planned target shares and the realized bucket counts after sampling. The final mixture contains $1{,}627{,}706$ rows and approximately 5B tokens, against an original design target of roughly 6B curated tokens for three epochs (~ 18B effective training tokens).

Table 8: Auto-selected SFT mixture: target domain shares (eval-driven design) and realized bucket counts after the six-step pipeline. Targets are the budget passed to the cluster-balanced sampler; realized counts are the number of unique prompts in the curated split.

Bucket	Target share	Prompts	Realized share
`math_direct`	22%	252,060	15.49%
`math_tool`	8%	85,120	5.23%
`code`	18%	100,328	6.16%
`precise_if`	12%	476,567	29.28%
`science`	12%	335,153	20.59%
`chat`	15%	167,156	10.27%
`tool_multi_turn`	8%	85,308	5.24%
`safety`	3%	66,368	4.08%
`multilingual`	2%	59,646	3.66%
Total	100%	1,627,706	100%

The deviation between target and realized share, especially the overshoot on `precise_if` and `science` and the undershoot on `math_direct` and `code`, reflects how many high-IFD examples each upstream source actually contributed once the cluster-balanced sampler attempted to fill the bucket budgets without collapsing diversity.

Controlled comparison with an unfiltered baseline. To assess whether the additional curation effort translated into downstream quality, we ran a head-to-head SFT comparison on Qwen3-1.7B at the same training budget against an unfiltered baseline mixture used elsewhere in the project (referred to here as the "baseline" mix). Both mixes were trained on the same seed checkpoint with the same optimizer settings and evaluated under the same 16K-token evaluation harness. Table 9 reports the per-benchmark scores together with the parameter-matched Qwen3-1.7B reference evaluated at 8K context.

Table 9: SFT comparison between the unfiltered baseline mixture and the auto-selected mixture under the same Qwen3-1.7B seed, training budget, and 16K evaluation harness. The first row is the original Qwen3-1.7B checkpoint evaluated at 8K context as a reference. Bold marks the better of the two SFT runs on each benchmark.

Run	GSM8K	MATH-500	IFEval	HumanEval	MBPP	ARC-C
Qwen3-1.7B reference (8K eval)	88.93	85.80	71.72	81.71	75.49	89.16
Baseline SFT, 16K eval	82.79	**84.40**	**67.10**	66.46	58.37	82.51
Auto-selected SFT, 16K eval	**85.14**	79.80	63.59	**70.12**	**66.54**	**85.58**
Δ Auto-selected $-$ baseline	$+2.35$	-4.60	-3.51	$+3.66$	$+8.17$	$+3.07$

Under this evaluation setting, the auto-selected mixture improves GSM8K, HumanEval, MBPP, and ARC-C relative to the unfiltered baseline mixture, while regressing on MATH-500 and IFEval. The pattern suggests that the IFD-driven composite ranking and cluster-balanced sampler effectively

concentrate budget on cleaner code and science instruction-following data, but that the realized bucket undershoot on `math_direct` (Table 8) translates into a measurable MATH-500 regression, and the heavy reweighting away from generic chat hurts IFEval despite the high `precise_if` share.

We did not pursue this automatic selection pipeline further for the paper-facing **FLARE** runs. Both SFT runs in Table 9 fall well short of the reference Qwen3-1.7B at 8K, indicating that under our fixed training budget the bottleneck is not a shortage of high-IFD examples but the same source-distribution shift discussed in §4: continuing SFT on external long-CoT data drawn from non-Qwen generators shifts the output distribution away from the post-trained seed. The **FLARE** `Long-CoT+Math+IF` mix in §F.3 therefore keeps filtering minimal and relies on length-based selection plus per-source mixture weighting rather than the heavier composite-ranking approach explored here.

G Additional Experimental Details and Results

In this section, we present training and inference details for **FLARE**, as well as additional experimental results for the reader's reference.

G.1 Inference and Training Hyperparameters

Training For all experiments in this paper, we use an AdamW optimizer with a learning rate of 10^{-5}, a 500-step warm-up, and no learning rate decay. We always use a global batch size of 256 and a maximum sequence length of 4096 during training for 9000 optimizer steps. For Qwen3-related experiments, we use 32 NVIDIA RTX A100 80GB for experiments. For Qwen3.5-related experiments, we use 32 NVIDIA RTX H100 80GB for experiments. Each experiment requires a wall-clock time of 12 to 36 hours.

Inference We use SGLang for all the benchmark inference. We use a max response length of 32768 tokens, sampling temperature 1, top-p 0.95, top-k 50 for all the model checkpoints.

G.2 Additional Experimental Results

G.2.1 Throughput of FLARE

Table 10: **Fixed-output throughput (tokens/s).** Measured on 1×A100 80GB (bf16, SGLang serving stack, `max_new_tokens` = 2048, `ignore_eos`=true, temperature 1.0, top-p 0.95, top-k 50). Columns give the concurrency level C. Results other than **FLARE** are run under each model's recommended decoding configuration; **FLARE** uses AR-Trust decoding.

Model	GSM8K			HumanEval			GPQA Diamond		
	C=1	C=4	C=8	C=1	C=4	C=8	C=1	C=4	C=8
SDAR-1.7B	137.0	376.6	438.3	135.6	364.7	427.0	130.4	383.1	461.5
SDAR-4B	102.8	327.3	426.9	105.1	326.1	398.3	95.7	313.6	441.7
SDAR-8B	83.0	267.5	388.9	83.8	276.8	372.1	72.2	229.6	379.3
SDAR-30B-A3B	43.9	124.6	216.4	39.8	122.3	214.1	43.2	120.6	210.2
LLaDA-2.0-mini	185.6	408.7	518.7	234.2	496.7	626.1	182.6	264.2	373.2
LLaDA-2.1-mini	524.5	1224.1	963.0	472.4	1471.4	1646.5	226.9	283.9	401.2
FLARE-2B	487.0	1314.7	2087.0	373.5	1113.3	1763.9	330.7	949.0	1440.8
FLARE-4B	292.1	868.7	1293.2	268.2	812.2	1178.9	237.4	649.3	990.4
FLARE-9B	217.4	696.1	1096.5	203.2	642.8	1007.1	170.8	499.0	786.5

Figure 19 reports the high-concurrency headline setting, where the serving mechanisms of Section 4.3 translate the parallel decoding advantage into real tokens-per-second. On a single A100-80GB at C=8, **FLARE**-2B reaches 2,087 tokens/s on GSM8K, giving a 2.2× gain over LLaDA-2.1-mini and a 4.8× gain over SDAR-1.7B. On GPQA-Diamond, it reaches 1,441 tokens/s, a 3.6× gain over LLaDA-2.1-mini. The gap is largest at high concurrency, the regime in which per-step overhead dominates and the fused kernels are most valuable for keeping the hybrid recurrent state out of the critical path. **FLARE**-4B and **FLARE**-9B are slower in absolute throughput, as expected from their larger backbones, but remain competitive with larger dLLM baselines while providing the capability gains reported above. Full throughput results across three benchmarks and $C \in \{1, 4, 8\}$ are reported in Table 10.



Figure 19: High-concurrency fixed-output throughput. Bars report tokens/s at C=8 on 1×A100-80GB with bf16 and the SGLang serving stack; `max_new_tokens`= 2048 and `ignore_eos`=true. **FLARE** uses AR-Trust decoding. The full 3×3 grid over benchmarks and $C \in \{1, 4, 8\}$ is in Table 10 in Appendix G.

Table 11 reports the full-family (2B/4B/9B) per-benchmark comparison of the released Qwen3.5 checkpoints, AR-SFT on the `Long-CoT+Math+IF` mix, and **FLARE** on the same mix, providing the source-model and AR-SFT references behind the main-text capability discussion in Section 4.

Table 11: **Full-family FLARE comparison on the `Long-CoT+Math+IF` transfer mix.** *: potentially under-reported due to answer extraction or length limits, consistent with the markers in Table 2.

	Knowledge & Instruction Following					*Math*				*Code*		
Model	ARC-C	MMLU	MMLU-Pro	GPQA-D	IFEval	GSM8K	MATH-500	AIME-24	AIME-25	HumanEval	MBPP	LCBv6
Qwen3.5-2B (released)	92.15	73.59	59.53	62.12	79.48	77.63*	72.20*	8.89*	12.22*	48.17	53.31	17.71
+ AR-SFT	88.74	68.63	56.67	47.98	67.65	83.93	85.80	27.78	22.22	67.07	71.60	21.14
FLARE-2B	85.07	67.60	53.57	37.37	68.95	84.46	84.40	31.11	26.67	64.02	68.09	15.43
Qwen3.5-4B (released)	96.33	85.22	77.88	80.30	90.02	89.16	95.40	63.33	48.89	87.80	82.49	50.86
+ AR-SFT	93.86	80.75	71.70	61.11	72.46	91.66	94.40	62.22	46.67	92.68	91.83	45.71
FLARE-4B	93.52	78.73	71.14	63.64	73.20	91.05	94.20	58.89	43.33	93.29	89.11	41.71
Qwen3.5-9B (released)	97.70	88.21	81.39	80.30	91.31	89.16	96.60	65.56	60.00	95.12	89.11	49.71
+ AR-SFT	96.25	86.07	77.66	71.21	76.52	94.01	95.00	66.67	61.11	96.34	93.77	46.29
FLARE-9B	96.33	84.80	77.39	71.21	71.35	93.33	95.20	63.33	54.44	92.07	91.05	49.71

G.2.2 Throughput on H100 at Longer Output Horizons

To test whether the throughput advantage depends on the benchmark, the concurrency level, or the generation horizon, we additionally evaluate **FLARE** and the LLaDA-2.0/2.1-mini baselines on one H100 per model (tensor parallelism 1) with the same SGLang serving framework, prompt sets, concurrency levels $C \in \{1, 4, 8\}$, and sampling constraints (temperature 1.0, top-p 0.95, top-k 50, `ignore_eos`=true), at fixed output lengths of 2,048 and 16,384 tokens. Each configuration uses one warm-up wave followed by 15 measured waves, and we report aggregate completion tokens per second; each method uses its optimized serving path and recommended decoding configuration. These measurements form a separate within-hardware comparison and are not mixed with the A100 results in Table 10. To isolate the decoding path, we also report *causal AR* decoding of the same **FLARE** checkpoints, which shares the checkpoint, hybrid backbone, hardware, serving framework, and workload with AR-Trust.

Table 12 and Figure 20 show that the throughput advantage is not specific to GSM8K or to short outputs. At C=8, **FLARE**-2B achieves the highest throughput among the compared models on all three benchmarks at both output lengths, reaching 3,533 and 4,671 tokens/s on GSM8K at 2,048 and 16,384 output tokens, respectively. At C=1, AR-Trust is faster than causal AR decoding of the same checkpoint for every model size, benchmark, and output length (by 1.1–2.9×); with a fixed output length, higher C=1 throughput also means lower end-to-end generation latency. For every **FLARE** model, benchmark, and concurrency level, throughput at 16,384 output tokens is no lower than at 2,048, and TPF increases with the longer horizon in all nine model–benchmark settings (Table 13), with **FLARE** advancing 2.8–4.0 tokens per verification forward.

Table 12: **Fixed-output throughput on H100 (tokens/s).** One H100 per model (TP= 1, SGLang serving stack), temperature 1.0, top-p 0.95, top-k 50, `ignore_eos`=true. Columns give the concurrency level C. **FLARE** rows use AR-Trust decoding; *causal AR* decodes the same **FLARE** checkpoint on the same serving stack.

Model	GSM8K			HumanEval			GPQA Diamond		
	C=1	C=4	C=8	C=1	C=4	C=8	C=1	C=4	C=8
Output length 2,048 *tokens*									
LLaDA-2.0-mini	211.4	464.6	636.4	235.5	555.9	688.5	198.7	404.5	563.3
LLaDA-2.1-mini	1053.2	1514.1	1390.2	2216.7	3321.9	2105.7	231.6	417.5	577.1
FLARE-2B (causal AR)	426.6	1567.6	2791.1	425.8	1563.8	2707.9	425.1	1360.1	2462.8
FLARE-2B (AR-Trust)	710.5	2130.0	3533.4	594.6	1729.2	3020.8	461.4	1443.4	2584.9
FLARE-4B (causal AR)	227.7	856.9	1582.0	227.4	854.2	1614.9	227.3	854.9	1605.4
FLARE-4B (AR-Trust)	468.5	1388.4	2465.2	417.9	1304.7	2282.7	347.1	1072.0	1877.9
FLARE-9B (causal AR)	143.2	548.8	1060.4	144.1	552.9	1057.1	144.0	550.2	1048.8
FLARE-9B (AR-Trust)	346.6	1173.3	1908.1	319.4	1083.1	1829.9	251.2	825.9	1440.4
Output length 16,384 *tokens*									
LLaDA-2.0-mini	241.9	521.6	655.8	249.3	569.3	703.0	203.9	420.4	564.6
LLaDA-2.1-mini	1816.7	3018.1	3373.0	2235.4	3695.6	2668.6	363.8	564.6	695.8
FLARE-2B (causal AR)	419.4	1511.3	2766.8	418.2	1504.9	2749.3	419.0	1507.5	2745.8
FLARE-2B (AR-Trust)	846.7	2687.1	4670.6	785.9	2536.1	4406.3	601.6	1724.8	2962.4
FLARE-4B (causal AR)	224.0	807.1	1443.1	225.7	806.9	1438.0	225.7	802.9	1435.5
FLARE-4B (AR-Trust)	565.7	1849.9	2964.4	512.8	1479.6	2439.3	458.2	1245.8	2004.1
FLARE-9B (causal AR)	142.2	530.2	983.5	142.7	530.6	981.3	142.6	531.8	980.1
FLARE-9B (AR-Trust)	416.4	1495.8	2501.1	413.1	1403.8	2396.2	333.8	992.8	1643.6

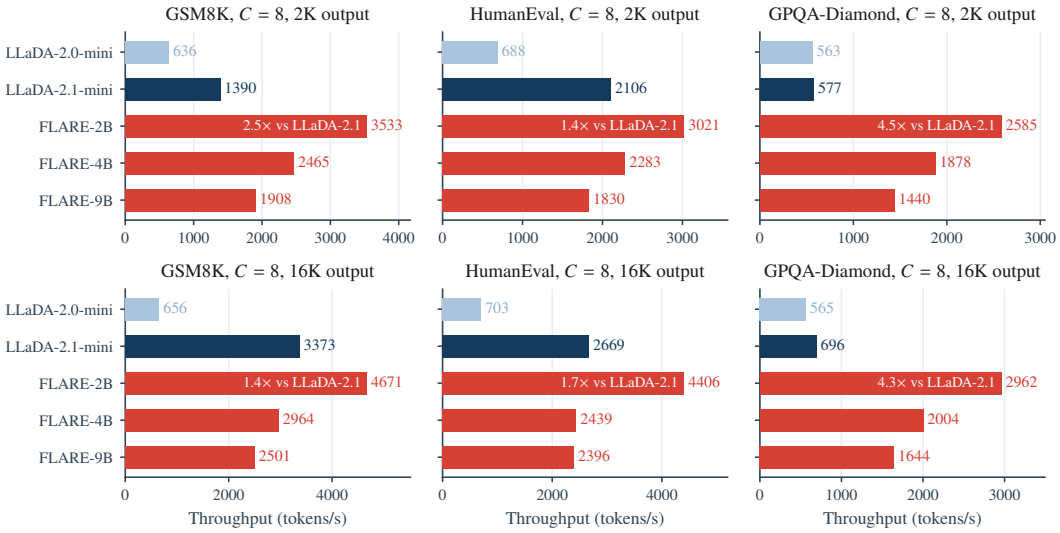


Figure 20: High-concurrency fixed-output throughput on H100. Bars report tokens/s at C=8 on one H100 per model (TP= 1, SGLang serving stack) at output lengths of 2,048 (top) and 16,384 (bottom); **FLARE** uses AR-Trust decoding. The full grid over $C \in \{1, 4, 8\}$, including causal AR decoding of the same checkpoints, is in Table 12.

Table 13: **Tokens per forward (TPF) of FLARE on H100.** Average number of tokens committed per verification forward (one plus the average number of accepted draft tokens) under AR-Trust at C=1; each cell reports output length 2,048 / 16,384.

Model	GSM8K	HumanEval	GPQA Diamond
FLARE-2B	3.666 / 3.951	3.305 / 3.822	2.769 / 3.291
FLARE-4B	3.616 / 3.921	3.402 / 3.748	2.980 / 3.533
FLARE-9B	3.665 / 3.936	3.481 / 3.926	2.965 / 3.535

G.2.3 Extended Analysis of the Transfer Study

This subsection gives the full discussion summarized in Section 3.

Data composition (Section 3.1). Figure 3(b)–(d) compares these four mixes across four transfer recipes under the same Qwen3-1.7B seed and training budget. AR fine-tuning (AR-SFT) denotes continued next-token fine-tuning on response tokens; all-mask causal (AM-Causal) and all-mask bidirectional (AM-Bidir) share the joint AR/diffusion supervision used by **FLARE** (formally introduced in Section 3.2) but replace random noisy masking with all-masked noisy views, differing only in whether the noisy stream is token-causal or bidirectional within each block (AM-Causal is close in spirit to concurrent I-DLM (Yu et al., 2026)). We read the resulting comparison along two axes. First, on the recipe axis, the all-mask variants tend to trail both AR fine-tuning and **FLARE**—most visibly on Code, where they drop sharply on the math-heavy Mix 2 and Mix 4—while AR fine-tuning and **FLARE** track each other closely within each mix. The first observation flags an objective-side gap that we dissect under a fixed data mix in Section 3.2; the second indicates that, with a properly aligned clean/noisy objective, AR fine-tuning is a faithful low-cost proxy for screening data mixes before running more expensive dLLM conversion. Second, on the data axis, no single data change is uniformly beneficial: Mix 2 slightly improves Math + Reasoning but substantially weakens Code, whereas Mix 4 gives the strongest Knowledge + Instruction Following results while keeping Math + Reasoning and Code competitive.

Two conclusions follow from the recipe and data axes of Figure 3. First, conditional on a properly aligned clean/noisy objective, the attainable transfer quality is governed primarily by the transfer data mix rather than by the algorithmic recipe; in particular, converted dLLMs track their AR fine-tuning counterparts under the same mix up to capability-group-specific deviations, so AR fine-tuning provides a low-cost proxy for screening data mixes prior to dLLM conversion. Second, comparisons of objective and attention-mask choices across different mixes are confounded by data quality and are therefore uninformative as algorithmic ablations. These observations motivate the controlled design used in the remainder of Section 3: Section 3.2 fixes Mix 1 (`Long-CoT`) and isolates the effects of objective design, clean/noisy visibility, logit shift, and noisy-mask sampling, while Section 4 adopts Mix 4 (`Long-CoT+Math+IF`) for the final hybrid-checkpoint conversions on the basis of its balanced behavior across capability groups in this sweep.

Objective and attention-mask ablations (Section 3.2). After isolating the data effect in Section 3.1, we fix the transfer data condition to Mix 1 (`Long-CoT`) and ask which objective and attention-mask ingredients prevent AR-to-dLLM conversion from losing the seed model's capability. All runs use the same document-level packing mask, which keeps variable-length prompt-response samples isolated inside packed batches while avoiding padding overhead. We therefore treat document-level masking as a fixed efficiency condition rather than an ablated factor, and focus on two conversion-specific choices: whether the clean stream retains AR-style token-causal supervision, and whether the noisy stream learns block-diffusion denoising.

Figure 4(b) isolates the noisy-stream side. At fixed AR-aligned clean-stream supervision, all-masked noisy views and randomly masked diffusion views yield comparable category-level scores, so noisy-mask sampling is not a primary driver of benchmark recovery; its role is instead to determine decoding compatibility, since random masking trains the noisy stream as a genuine denoising path and preserves Diffusion-Trust decoding, while logit shift aligns the block-diffusion logits with inference-time token positions and avoids a wasted block-boundary prediction.

Two conclusions follow from the clean- and noisy-stream axes of Figure 4. First, capability-preserving AR-to-dLLM conversion is governed primarily by clean-stream alignment, with logit shift and noisy-mask sampling contributing only marginal additional benchmark accuracy. Second, these auxiliary ingredients are retained for decoding compatibility, since they preserve both AR-Trust and Diffusion-Trust sampling paths. Combined with Section 3.1, this isolates the residual transfer gap to the quality of the data mix.

G.2.4 Study Summary Results

G.2.5 Extended Discussion of Main Results

This subsection gives the full discussion of Tables 1 and 2 summarized in Section 4.

Table 14: Ablation on the algorithmic design space. Rows (b)–(e) add one ingredient at a time: (a) AR next-token finetune baseline; (b) block diffusion with block-causal clean stream and no clean-stream loss; (c) adds a token-causal clean stream attention mask on top of (b); (d) adds the clean-stream NTP loss; (e) adds logit-shift on the noisy stream (**FLARE** setting). All rows share the same Qwen3-1.7B seed and the `Long-CoT` data.

Setting	Math + Reasoning	Knowledge + IF	Code
(a) AR-SFT	51.93	57.29	54.81
(b) Block Diffusion	40.98	31.29	26.47
(c) + causal context	43.09	48.42	49.19
(d) + NTP loss	52.70	54.53	51.41
(e) + logit shift (**FLARE**)	51.00	56.91	51.93

Table 15: Training mask sampling distribution and logit-shift ablation on `Long-CoT`, built on top of (c)+(d) of Table 14. All-masked (AM) represents diffusion loss computed with deterministic pure mask sampling; Diffusion (D) represents diffusion loss computed with random mask sampling. Per-benchmark numbers are in Table 18.

Noisy loss	Shift	Math + Reasoning	Knowledge + IF	Code
All-masked (AM)	✗	52.83	55.67	51.59
All-masked (AM)	✓	52.05	56.44	52.45
Diffusion (D)	✗	52.70	54.53	51.41
Diffusion (D; full **FLARE**)	✓	51.00	56.91	51.93

At the 9B scale, **FLARE**-9B matches or exceeds LLaDA-2.1-flash on most shared benchmarks, despite using roughly $1/10$ of its total parameters: GPQA-Diamond (71.21 vs. 66.67), MMLU-Pro (77.39 vs. 75.31), MBPP (91.05 vs. 88.29), and LiveCodeBench v6 (49.71 vs. 44.05). Compared with the commercial Mercury-2 system, **FLARE**-9B is clearly stronger on math-reasoning benchmarks (MATH-500 by $+14.2$ points, AIME-24 by $+12.2$ points); we note in fairness a remaining gap on LiveCodeBench v6 (49.71 vs. 67.30), reflecting Mercury-2's additional training advantage on code generation.

More directly, relative to the source-model Qwen3.5-9B, **FLARE**-9B retains the majority of the original capability across most tasks: 98.6% on MATH-500 (95.20 vs. 96.60), 96.6% on AIME-24 (63.33 vs. 65.56), and 95.1% on MMLU-Pro (77.39 vs. 81.39), while *exceeding* the original AR checkpoint on MBPP (91.05 vs. 89.11). The conversion thus preserves high-level reasoning and coding ability rather than simply improving easier short-form benchmarks. Together, the 9B results support both aspects of question (i) at the start of this section: parameter efficiency relative to substantially larger dLLM baselines, and capability retention relative to the same-family AR checkpoint.

The same pattern extends to the smaller scales. **FLARE**-4B outperforms SDAR-30B-A3B on nearly all reported benchmarks despite using a much smaller dense backbone, with the largest margins on GPQA-Diamond (63.64 vs. 36.7), MATH-500 (94.20 vs. 77.8), and AIME-24 (58.89 vs. 16.7). At the 2B scale, **FLARE**-2B improves substantially over the parameter-matched SDAR-1.7B on math and code tasks, including AIME-24 (31.11 vs. 10.0) and MATH-500 (84.40 vs. 63.2). Relative to the original Qwen3.5 checkpoints at both scales, the converted models retain a large fraction of the inherited capability, further indicating that the method functions as capability-preserving transfer rather than training a new dLLM from scratch.

Each **FLARE** row is reported under both AR-Trust and Diffusion-Trust sampling, but the two columns correspond to a single trained checkpoint that only switches its sampling path. The two modes are close on math and on most knowledge benchmarks (with larger AR-Trust advantages on GPQA-Diamond and IFEval at 9B), while Diffusion-Trust is consistently weaker on code-generation tasks. We attribute this gap to two factors: diffusion-style decoding is more sensitive to brittle answer extraction (which fails under minor syntax variations) and long-output truncation under the current code-evaluation scripts, but the parallel-decoding interface also faces an inherent challenge on long, strictly structured outputs such as multi-function code, where each block commits without left-to-right syntactic verification. Results marked with * should therefore be interpreted conservatively. Even

Table 16: Ablation of data recipe. Methods: **AR-SFT**, continued finetuning in standard AR fashion; **AM-Causal**, all-mask noisy stream with pure causal attention both for clean stream and within noisy blocks; **AM-Bidir**, all-mask noisy stream with pure causal attention for clean stream and bidirectional attention within noisy blocks; **FLARE**. Per-benchmark numbers are in Table 19.

Data mix	Method	Math + Reasoning	Knowledge + IF	Code
(reference)	Qwen3-1.7B	64.70	66.74	65.60
`Long-CoT`	AR-SFT	51.93	57.29	54.81
	AM-Causal	50.81	57.57	53.86
	AM-Bidir	52.05	56.44	52.45
	FLARE	51.00	56.91	51.93
`Short-CoT+Math`	AR-SFT	48.64	57.25	50.69
	AM-Causal	54.71	56.35	33.99
	AM-Bidir	54.30	55.92	36.64
	FLARE	54.03	56.70	41.17
`Long-CoT+Math`	AR-SFT	52.55	55.84	53.62
	AM-Causal	53.23	54.73	51.16
	AM-Bidir	51.80	54.95	51.91
	FLARE	51.48	55.54	54.17
`Long-CoT+Math+IF`	AR-SFT	53.31	59.35	54.43
	AM-Causal	47.88	57.57	50.35
	AM-Bidir	49.09	59.14	52.06
	FLARE	53.09	59.19	52.66

with this caveat, the comparison supports the feasibility side of question (iii): a single **FLARE** checkpoint can support both a clean-stream verified causal sampling path and a block-diffusion sampling path, without training separate models for different serving regimes.

A residual gap to the original AR checkpoints remains, especially on instruction following and some coding benchmarks. We attribute this partly to the post-trained nature of the Qwen3.5 source models: continued SFT with external data drawn from a different distribution than the original post-training data can shift the output distribution away from the source checkpoint, consistent with the data-quality findings of Section 3. Future conversions may therefore benefit from transfer data more tightly matched to the source AR model's own reasoning and instruction-following style. With capability retention established, the next subsection turns to question (ii): whether this two-stream supervision can be executed on a hybrid backbone at acceptable training efficiency.

Capability retention. Table 17 expresses AR-SFT and **FLARE** scores as a percentage of the released Qwen3.5 source checkpoints, using the same-family comparison of Table 11. At 9B, **FLARE** retains more than 95% of the source quality on 9 of the 12 benchmarks and more than 90% on 10, matches the source on LiveCodeBench v6, and exceeds it on GSM8K and MBPP; the two main gaps are IFEval (78.1%) and GPQA-Diamond (88.7%). The AR-SFT control helps localize where these gaps arise: AR-SFT on the same data already lowers average retention to 97.6% at 9B and 95.3% at 4B, compared with 95.5% and 93.3% for **FLARE**, and on GPQA-Diamond at 9B the AR-SFT and **FLARE** scores coincide (71.21). The residual gap is therefore concentrated on a few benchmarks rather than systematic, and a large part of it is introduced by the SFT stage rather than by diffusion conversion; better aligning the transfer data with the source model's instruction-following distribution is the most direct path to further improvement. We omit the 2B scale because several Qwen3.5-2B source scores are potentially under-reported (marked with * in Table 2), which would inflate the ratios.

Cited and measured results. In Tables 1 and 2, the LLaDA-2.0/2.1 and SDAR rows are cited from their technical reports, Mercury-2 is evaluated through its official API, and the Qwen3.5 and **FLARE** rows are measured with our evaluation pipeline. We also re-evaluated the LLaDA and SDAR baselines with the same evaluation scripts as **FLARE** and consistently obtained lower scores than those reported in the original papers; to avoid underestimating the baselines, we report the original, higher numbers. The capability-retention analysis above rests on the same-family comparison of Table 11, which does not depend on cited numbers.

Table 17: **Capability retention relative to the released Qwen3.5 checkpoints (%).** Scores from Table 11 divided by those of the released source model; **FLARE** uses AR-Trust. Values above 100 indicate that the model exceeds its source.

	Knowledge & Instruction Following					*Math*				*Code*			
Model	ARC-C	MMLU	MMLU-Pro	GPQA-D	IFEval	GSM8K	MATH-500	AIME-24	AIME-25	HumanEval	MBPP	LCBv6	Avg.
Qwen3.5-4B + AR-SFT	97.4	94.8	92.1	76.1	80.5	102.8	99.0	98.2	95.5	105.6	111.3	89.9	95.3
FLARE-4B (AR-Trust)	97.1	92.4	91.3	79.3	81.3	102.1	98.7	93.0	88.6	106.3	108.0	82.0	93.3
Qwen3.5-9B + AR-SFT	98.5	97.6	95.4	88.7	83.8	105.4	98.3	101.7	101.8	101.3	105.2	93.1	97.6
FLARE-9B (AR-Trust)	98.6	96.1	95.1	88.7	78.1	104.7	98.6	96.6	90.7	96.8	102.2	100.0	95.5

G.2.6 Per-benchmark Detailed Results of Main Tables

In this section, we include the per-benchmark score of each model we trained that is included in the main study and depicted in Tables 15–16. Every non-AR checkpoint is decoded under the AR-Trust sampling path (clean-stream trust) through speculative decoding unless it is not applicable; AR baselines use native AR decoding. Benchmark columns group by capability (Knowledge & Instruction Following, Math, Code).

Table 18: Per-benchmark numbers behind Table 15

Loss	Shift	ARC-C	MMLU	MMLU-Pro	GPQA	IFEval	GSM8K	MATH500	AIME24	AIME25	HumanEval	MBPP	LCBv6
AM	✗	80.89	62.23	45.63	30.30	59.15	83.78	84.20	16.67	21.11	70.12	66.93	17.71
AM	✓	83.02	62.62	47.37	24.75	59.33	83.09	84.80	17.78	18.89	71.95	66.54	18.86
D	✗	82.59	62.63	45.88	30.81	55.08	83.24	81.80	20.00	17.78	67.68	67.70	18.86
D	✓	84.04	62.70	45.92	26.26	62.11	83.02	83.80	10.00	18.89	71.34	67.32	17.14

Table 19: Per-benchmark numbers behind Table 16: the four transfer data mixes × four methods, all decoded under AR-mode. Each row block is one data mix; within a block, the four rows are AR-SFT / AM-Causal / AM-Bidir / **FLARE** respectively.

Data mix	Method	ARC-C	MMLU	MMLU-Pro	GPQA	IFEval	GSM8K	MATH500	AIME24	AIME25	HumanEval	MBPP	LCBv6
`Long-CoT`	AR-SFT	84.90	63.77	48.77	28.79	59.33	84.31	85.80	14.44	13.33	76.83	71.60	16.00
	AM-Causal	80.89	62.83	47.39	27.78	62.48	83.40	85.00	13.33	14.44	75.00	68.87	17.71
	AM-Bidir	83.02	62.62	47.37	24.75	59.33	83.09	84.80	17.78	18.89	71.95	66.54	18.86
	FLARE	84.04	62.70	45.92	26.26	62.11	83.02	83.80	10.00	18.89	71.34	67.32	17.14
`Short-CoT +Math`	AR-SFT	81.40	63.82	51.56	31.31	56.38	83.93	85.20	4.44	5.56	71.34	70.43	10.29
	AM-Causal	84.13	64.85	50.03	30.30	54.16	84.69	83.60	26.67	18.89	42.07	49.03	10.86
	AM-Bidir	84.73	64.31	50.76	28.28	52.68	84.15	84.20	21.11	23.33	46.34	52.14	11.43
	FLARE	82.94	64.36	50.28	31.82	55.45	83.40	83.80	22.22	20.00	54.88	57.20	11.43
`Long-CoT +Math`	AR-SFT	82.59	62.46	47.19	31.82	57.86	83.40	86.40	16.67	14.44	71.95	71.21	17.71
	AM-Causal	81.48	61.07	45.82	29.80	57.30	83.78	83.20	18.89	22.22	71.34	66.15	16.00
	AM-Bidir	82.25	61.47	45.52	26.26	57.86	81.73	85.00	17.78	17.78	70.73	66.15	18.86
	FLARE	81.14	61.57	46.10	29.29	58.96	83.93	83.40	12.22	18.89	73.78	70.43	18.29
`Long-CoT +Math+IF`	AR-SFT	82.76	62.91	48.40	31.82	66.73	76.72	83.00	22.22	23.33	71.95	68.48	22.86
	AM-Causal	82.25	62.75	45.45	24.24	64.51	75.82	81.60	6.67	16.67	70.12	62.65	18.29
	AM-Bidir	83.02	63.22	47.29	21.21	66.91	79.00	82.40	13.33	15.56	71.34	66.54	18.29
	FLARE	83.19	64.21	49.58	30.30	63.77	83.55	84.80	18.89	17.78	69.51	68.48	20.00

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: All the claims are well aligned with the paper and supported with evidence.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We include a dedicated discussion of limitations in the final section.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

Justification: We don't have theoretical results.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: We include full experiments details in the appendix.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: We release the code and model checkpoints (GitHub and Hugging Face links on the first page). The transfer data are built from publicly available corpora, with sources and curation pipeline described in Appendix F.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: We include all the experiments details.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Many of our results come with standard deviations or are averaged over multiple runs.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).

- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We include all details of experiments in the appendix.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics https://neurips.cc/public/EthicsGuidelines?

   Answer: [Yes]

   Justification: We respect the code of ethics.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: We have a dedicated section discussing it.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.

- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper poses no such risk.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We clearly mentioned all the source of the data used.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.

- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: We release the code and the **FLARE**-2B/4B/9B checkpoints, documented in the paper and in the project page, GitHub repository, and Hugging Face collection linked on the first page.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: We do not involve crowdsourcing with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [N/A]

Justification: The core method development does not involve LLM.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.