DSTREAMER: ACCEPTANCE-ALIGNED DIFFUSION STREAMING FOR HIGH-THROUGHPUT SERVING AND ONLINE RLVR

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-form generation is increasingly the dominant cost in high-throughput LLM serving and online reinforcement learning with verifiable rewards (RLVR). Speculative decoding increases token progress per target-model invocation by verifying multiple proposals in parallel, but at throughput-oriented batch sizes its effectiveness depends on keeping proposal and state overhead low as concurrency and context grow. Online RLVR adds a distinct challenge: as the target policy changes during training, a drafter calibrated once can become progressively stale.

We introduce DStreamer, a lossless speculative decoding framework with a single-pass diffusion-based in-model adapter that reuses the target model's cached KV state and requires no separate draft KV cache. DStreamer warm-starts the adapter with consistency distillation, then optimizes proposal paths with an acceptance-aligned reinforcement-learning objective under autoregressive verification, directly targeting the prefix-sensitive criterion that determines speculative utility.

We evaluate DStreamer in the throughput-oriented regimes where speculative decoding is most constrained by proposal and state overhead: continuous-batched serving, long-form generation, and synchronous online RLVR. Across Qwen and GPT-OSS models, DStreamer expands the serving frontier under concurrencies 64–1024, outperforming DFlash and EAGLE-3 throughout the evaluated range and reaching up to $3.8\times$ higher aggregate throughput than autoregressive decoding at matched per-request rate, while requiring no separate draft KV cache. DStreamer++ extends acceptance-oriented training to evolving policies while preserving exact on-policy rollouts, achieving 1.93–$2.15\times$ end-to-end RLVR speedup across Qwen3-4B, 8B, and 14B under synchronous $8\times$H100 training, while preserving final policy quality.

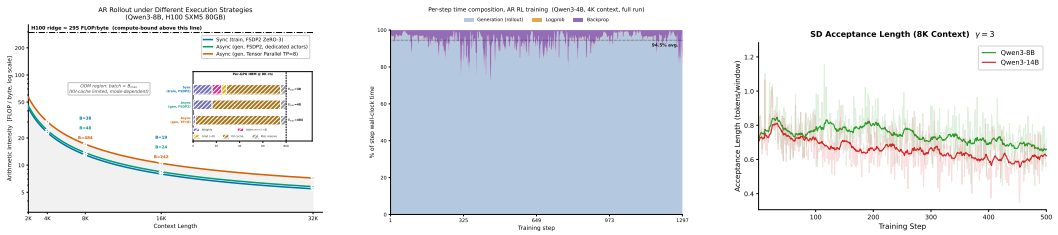
1 INTRODUCTION

Long-form generation is becoming a first-order cost in both high-throughput LLM serving and online reinforcement learning with verifiable rewards (RLVR). RLVR has emerged as a prominent post-training paradigm for reasoning (DeepSeek-AI, 2025; OpenAI, 2024; Kimi Team, 2025; Qwen Team, 2025), with strong results across challenging benchmarks (Yu et al., 2025; Shao et al., 2024; DeepSeek-AI, 2025). In common on-policy methods such as GRPO, each update requires sampling multiple completions from the current policy before rewards can be evaluated and the policy updated, placing generation on the critical path of training (Shao et al., 2024). As trajectories grow longer, this cost can dominate the iteration: in a representative synchronous Qwen3-4B GRPO run with 4K context, generation accounts for 94.5% of average step wall-clock time (Figure 1b). This makes generation throughput a natural optimization target for both serving and online RLVR.

A natural systems response is to increase concurrency through batching (Yu et al., 2022; Kwon et al., 2023). By amortizing shared model-weight reads across concurrent sequences, batching substantially increases arithmetic intensity in high-throughput decoding. Its benefits eventually diminish, however, because sequence-specific KV-cache traffic grows with both batch size and context length (Pope et al., 2023). Under a first-order HBM traffic model,

$$\mathrm{AI}_{\mathrm{AR}}(B, L) \approx \frac{B\, F_{\mathrm{tok}}(L)}{W + B\, C_{\mathrm{KV}}(L)},$$

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



(a) (b) (c)

Figure 1: **Generation bottlenecks in online RLVR.** (a) Batched autoregressive rollout remains bandwidth-bound at long context. (b) Generation accounts for 94.5% of step time in a representative Qwen3-4B RLVR run. (c) Acceptance declines for a frozen drafter as the target policy evolves.

where W is model-weight traffic and $F_{\text{tok}}(L)$ and $C_{\text{KV}}(L)$ are the per-sequence decode computation and KV-cache traffic at context length L. As B grows, the weight term is amortized while KV traffic continues to scale with the batch, so arithmetic intensity approaches $F_{\text{tok}}(L)/C_{\text{KV}}(L)$. For the Qwen3 model sizes, context lengths, and H100 configurations we evaluate, this ceiling remains below the hardware ridge point (Figure 1a), leaving batched autoregressive decoding on the bandwidth side of the roofline.

Once batching approaches this regime, further gains require more useful token progress per target-model invocation. Speculative decoding provides this by proposing multiple future tokens and verifying them together (Leviathan et al., 2023; Chen et al., 2023). In throughput-oriented settings, realized speedup can depend not only on acceptance, but also on proposal computation and state overhead (Yan et al., 2024; Bhendawade et al., 2025; Chen et al., 2026). Separate draft models introduce auxiliary model and KV-cache state, while autoregressive drafting adds sequential proposal steps (Chen et al., 2026; Jiang et al., 2026b), motivating lightweight parallel proposal mechanisms (Bhendawade et al., 2025; Chen et al., 2026). Speculative Streaming (Bhendawade et al., 2025) further integrates parallel speculation into the target model and directly reuses its KV state, avoiding a separate draft KV history. DStreamer adopts this architecture as its starting point and focuses instead on how its parallel proposals should be trained for autoregressive verification.

The joint-marginal gap. This single-pass parallel drafting design however introduces a learning mismatch (Bhendawade et al., 2025; Cai et al., 2024; Chen et al., 2026). A parallel drafter induces a factorized proposal distribution, $q_\phi(\mathbf{a} \mid x) = \prod_i q_{\phi,i}(a_i \mid x)$, whereas the target evaluates the candidate path autoregressively, $\pi_\theta(\mathbf{a} \mid x) = \prod_i \pi_\theta(a_i \mid x, \mathbf{a}_{<i})$. Token-wise training can therefore optimize position marginals without necessarily favoring proposal paths that remain compatible under autoregressive verification. This matters because verification is prefix-sensitive: an early mismatch makes the remaining proposals unusable (Leviathan et al., 2023; Chen et al., 2023). We refer to this mismatch between position-wise proposal training and path-level verification as the *joint-marginal gap* (Figure 2a). Modeling within-window dependencies autoregressively would capture the target's joint structure more directly, but would reintroduce sequential proposal steps. DStreamer instead preserves single-pass parallel drafting: we warm-start a one-step masked-diffusion adapter with consistency distillation (Sahoo et al., 2024; Lou et al., 2024; Amin et al., 2026), then treat it as a policy and optimize proposal paths with an acceptance-oriented RL objective evaluated under the target model's autoregressive conditionals. The proposal remains factorized, but training shifts probability toward paths better aligned with verification.

Speculative drift. In online RLVR, this alignment problem becomes dynamic: the target policy evolves during training, so a drafter calibrated to an earlier policy can become stale and lose acceptance (Chen et al., 2025). We call this degradation *speculative drift*, and observe it for frozen drafters in our experiments (Figure 1c). DStreamer++ carries acceptance-oriented adaptation into the RLVR loop: task rewards update the base policy, while acceptance rewards evaluated against the current policy update the adapter through disjoint parameters. Because the adapter changes only the proposal mechanism and lossless verification preserves the exact current-policy rollout distribution, DStreamer++ can track the evolving verifier without altering the distribution of RLVR trajectories.

Our main contributions are:

- **Acceptance-aligned parallel speculation.** DStreamer addresses the joint-marginal gap by optimizing complete proposal paths for autoregressive acceptance while preserving parallel drafting.
- **Online adaptation under speculative drift.** DStreamer++ continually realigns the drafter to the evolving RLVR policy while preserving the exact current-policy rollout distribution.
- **High-throughput validation.** DStreamer outperforms DFlash and EAGLE-3 under continuous batching; DStreamer++ sustains speculative gains throughout RLVR, reaching up to $2.15\times$ end-to-end speedup with matched final accuracy.

2 BACKGROUND

Lossless in-model speculation. To accelerate generation without changing the policy being sampled, we build on the lossless variant of Speculative Streaming (Bhendawade et al., 2025). Its speculative adapter q_ϕ attends directly to the target model's cached KV representations and requires no separate draft-model KV cache. At each speculative step, q_ϕ proposes a window of w future tokens in parallel, which the target model evaluates in a single verification forward pass. The adapter affects only which candidates are proposed; lossless verification ensures that the emitted sequence follows exactly the target sampling distribution (Appendix G).

Masked diffusion and consistency distillation. Masked discrete diffusion learns to reconstruct clean token sequences from progressively corrupted versions in which subsets of tokens are replaced by a mask symbol (Sahoo et al., 2024; Lou et al., 2024; Nie et al., 2025). Rather than running this denoising process iteratively at inference, consistency distillation trains predictions from more heavily corrupted states to agree with those from less corrupted states, allowing the learned denoising behavior to be compressed into a single prediction step (Amin et al., 2026; Ren et al., 2026). We use masked diffusion and consistency distillation only during warm-start training; at inference, the adapter still produces the entire speculative window in a single parallel pass.

On-policy RLVR. This exactness is particularly important in online RLVR, where the target policy itself evolves during training. We consider synchronous on-policy RLVR with GRPO (Shao et al., 2024; Yu et al., 2025), with π_θ denoting the current language-model policy. At each optimization step, prompts $x \sim \mathcal{P}$ are sampled and G completions per prompt are generated while θ remains fixed. The completions are scored with a verifiable task reward and then used to update θ. Because the rollout distribution changes across optimization steps as θ evolves, any acceleration mechanism must preserve sampling from the current policy at every step.

3 METHOD

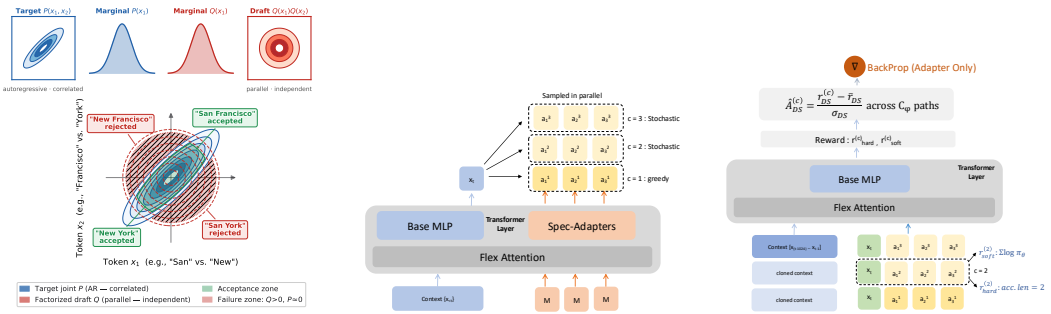
DStreamer trains the speculative adapter in two stages: diffusion-based initialization followed by acceptance-oriented reinforcement learning against a fixed target policy. The latter addresses a mismatch between position-wise proposal training and the path-level criterion imposed by speculative verification. DStreamer++ extends this objective to online RLVR, continuously adapting the drafter as the target policy evolves.

3.1 DSTREAMER: ACCEPTANCE-ALIGNED PARALLEL DRAFTING

Diffusion warm start. Speculative Streaming produces an entire speculative window in a single parallel pass from masked future positions (Bhendawade et al., 2025). We retain this inference architecture, but change how the adapter is trained: rather than learning the fully masked-to-clean mapping directly, we first train it as a masked-diffusion denoiser (Sahoo et al., 2024; Lou et al., 2024) and then use consistency distillation to collapse this behavior back to the single proposal pass required at inference /(Amin et al., 2026; Ren et al., 2026; Deschenaux & Gulcehre, 2025).

Given a target window $\mathbf{y} = (y_1, \ldots, y_w)$, we construct partially masked versions of $\mathbf{y}$ at different corruption levels and train the adapter to recover the original tokens. Let v_{early} and v_{late} denote two nested corruptions of the same target window, with v_{early} more heavily masked, and let $f_\phi(v)$ denote the adapter's token distributions for corrupted input v. We optimize

$$\mathcal{L}_{\text{init}} = \lambda_{\text{sc}}\,\mathrm{KL}\left(f_\phi(v_{\text{early}}) \,\|\, \mathrm{sg}[f_\phi(v_{\text{late}})]\right) + \lambda_{\text{ce}}\,\mathrm{CE}\left(f_\phi(v_{\text{early}}), \mathbf{y}\right), \tag{1}$$



(a) (b) (c)

Figure 2: **DStreamer acceptance-aligned training.** **(a)** Parallel drafting factorizes the proposal across future positions, whereas the target evaluates the same candidate autoregressively, creating the joint-marginal gap that motivates path-level training. **(b)** A single adapter forward pass produces the logits used to form one greedy proposal and $C_\phi - 1$ stochastic proposal paths. **(c)** The frozen target scores each path with the hard prefix-compatibility and soft target-likelihood rewards; group-normalized rewards define the GRPO update, with gradients applied only to the adapter.

following consistency distillation (Amin et al., 2026). The consistency term encourages the more corrupted state to match the prediction from the less corrupted state, while the cross-entropy term anchors the prediction to the clean target; sg denotes stop-gradient. After distillation, the adapter is queried only once per speculative window. This stage provides the initialization for DStreamer, but its supervision remains position-wise, motivating the path-level objective introduced next.

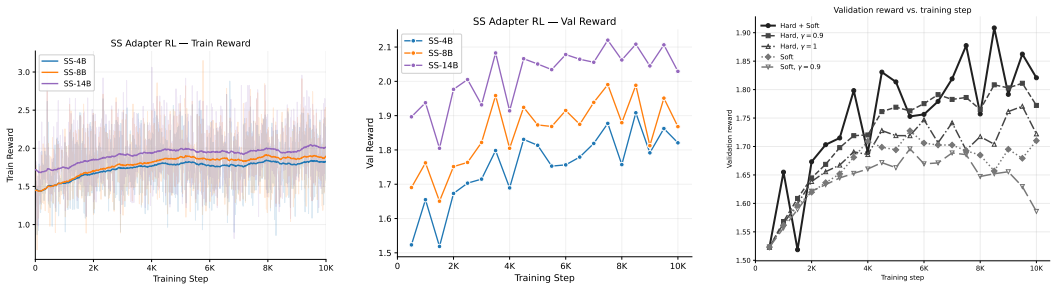
From token prediction to path acceptance. The limitation of position-wise training is structural. Let $\mathbf{a} = (a_1, \ldots, a_w)$ denote a speculative window proposed after context $x_{<t}$. Because the adapter predicts all w positions in parallel, its proposal distribution factorizes across positions conditioned on the same context (Chen et al., 2026; Bhendawade et al., 2025). The target model, however, evaluates the candidate as an autoregressive path, conditioning each position on the preceding proposals:

$$\underbrace{q_\phi(\mathbf{a} \mid x_{<t}) = \prod_{i=1}^{w} q_{\phi,i}(a_i \mid x_{<t})}_{\text{parallel, factorized}} \qquad \underbrace{\pi_\theta(\mathbf{a} \mid x_{<t}) = \prod_{i=1}^{w} \pi_\theta\left(a_i \mid x_{<t} \oplus \mathbf{a}_{<i}\right).}_{\text{autoregressive, prefix-conditioned}} \tag{2}$$

Matching the adapter's position-wise marginals therefore does not, in general, match the path distribution used for verification (Figure 2(a)). Individually accurate token predictions can still form a proposal path that becomes unlikely once earlier tokens are taken into account. This matters directly for speculative decoding: verification is prefix-sensitive, so an early rejection makes all subsequent proposals in the window unusable (Leviathan et al., 2023; Chen et al., 2023).

Acceptance-aligned adapter training. This mismatch motivates changing the unit of optimization from individual tokens to complete speculative paths. We treat q_ϕ as a policy and optimize its proposals for an acceptance-oriented reward under the fixed target policy π_θ. For each speculative window, we form a group of C_ϕ candidate paths from the same adapter logits: the greedy proposal $\mathbf{a}^{(1)}$ and $C_\phi - 1$ stochastic samples $\mathbf{a}^{(c)} \sim q_\phi(\cdot \mid x_{<t})$. Because all candidates are obtained from the same logits, this requires no additional adapter forward passes (Figure 2(b)). We score each path under the frozen target model, normalize rewards within the group, and update ϕ with a PPO-clipped GRPO objective defined over the likelihood ratio of the complete proposal path (Schulman et al., 2017; Shao et al., 2024), while keeping θ fixed. The full optimization procedure, including the path-level PPO-clipped objective, is given in Appendix B. The additional stochastic paths are used only for training; at inference, the adapter produces a single proposal followed by the lossless verification procedure of Section 2.

Acceptance-aligned reward. Speculative verification is prefix-sensitive: if the proposal at position j is rejected, the remaining positions $j+1, \ldots, w$ are discarded for that decoding step



(a) (b) (c)

Figure 3: **Acceptance-oriented RL improves beyond consistency distillation.** Starting from consistency-distilled adapters, DStreamer improves reward on **(a)** training contexts and **(b)** held-out validation contexts across Qwen3-4B, Qwen3-8B, and Qwen3-14B. **(c)** On Qwen3-4B, reward ablations from the same initialization show that combining hard and soft rewards gives the highest held-out accepted length under lossless verification. The target policy remains frozen throughout.

(Leviathan et al., 2023; Chen et al., 2023). We therefore score candidate paths using both prefix compatibility and target likelihood. For training, let $I_j(\mathbf{a})$ indicate whether a_j lies in the target model's top-K_{acc} set conditioned on the preceding candidate prefix $x_{<t} \oplus \mathbf{a}_{<j}$. We use $K_{\text{acc}} = 1$ in all experiments. We define

$$\underbrace{r_{\text{hard}}(\mathbf{a}) = \sum_{i=1}^{w} \gamma^{i-1} \prod_{j=1}^{i} I_j(\mathbf{a})}_{\text{prefix compatibility}} \qquad \underbrace{r_{\text{soft}}(\mathbf{a}) = \sum_{i=1}^{w} \log \pi_\theta \left(a_i \mid x_{<t} \oplus \mathbf{a}_{<i}\right).}_{\text{target likelihood}} \tag{3}$$

The hard term rewards a candidate prefix only while every preceding proposal remains top-K_{acc} compatible with the target. Thus, once compatibility fails at position j, no position at or after j contributes further hard reward. The factor γ^{i-1} places greater weight on earlier compatible positions, where failure would truncate more of the speculative window. The soft term provides a dense signal favoring paths that remain likely under the target's autoregressive conditionals. We combine the two as $r_{\text{DS}} = \lambda_{\text{hard}} r_{\text{hard}} + \lambda_{\text{soft}} r_{\text{soft}}$, with $\lambda_{\text{hard}} + \lambda_{\text{soft}} = 1$; we use equal weights in all experiments. For efficiency, reward scoring reuses the target KV cache and retains the most recent $L_{\text{local}} = 1024$ context tokens together with the candidate prefix. This context truncation and the top-K_{acc} compatibility criterion are used only for adapter training; inference continues to use the exact lossless verification rule G.

3.2 DSTREAMER++: ONLINE ADAPTATION

Speculative drift. DStreamer aligns q_ϕ to a fixed target policy π_θ, but in online RLVR the verifier itself moves. Each update to θ changes the autoregressive conditionals that determine whether a speculative path remains compatible with the target. A drafter trained against an earlier policy can therefore become stale even if its own parameters are unchanged (Chen et al., 2025; Zhang et al., 2025). Empirically, this appears as a decline in accepted prefix length for a frozen drafter; we refer to this degradation as *speculative drift* (Figure 10). In this setting, speculative efficiency is not a one-time alignment problem: the drafter must track the policy as it evolves.

Acceptance-aligned online adaptation. DStreamer++ addresses speculative drift by carrying the same path-level acceptance objective into the RLVR loop. As illustrated in Figure 8, each rollout batch is first generated losslessly under the current state (θ_t, ϕ_t) and then feeds two disjoint optimization branches. Task rewards update only the base policy θ, while retained speculative windows are scored under the pre-update policy θ_t and used to optimize ϕ with the acceptance-aligned objective of Equation 3. Thus, the policy learns from task reward while the drafter is optimized directly for speculative usefulness under the current verifier. Because ϕ changes only the proposal mechanism, lossless verification preserves exact sampling from $\pi_{\theta_t,T}$ throughout training (Leviathan et al., 2023; Chen et al., 2023).

4 EXPERIMENTS

4.1 EXPERIMENTAL PROTOCOL

Models and initialization. We evaluate DStreamer across two settings. For high-throughput serving, we use Qwen3-4B (Qwen Team, 2025) and GPT-OSS-20B (OpenAI, 2025); for online RLVR, we evaluate Qwen3-4B, Qwen3-8B, and Qwen3-14B (Qwen Team, 2025) . For the Qwen3 experiments, the speculative adapter is warm-started with consistency distillation and subsequently optimized for 10,000 steps with the base model frozen, using on-policy GRPO with the acceptance-oriented objective described in Section 3.1. This produces the DStreamer checkpoint used for fixed-target inference and the common initialization (θ_0, ϕ_0) for our online RLVR experiments.

Adapter architecture. DStreamer uses lightweight rank-256 adapter layers attached to selected decoder layers of the target model. Each adapter attends directly to the corresponding target KV cache, allowing speculative drafting to reuse the target's contextual state without allocating a separate draft-model KV cache (Bhendawade et al., 2025). For Qwen3, adapters are attached to strided decoder layers; for GPT-OSS-20B, they are attached to its full-attention layers (Qwen Team, 2025; OpenAI, 2025). The exact layer indices and adapter configuration are provided in Appendix C.3. We evaluate the same shared-state drafting design in two settings: fixed-target serving, where θ remains unchanged, and online RLVR, where θ evolves during training. We specify the systems configuration, baselines, and metrics for each setting below.

4.2 HIGH-THROUGHPUT SERVING

Serving setup. We evaluate fixed-target generation in a standard vLLM continuous-batching server (Kwon et al., 2023) on a single 80 GB H100 GPU. For Qwen3-4B, all methods use the same target model and serving configuration: BF16 execution, FlashAttention-3 (Shah et al., 2024), no weight or KV-cache quantization, CUDA graphs enabled, and tensor parallelism of one. Methods differ only in the speculative mechanism. We compare autoregressive decoding (AR), DStreamer and EAGLE-3 (Li et al., 2025) with three speculative tokens per step, and DFlash (Chen et al., 2026)with block sizes 4 and 16. We sweep $B = \texttt{max_num_seqs}$ from 64 to 1024 and continuously submit approximately $4B$ requests to keep the scheduler saturated. We report aggregate output-token throughput and per-request output rate. Speedup relative to AR is measured over the complete generation call, including prefill and decoding, while excluding engine initialization, CUDA-graph capture, warm-up, and tokenization.

Throughput under continuous batching. Figure 4 characterizes the serving frontier obtained by sweeping concurrency on two reasoning workloads, GSM8K (Cobbe et al., 2021) and MATH-500 (Lightman et al., 2023), and two coding workloads, HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021). Rather than comparing methods at a single concurrency, we report the trade-off between per-request output rate and aggregate server throughput, since increasing concurrency improves hardware utilization while reducing the rate observed by an individual request (Yu et al., 2022; Kwon et al., 2023). This frontier view removes dependence on any particular concurrency choice and directly measures whether a decoding method expands the set of achievable serving operating points. Across all four workloads, DStreamer traces the outer frontier over the evaluated range. At matched per-request output rate, it provides up to $3.8\times$ higher aggregate throughput than AR on GSM8K, with gains of approximately 2.4–$2.5\times$ on HumanEval and MBPP. Conversely, for a fixed aggregate-throughput target, DStreamer sustains a substantially higher per-request generation rate. The advantage therefore persists as the server moves from lightly loaded to throughput-oriented operation, rather than relying on a favorable fixed-batch setting .

Why speculative efficiency persists at high concurrency. To understand why this frontier remains separated as load increases, we decompose the speculative work itself in Appendix Figure 9. Across all four workloads, DStreamer maintains higher acceptance at later positions within its speculative window, whereas the benefit of extending the speculative horizon diminishes rapidly for DFlash as acceptance falls toward the tail of longer blocks. The rightmost panel shows the complementary systems effect: DStreamer's per-window drafting cost remains substantially lower than DFlash's and grows more slowly with concurrency. Together, these results separate proposal quality

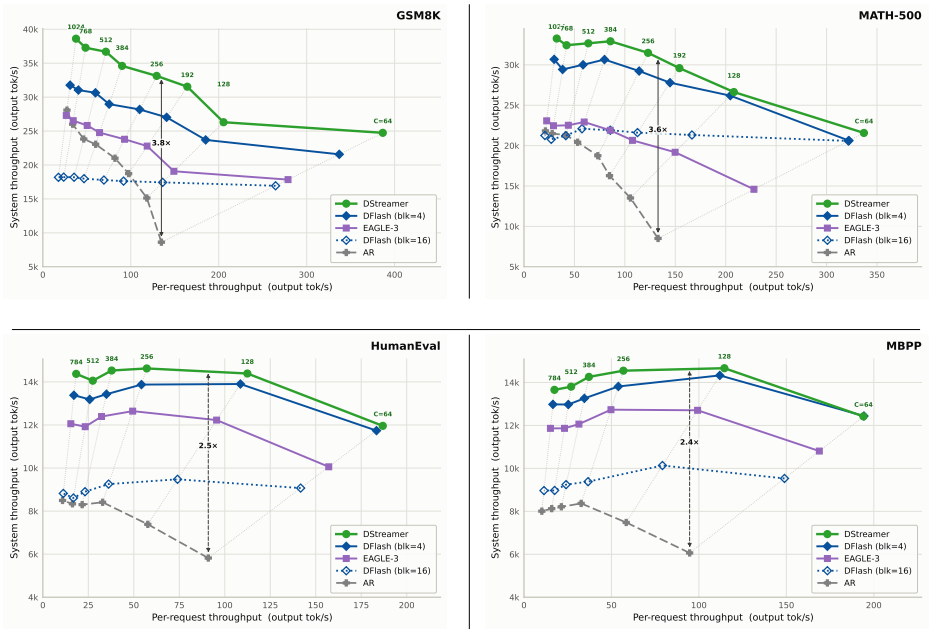


Figure 4: **Serving frontiers under vLLM continuous batching.** DStreamer traces the outer frontier across GSM8K, MATH-500, HumanEval, and MBPP. Arrows report aggregate-throughput gain over AR at matched per-request output rate.

(a) Generation-length scaling

		EAGLE-3		DFlash-4		DStreamer	
Dataset	$L_{\max}$	τ	spd.	τ	spd.	τ	spd.
AIME 2025	512	2.17	1.32	3.26	1.47	2.98	**1.94**
	2048	2.32	1.51	3.04	1.56	3.10	**2.33**
	4096	2.50	1.64	2.67	1.64	3.19	**2.71**
	8192	2.70	1.74		OOM	3.19	**2.65**
AIME 2026	512	2.16	1.25	3.18	1.30	2.87	**1.85**
	2048	2.33	1.50	3.17	1.34	3.15	**2.33**
	4096	2.48	1.63	2.81	1.53	3.19	**2.59**
	8192	2.71	1.87		OOM	3.17	**2.66**

(b) Additional draft KV required

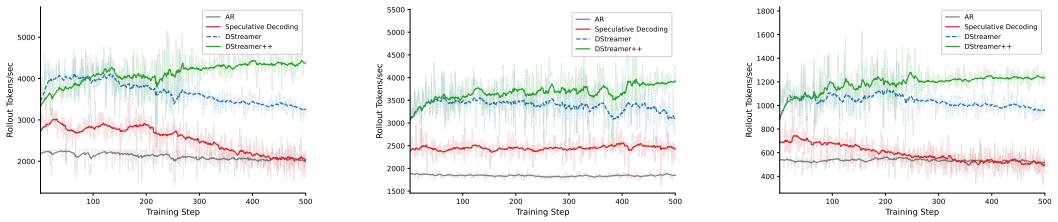
Sweep	Set.	EAGLE-3	DFlash-4	**DStreamer**	*Target*
Context	512	0.64	1.28	**0**	*3.8*
	2048	2.14	4.28	**0**	*12.8*
	4096	4.14	8.28	**0**	*24.8*
	8192	8.14	16.28	**0**	*48.8*
Batch	64	1.07	2.14	**0**	*6.4*
	128	2.14	4.28	**0**	*12.8*
	256	4.28	8.56	**0**	*25.7*
	512	8.56	17.12	**0**	*51.4*

Draft KV required (GiB), batch 128 / context 2048.

Table 1: Scaling behavior on GPT-OSS-20B. DStreamer combines increasing long-generation speedup with zero additional draft KV state across context and batch scaling.

from proposal efficiency: useful speculative progress depends not only on how many tokens are proposed, but on how reliably those positions survive verification and how much computation is required to produce them (Yan et al., 2024; Bhendawade et al., 2025). DStreamer improves both sides of this tradeoff, concentrating speculation in a short, high-acceptance window while keeping its drafting cost low. This combination allows accepted speculative work to continue translating into end-to-end throughput as the server moves toward the high-concurrency regime.

Scaling with generation length. Table 1 studies GPT-OSS-20B (OpenAI, 2025) on AIME 2025 and AIME 2026 (Zhang & Math-AI, 2025; 2026) at batch size 128 as $L_{\max}$ increases from 512 to 8192 tokens. DStreamer's wall-clock speedup increases substantially with generation length even though its accepted length τ changes only modestly, indicating that the gain is not driven primarily by higher acceptance. As sequences grow, target decoding becomes increasingly dominated by KV-state access (Kwon et al., 2023; Pope et al., 2023). Because DStreamer drafts directly from the target KV state rather than maintaining a separate draft history, its speculative overhead grows more slowly relative to the target computation being amortized. Each accepted window therefore becomes increasingly valuable as generation length grows.



(a) (b) (c)

Figure 5: Rollout throughput over RLVR training: **(a)** Qwen3-4B, 4k context; **(b)** Qwen3-8B, 4k context; **(c)** Qwen3-14B, 8k context. AR and Speculative Decoding are shown alongside DStreamer (adapter trained once, then frozen) and DStreamer++ (adapter co-trained with the base policy).

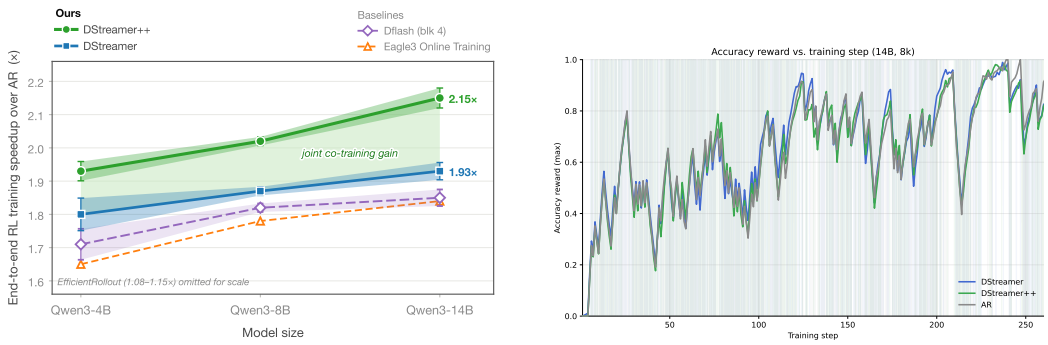
4.3 Online RLVR

RLVR setup. We evaluate DStreamer++ in synchronous GRPO training (Shao et al., 2024) on 8×H100 SXM5 GPUs with FSDP2 full sharding. Rollout generation and target-policy optimization use the same FSDP2-sharded model state, allowing us to keep training synchronous without maintaining a separate inference replica or redistributing parameters between rollout and update phases. We evaluate Qwen3 models from 4B to 14B parameters (Qwen Team, 2025) and context lengths from 4k to 8k. Starting from the common initialization in Section 4.1, we train the Qwen3 target policy for 500 steps on DAPO-Math (Yu et al., 2025) using verifiable math rewards. Across comparisons, the target-policy training configuration is held fixed; methods differ only in the mechanism used to generate rollouts and, for adaptive speculative methods, how the drafter is updated as the target policy evolves. We compare AR-GRPO, frozen DStreamer, DStreamer++, DFlash with block size 4 (Chen et al., 2026), EAGLE-3 Online Training (Iso et al., 2026), and EfficientRollout (Kim et al., 2026a), spanning autoregressive, fixed-drafter, and online-adaptive speculative baselines. All RLVR training experiments in our system are repeated with three independent random seeds. We report means across seeds, with shaded regions indicating ± 1 standard deviation where shown.

Speculative drift and acceptance-aligned adaptation. A speculative drafter calibrated to the policy at the start of RLVR can become increasingly stale as that policy evolves (Chen et al., 2025). Figure 10 tracks accepted tokens per speculative window throughout training across Qwen3-4B, 8B, and 14B. This trend is visible for both conventional speculative decoding and DStreamer when their drafters are held fixed: acceptance generally declines as the target policy moves. DStreamer starts from a checkpoint explicitly optimized for speculative acceptance and retains higher acceptance when frozen, but remains exposed to the same target nonstationarity. DStreamer++ instead continually optimizes complete proposal paths for acceptance under the current verifier, largely maintaining, and in some configurations improving, acceptance throughout training.

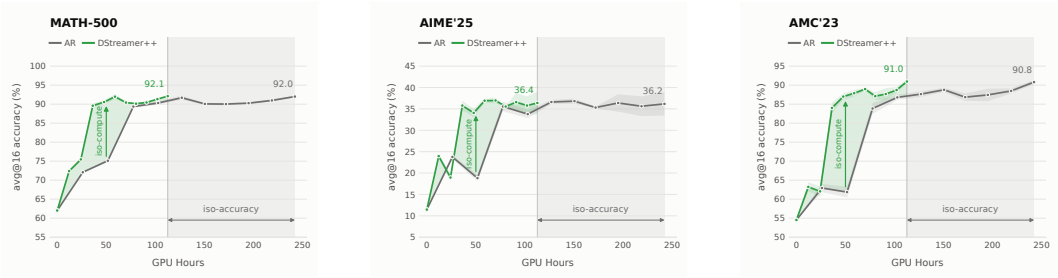
This loss of alignment has a direct systems consequence. Figure 5 examines how rollout throughput evolves across representative configurations from the same model and context-length range. As accepted prefix length decreases, proposal and verification work is amortized over fewer useful tokens, reducing the efficiency of speculative rollout. Consistent with this effect, the throughput advantage of fixed speculative methods erodes over training, most noticeably in the larger-model, longer-context regime, where conventional speculative decoding can lose nearly all of its advantage over autoregressive rollout. In contrast, DStreamer++ sustains, and in several configurations increases, rollout throughput as training proceeds. AR remains comparatively stable, indicating that the degradation is specific to the speculative mechanism rather than a uniform slowdown of the training workload. Continually optimizing sampled proposal paths for acceptance under the evolving verifier therefore preserves the systems benefit of speculation throughout RLVR.

End-to-end RLVR efficiency and learning quality. The acceptance gains above translate into lower end-to-end RLVR cost, rather than remaining a rollout-only improvement. Figure 6(a) reports speedup over AR-GRPO for the complete training loop. The gain increases with model scale over the evaluated range, consistent with target-model rollout becoming more expensive relative to DStreamer's lightweight adapter. More importantly, DStreamer++ consistently improves over the



(a) End-to-end training speedup (b) Training reward

Figure 6: **End-to-end RLVR efficiency and learning behavior.** **(a)** Training speedup over AR-GRPO across Qwen3 model scales; error bars show ± 1 s.d. over three seeds. EfficientRollout is omitted for scale. **(b)** Task-reward trajectories during Qwen3-14B RLVR at 8k context.



(a) (b) (c)

Figure 7: Accuracy versus training compute for Qwen3-14B at 8k context: **(a)** MATH-500, **(b)** AIME 2025, and **(c)** AMC 2023. AR-GRPO and DStreamer++ are trained independently and evaluated with `avg@16`. Shaded regions indicate the accuracy advantage at matched compute, while arrows indicate the compute difference at matched accuracy.

same adapter held frozen despite the cost of optimizing sampled proposal paths against the current verifier. Thus, closed-loop acceptance optimization yields a net end-to-end benefit: its additional training cost is outweighed by the rollout savings as the policy evolves.

These systems gains translate directly into learning efficiency. Figure 7 plots held-out accuracy against cumulative GPU-hours for Qwen3-14B at 8k context on MATH-500 (Lightman et al., 2023), AIME 2025 (Zhang & Math-AI, 2025), and AMC 2023 (Community, 2024). Curves report the mean over three independent training seeds, with narrow bands showing ± 1 standard deviation. At fixed compute, DStreamer++ reaches later stages of policy optimization sooner and attains higher accuracy through the early and middle portions of training; at matched accuracy, AR-GRPO requires substantially more GPU-hours. Both methods converge to comparable final accuracy within run-to-run variation, providing no evidence of degradation in learning quality from speculative rollouts. DStreamer++ therefore shifts the observed RLVR accuracy-versus-compute frontier by reducing the compute required to reach a given level of policy quality.

5 Conclusion

DStreamer demonstrates that single-pass parallel speculation can be optimized directly for the path-level acceptance criterion imposed by autoregressive verification, while reusing target KV state and preserving lossless decoding. DStreamer++ extends this objective to evolving RLVR policies, continually tracking the current verifier without altering the on-policy rollout distribution. Across high-throughput serving and synchronous RLVR, this yields up to $3.8\times$ higher aggregate throughput and $2.15\times$ end-to-end training speedup, with the same observed final accuracy.

AI USE STATEMENT

Generative AI tools were used for proofreading, sentence-level editing, and improving the clarity and presentation of the manuscript, as well as for limited assistance in writing and debugging auxiliary experimental and plotting scripts. The research ideas, method development, experimental decisions, and interpretation of results were carried out by the authors. All AI-assisted text and code were reviewed and verified by the authors, who take responsibility for the final content and results of this work.

ETHICS STATEMENT

This work focuses on improving the computational efficiency of language-model generation and reinforcement-learning rollouts. DStreamer and DStreamer++ use lossless speculative verification and therefore do not intentionally alter the output distribution of the underlying target policy. As with other efficiency improvements, reducing the cost of model serving and training may broaden access to both beneficial and potentially harmful applications of the underlying models; these broader impacts depend on how those models are developed and deployed.

REPRODUCIBILITY STATEMENT

The main text specifies the DStreamer and DStreamer++ objectives, architecture, training procedure, and experimental setup. The appendices provide complete training algorithms, optimization and replay details, hyperparameters and implementation configurations, serving and RLVR evaluation protocols, baseline configurations, and additional results. We further report runtime and memory overheads, provide proofs of losslessness and on-policy correctness, and give the derivations underlying our systems analysis.

To facilitate reproduction in standard research infrastructure, our serving evaluation builds on the open-source vLLM stack with continuous batching, while synchronous RLVR training uses PyTorch FSDP2. These setups mirror the high-throughput serving and distributed training regimes evaluated in the paper without relying on proprietary execution infrastructure. We intend to release the full DStreamer and DStreamer++ implementation, including adapter training, serving integration, online RLVR adaptation, experiment configurations, and evaluation scripts, pending institutional approval.

REFERENCES

Hasan Amin, Yuan Gao, Yaser Souri, Subhojit Som, Ming Yin, Rajiv Khanna, and Xia Song. Consistent diffusion language models. *arXiv preprint arXiv:2605.00161*, 2026.

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

Nikhil Bhendawade, Irina Belousova, Qichen Fu, Henry Mason, Antonie Lin, Mohammad Rastegari, and Mahyar Najibi. Speculative streaming: Efficient and scalable speculative decoding with multi-stream attention. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 19536–19559, Suzhou, China, November 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.986. URL https://aclanthology.org/2025.emnlp-main.986/.

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. In *Proceedings of the 41st International Conference on Machine Learning*, 2024.

Sahil Chaudhary. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca, 2023.

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

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, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Qiaoling Chen, Zijun Liu, Peng Sun, Shenggui Li, Guoteng Wang, Ziming Liu, Yonggang Wen, Siyuan Feng, and Tianwei Zhang. Respec: Towards optimizing speculative decoding in reinforcement learning systems. *arXiv preprint arXiv:2510.26475*, 2025.

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

Math-AI Community. Amc23: Evaluation dataset for 2023 american mathematics competitions, 2024. URL https://huggingface.co/datasets/math-ai/amc23.

DeepSeek-AI. Deepseek-v3 technical report, 2024. URL https://arxiv.org.

DeepSeek-AI. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645:633–638, 2025.

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

Shijing Hu, Jingyang Li, Zhihui Lu, and Pan Zhou. Bridging draft policy misalignment: Group tree optimization for speculative decoding. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=dwPdYFqVWO.

Hayate Iso, Tiyasa Mitra, Sudipta Mondal, Rasoul Shafipour, Venmugil Elango, et al. Accelerating rl post-training rollouts via system-integrated speculative decoding. *arXiv preprint arXiv:2604.26779*, 2026.

Jie Jiang, Xing Sun, Ruotian Chen, Jianan Su, and Kaixin Shen. Performance-driven policy optimization for speculative decoding with adaptive windowing, 2026a. URL https://arxiv.org/abs/2605.14978.

Weifan Jiang, Krishna Teja Chitty-Venkata, Megan Flynn, Reed Meyerson, Zhenting Qi, Tianyu Wu, Eldar Kurtic, Minlan Yu, and Alexandre Marques. H-spec: Parallel speculative decoding without a drafter-side kv cache. *arXiv preprint arXiv:2609.24197*, 2026b.

Minseo Kim, Minjae Lee, Seunghyuk Oh, Kevin Galim, Donghoon Kim, Coleman Hooper, Harman Singh, Amir Gholami, Hyung Il Koo, and Wonjun Kang. Efficientrollout: System-aware self-speculative decoding for rl rollouts. *arXiv preprint arXiv:2606.18967*, 2026a.

Minseo Kim, Chenfeng Xu, Coleman Hooper, Harman Singh, Ben Athiwaratkun, Ce Zhang, Kurt Keutzer, and Amir Gholami. Cdlm: Consistency diffusion language models for faster sampling. In *Proceedings of the 9th MLSys Conference*, 2026b.

Kimi Team. Kimi k1.5: Scaling reinforcement learning with LLMs. *arXiv preprint arXiv:2501.12599*, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. 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 (SOSP '23)*, pp. 611–626, 2023. doi: 10.1145/3600006.3613165.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *PMLR*, pp. 19274–19286, 2023.

Jia LI, Edward Beeching, Lewis Tunstall, Ben Lipkin, Roman Soletskyi, Shengyi Costa Huang, Kashif Rasul, Longhui Yu, Albert Jiang, Ziju Shen, Zihan Qin, Bin Dong, Li Zhou, Yann Fleureau, Guillaume Lample, and Stanislas Polu. Numinamath. `[https://huggingface.co/datasets/AI-MO/NuminaMath-1.5](https://github.com/project-numina/aimo-progress-prize/blob/main/report/numina_dataset.pdf)`, 2024.

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

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. *arXiv preprint arXiv:2305.20050*, 2023.

Aaron Lou, Chenlin Meng, and Stefano Ermon. Discrete diffusion modeling by estimating the ratios of the data distribution. In *Proceedings of the 41st International Conference on Machine Learning*, 2024.

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.

Team Olmo, Allyson Ettinger, Amanda Bertsch, Bailey Kuehl, David Graham, David Heineman, Dirk Groeneveld, Faeze Brahman, Finbarr Timbers, Hamish Ivison, Jacob Morrison, Jake Poznanski, Kyle Lo, Luca Soldaini, Matt Jordan, Mayee Chen, Michael Noukhovitch, Nathan Lambert, Pete Walsh, Pradeep Dasigi, Robert Berry, Saumya Malik, Saurabh Shah, Scott Geng, Shane Arora, Shashank Gupta, Taira Anderson, Teng Xiao, Tyler Murray, Tyler Romero, Victoria Graf, Akari Asai, Akshita Bhagia, Alexander Wettig, Alisa Liu, Aman Rangapur, Chloe Anastasiades, Costa Huang, Dustin Schwenk, Harsh Trivedi, Ian Magnusson, Jaron Lochner, Jiacheng Liu, Lester James V. Miranda, Maarten Sap, Malia Morgan, Michael Schmitz, Michal Guerquin, Michael Wilson, Regan Huff, Ronan Le Bras, Rui Xin, Rulin Shao, Sam Skjonsberg, Shannon Zejiang Shen, Shuyue Stella Li, Tucker Wilde, Valentina Pyatkin, Will Merrill, Yapei Chang, Yuling Gu, Zhiyuan Zeng, Ashish Sabharwal, Luke Zettlemoyer, Pang Wei Koh, Ali Farhadi, Noah A. Smith, and Hannaneh Hajishirzi. Olmo 3, 2025. URL `https://arxiv.org/abs/2512.13961`.

OpenAI. OpenAI o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

OpenAI. gpt-oss-120b & gpt-oss-20b model card. OpenAI Model Card, 2025.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. In *Proceedings of Machine Learning and Systems*, volume 5, pp. 606–624, 2023.

Yu-Yang Qian, Junda Su, Lanxiang Hu, Peiyuan Zhang, Zhijie Deng, Peng Zhao, and Hao Zhang. d3llm: Ultra-fast diffusion LLM using pseudo-trajectory distillation. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026.

Qwen Team. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Yuji Ren, Chenkai Xu, Zhuocheng Gong, Jianguo Li, and Zhijie Deng. CForce: Boosting parallel decoding for dllms via consistency forcing. *arXiv preprint arXiv:2608.13925*, 2026.

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

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *arXiv preprint arXiv:2407.08608*, 2024.

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

Shenzhi Wang, Le Yu, Chang Gao, Chujie Zheng, Shixuan Liu, Rui Lu, Kai Dang, Xiong-Hui Chen, Jianxin Yang, Zhenru Zhang, Yuqiong Liu, An Yang, Andrew Zhao, Yang Yue, Shiji Song, Bowen Yu, Gao Huang, and Junyang Lin. Beyond the 80/20 rule: High-entropy minority tokens drive effective reinforcement learning for llm reasoning. In *Advances in Neural Information Processing Systems*, 2025a.

Siqi Wang, Hailong Yang, Junjie Zhu, Xuezhu Wang, Yufan Xu, and Depei Qian. Rlhfspec: Breaking the efficiency bottleneck in rlhf training via adaptive drafting. *arXiv preprint arXiv:2512.04752*, 2025b.

Minghao Yan, Saurabh Agarwal, and Shivaram Venkataraman. Decoding speculative decoding. *arXiv preprint arXiv:2402.01528*, 2024.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for transformer-based generative models. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, 2022. URL https://www.usenix.org/conference/osdi22/presentation/yu.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. DAPO: An open-source LLM reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

Weihao Zeng, Yuzhen Huang, Qian Liu, Wei Liu, Keqing He, Zejun Ma, and Junxian He. SimpleRL-Zoo: Investigating and taming zero reinforcement learning for open base models in the wild. In *Conference on Language Modeling (COLM)*, 2025. URL https://arxiv.org/abs/2503.18892.

Zhang et al. Few-step diffusion language models via trajectory self-distillation. *arXiv preprint arXiv:2602.12262*, 2026a.

Jiebin Zhang, Zhenghan Yu, Liang Wang, Nan Yang, Eugene J. Yu, Zheng Li, Yifan Song, Dawei Zhu, Xingxing Zhang, Furu Wei, and Sujian Li. Learning to draft: Adaptive speculative decoding with reinforcement learning. In *International Conference on Learning Representations*, 2026b.

Yifan Zhang and Team Math-AI. American invitational mathematics examination (aime) 2025, 2025.

Yifan Zhang and Team Math-AI. American invitational mathematics examination (aime) 2026, 2026.

Yizhou Zhang, Ning Lv, Teng Wang, and Jisheng Dang. Fastgrpo: Accelerating policy optimization via concurrency-aware speculative decoding and online draft learning. *arXiv preprint arXiv:2509.21792*, 2025.

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, et al. Pytorch fsdp: Experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

Haizhong Zheng, Yang Zhou, Brian R. Bartoldson, Bhavya Kailkhura, Fan Lai, Jiawei Zhao, and Beidi Chen. Act only when it pays: Efficient reinforcement learning for llm reasoning via selective rollouts. In *Advances in Neural Information Processing Systems*, 2025.

APPENDICES

A Related Work **16**

B Algorithms and Optimization **17**

B.1 DStreamer Acceptance-Aligned Training 17

B.2 DStreamer++ Online Co-Training 17

B.3 Adapter GRPO Objective and Replay 18

C Training and Implementation Details **19**

C.1 DStreamer Initialization and RL Training 19

C.2 DStreamer++ Online Adaptation 20

C.3 Adapter-to-Target Layer Mapping 21

C.4 Hyperparameters and Implementation Configuration 22

D Experimental Protocol **23**

D.1 Fixed-Target High-Throughput Serving 23

D.2 Online RLVR 23

D.3 Baselines, Decoding Configuration, and Metrics 23

E Additional DStreamer Analysis **24**

E.1 Acceptance-Oriented RL from the Distillation Initialization 24

E.2 Reward Ablations 24

E.3 Speculative Drift and Online Adaptation 24

F Runtime and Memory Overhead **24**

F.1 Online Adaptation Overhead 24

F.2 Memory and State Retention 25

G Losslessness and On-Policy Correctness **25**

G.1 Exactness of Lossless Verification 25

G.2 On-Policy Rollouts Under an Evolving Target 26

H Arithmetic-Intensity Analysis **27**

H.1 Roofline Model and Assumptions 27

H.2 Autoregressive Decoding 27

H.3 Batching and the KV-Limited Ceiling 28

H.4 Implication for Multi-Token Verification 28

A Related Work

Speculative decoding and parallel drafting. Speculative decoding accelerates autoregressive generation by proposing multiple future tokens with a cheaper mechanism and verifying them in parallel with the target model (Leviathan et al., 2023; Chen et al., 2023). Early approaches use a separate autoregressive draft model, while subsequent work reduces drafting overhead through multi-token prediction heads, adaptive draft trees, or target-model features (Cai et al., 2024; DeepSeek-AI, 2024; Li et al., 2025). Speculative Streaming (Bhendawade et al., 2025) instead integrates parallel speculation into the target model, allowing candidate tokens to reuse its contextual representations without a separate autoregressive drafter. More recently, DFlash (Chen et al., 2026) uses a lightweight block-diffusion drafter conditioned on target-model features to generate a speculative block in parallel. DStreamer builds on this direction of lightweight parallel drafting, using an in-model adapter that operates directly on the target KV state. Our focus is on making such speculation effective in throughput-oriented generation: optimizing parallel proposals for joint, prefix-sensitive verification, maintaining efficiency under high-concurrency batching, and continually adapting the drafter when the verifier itself evolves during online RLVR.

Diffusion language models and step distillation. Masked discrete diffusion language models generate tokens through iterative denoising and naturally support parallel prediction over masked positions (Sahoo et al., 2024; Lou et al., 2024; Nie et al., 2025). Recent work reduces the number of denoising steps through trajectory or consistency-based distillation, training models to approximate multi-step denoising trajectories with fewer sampling steps (Amin et al., 2026; Kim et al., 2026b; Zhang et al., 2026a; Qian et al., 2026). Closest to our initialization, Amin et al. (2026) enforce consistency across different masking levels while anchoring predictions to ground-truth tokens. We use these ideas only to warm-start the speculative adapter: consistency distillation collapses iterative denoising into the single parallel proposal pass required by our decoder, after which the adapter is optimized directly for speculative acceptance rather than diffusion-model likelihood or sampling efficiency.

Learning objectives for speculative decoding. Beyond conventional token-level distillation, recent work has explored objectives that more closely reflect the inference-time behavior of speculative decoding. Group Tree Optimization (GTO) (Hu et al., 2026) addresses the mismatch between single-path training and tree-structured speculation by optimizing a tree-level reward based on expected acceptance length. Learning to Draft (LTD) (Zhang et al., 2026b) instead learns to allocate computation within each draft-and-verify cycle, directly optimizing throughput as the reward. PPOW (Jiang et al., 2026a) shifts drafter training from token-level imitation to window-level reinforcement learning, using a prefix-sensitive reward that reflects speculative utility. These methods align speculative training or control more closely with decoding-time performance, but do not address the proposal mismatch that arises in one-shot parallel drafting: the proposal factorizes predictions across positions, whereas verification evaluates the resulting sequence under autoregressive target conditionals. DStreamer targets this joint–marginal gap directly by optimizing complete sampled proposal paths for acceptance under the target verifier.

Speculative decoding for evolving RL policies. Speculative decoding has increasingly been used to accelerate RL rollout generation, where the target policy itself changes throughout training. Some approaches keep the speculative mechanism fixed or adapt only the surrounding system: System-Integrated Speculative Decoding (Iso et al., 2026) primarily uses a frozen EAGLE-3 drafter, while RLHF-Spec (Wang et al., 2025b) adapts scheduling and batch placement without retraining the drafter. EfficientRollout (Kim et al., 2026a) instead avoids a persistent learned drafter by deriving a self-speculative model directly from the current policy, thereby remaining synchronized as the policy evolves. ReSpec (Chen et al., 2025) and FastGRPO (Zhang et al., 2025) directly address drafter staleness by periodically or continuously updating a learned drafter toward the evolving target through supervised distillation. DStreamer++ takes a different approach. Rather than repeatedly distilling token-level marginals from the evolving policy, it treats the speculative adapter as a policy and optimizes the complete proposal paths generated by its own current behavior. Each update changes the distribution of proposals the adapter will subsequently produce and therefore the mistakes it encounters under autoregressive verification, so training proceeds in a closed loop: sample with the current adapter, measure acceptance under the current verifier, and optimize directly against that feedback.

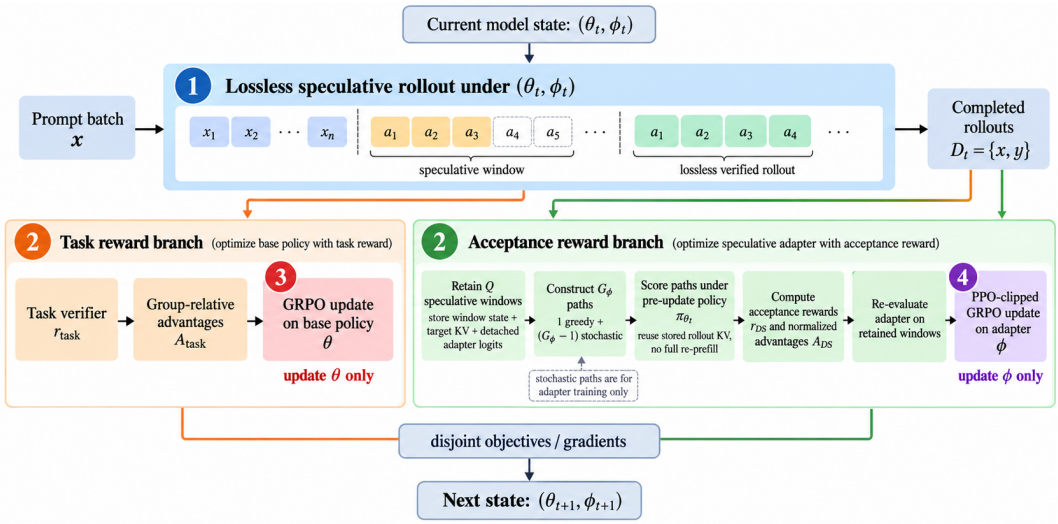


Figure 8: **DStreamer++: acceptance-aligned adaptation during RLVR.** Rollouts are generated losslessly under the current state (θ_t, ϕ_t), after which task rewards update only θ and acceptance rewards update only ϕ. Candidate paths are scored under the pre-update policy θ_t, keeping the drafter aligned to the current verifier while preserving exact on-policy rollouts.

This continually trains the adapter for the behavior actually exercised during speculative decoding, while a disjoint task reward independently updates the base policy.

Orthogonal RL efficiency. Other work improves RL efficiency by reducing the amount of computation spent on uninformative training examples, for example by selecting which prompts to roll out (Zheng et al., 2025) or which generated tokens contribute to the policy update (Wang et al., 2025a). These approaches target the amount of training computation performed, whereas DStreamer targets the cost of generating the rollouts themselves; the two directions are complementary.

B Algorithms and Optimization

B.1 DStreamer Acceptance-Aligned Training

Algorithm 1 expands the fixed-target DStreamer procedure introduced in Section 3. Starting from the consistency-distilled initialization, we optimize only the adapter parameters while keeping the target policy fixed. For each training context, a single adapter forward pass produces proposal logits for the full speculative window. These logits define one greedy path and $C_\phi - 1$ stochastic paths, so the entire path group is constructed without additional proposal passes. Each path is then evaluated autoregressively under the frozen target using the acceptance-aligned reward of Eq. 3. Rewards are normalized within the path group to obtain a path-level advantage, which is used to update the adapter with the replay objective described in Section B.3.

The stochastic paths are used only for adapter training. At inference, DStreamer produces a single proposal window and applies the lossless verification procedure of Section 2; C_ϕ therefore affects training cost but not inference-time decoding.

B.2 DStreamer++ Online Co-Training

Algorithm 2 expands the online adaptation procedure introduced in Section 3. At RLVR step t, rollouts are generated with the current pair (θ_t, ϕ_t) using lossless speculative decoding. A bounded set of speculative windows is retained from each completion together with the detached adapter proposal logits and target state required for later scoring. For every retained window, we construct the same greedy-plus-stochastic path group used by DStreamer and evaluate its acceptance reward

Algorithm 1 DStreamer: Acceptance-Aligned Adapter Training

Require: Frozen target policy π_θ, adapter q_ϕ initialized by consistency distillation, context dataset $\mathcal{D}$, speculative window size w, path-group size C_ϕ, compatibility cutoff K_{acc}, discount γ, reward weights $\lambda_{\text{hard}}, \lambda_{\text{soft}}$, local target context L_{local}

for each adapter training step **do**
  Sample context prefix $x_{<t} \sim \mathcal{D}$
  Run the adapter policy q_ϕ once on $x_{<t}$ to obtain proposal logits $\{\ell_i\}_{i=1}^{w}$
  Construct greedy path $\mathbf{a}^{(1)}$: $a_i^{(1)} \leftarrow \arg\max_v \ell_i(v)$
  for $c = 2, \ldots, C_\phi$ **do**
    for $i = 1, \ldots, w$ **do**
      $a_i^{(c)} \sim \text{Categorical}\left(\text{softmax}(\ell_i)\right)$
    end for
  end for
  Let c_t denote the most recent L_{local} tokens of $x_{<t}$
  for $c = 1, \ldots, C_\phi$ **do**
    for $i = 1, \ldots, w$ **do**
      $p_i^{(c)} \leftarrow \pi_\theta\left(\cdot \mid c_t \oplus \mathbf{a}_{<i}^{(c)}\right)$
      $I_i^{(c)} \leftarrow \mathbf{1}\left[a_i^{(c)} \in \text{TopK}\left(p_i^{(c)}, K_{\text{acc}}\right)\right]$
    end for
    $r_{\text{hard}}^{(c)} \leftarrow \sum_{i=1}^{w} \gamma^{i-1} \prod_{j=1}^{i} I_j^{(c)}$
    $r_{\text{soft}}^{(c)} \leftarrow \sum_{i=1}^{w} \log p_i^{(c)}\left(a_i^{(c)}\right)$
    $r_{\text{DS}}^{(c)} \leftarrow \lambda_{\text{hard}} r_{\text{hard}}^{(c)} + \lambda_{\text{soft}} r_{\text{soft}}^{(c)}$
  end for
  Group-normalize path rewards: $\hat{A}_{\text{DS}}^{(c)} \leftarrow \dfrac{r_{\text{DS}}^{(c)} - \bar{r}_{\text{DS}}}{\sigma_{\text{DS}} + \epsilon_{\text{norm}}}$
  Re-evaluate the adapter with gradients enabled and update ϕ using the path-level PPO-clipped objective of Section B.3
  Keep θ frozen
end for

under the pre-update target policy π_{θ_t}. The resulting adapter advantages therefore correspond to the same verifier that generated the rollout.

Only after all acceptance rewards have been computed do we update the base policy with the task objective. The retained windows are then replayed through the adapter with gradients enabled, and ϕ is updated using the acceptance-oriented objective while gradients into the target model are disabled. Thus, task optimization changes θ, while acceptance optimization independently tracks that changing policy through ϕ.

The rollout-time adapter graph is not retained; detached proposal logits define the old adapter policy for replay, while the gradient-bearing forward pass is performed only during the subsequent adapter update described in Section B.3.

B.3 Adapter GRPO Objective and Replay

DStreamer and DStreamer++ use the same path-level replay objective for the speculative adapter. For a training window n, let $\mathbf{a}^{(n,c)} = (a_1^{(n,c)}, \ldots, a_w^{(n,c)})$ denote candidate path c, and let $\hat{A}_{\text{DS}}^{(n,c)}$ denote its group-normalized acceptance advantage. During path construction, the adapter proposal logits are detached and retained as the old-policy reference; the rollout computation graph is not preserved. During optimization, the retained window state is replayed through the current adapter with gradients enabled.

Algorithm 2 DStreamer++: Online Policy and Adapter Co-Training

Require: Policy π_{θ_0}, DStreamer adapter q_{ϕ_0}, dataset $\mathcal{D}$, prompt batch size B, rollout group size G, path-group size C_ϕ, retained-window limit N_{ret}, speculative window size w, compatibility cutoff K_{acc}, discount γ, reward weights $\lambda_{\text{hard}}, \lambda_{\text{soft}}$, local target context L_{local}, adapter minibatch size M_ϕ

for training step $t = 0, 1, \ldots$ **do**
  Sample prompts $\{x_b\}_{b=1}^{B} \sim \mathcal{D}$
  Generate completions $\{y^{(b,g)}\}_{b=1,g=1}^{B,G}$ with lossless speculative decoding under (θ_t, ϕ_t)
  for each completion **do**
    Uniformly retain up to N_{ret} speculative windows across generated positions
    Store window state, target KV state, and detached adapter logits $\{\ell_{n,i}^{\text{old}}\}_{i=1}^{w}$
  end for
  Compute task rewards and group-relative advantages $\hat{A}_{\text{task}}^{(b,g)}$
  for each retained window n **do**
    Construct C_ϕ paths from $\{\ell_{n,i}^{\text{old}}\}_{i=1}^{w}$: one greedy and $C_\phi - 1$ stochastic paths
    for $c = 1, \ldots, C_\phi$ **do**
      Score $\mathbf{a}^{(n,c)}$ under the pre-update π_{θ_t} using the DStreamer reward of Algorithm 1 and the most recent L_{local} context tokens
      Compute $r_{\text{DS}}^{(n,c)} = \lambda_{\text{hard}} r_{\text{hard}}^{(n,c)} + \lambda_{\text{soft}} r_{\text{soft}}^{(n,c)}$
    end for
    Group-normalize $\{r_{\text{DS}}^{(n,c)}\}_{c=1}^{C_\phi}$ to obtain $\hat{A}_{\text{DS}}^{(n,c)}$
  end for
  Update $\theta_t \to \theta_{t+1}$ with task GRPO; keep ϕ_t fixed
  Partition retained windows into adapter minibatches of size M_ϕ
  for each adapter minibatch **do**
    Re-evaluate q_ϕ on the stored pre-update window/KV states with gradients enabled
    Compute path-level PPO ratios against detached ℓ_n^{old}
    Update ϕ with PPO-clipped GRPO using path advantages $\hat{A}_{\text{DS}}^{(n,c)}$
  end for
  Set resulting adapter parameters to ϕ_{t+1}
end for

Because the adapter predicts the w proposal positions in parallel, the probability assigned to a sampled path factorizes across positions. We define the old and current path log-probabilities as

$$\log p_{\text{old}}^{(n,c)} = \sum_{i=1}^{w} \log q_{\phi_{\text{old}},i}\left(a_i^{(n,c)} \mid n\right), \qquad \log p_{\text{new}}^{(n,c)} = \sum_{i=1}^{w} \log q_{\phi,i}\left(a_i^{(n,c)} \mid n\right). \tag{4}$$

The corresponding path-level importance ratio is

$$\rho_n^{(c)} = \exp\left(\log p_{\text{new}}^{(n,c)} - \text{sg}\left[\log p_{\text{old}}^{(n,c)}\right]\right). \tag{5}$$

We optimize the PPO-clipped surrogate (Schulman et al., 2017; Shao et al., 2024)

$$\mathcal{L}_{\text{DS}} = -\mathbb{E}_{n,c}\left[\min\left(\rho_n^{(c)} \hat{A}_{\text{DS}}^{(n,c)}, \text{clip}\left(\rho_n^{(c)}, 1 - \epsilon_{\text{clip}}, 1 + \epsilon_{\text{clip}}\right) \hat{A}_{\text{DS}}^{(n,c)}\right)\right]. \tag{6}$$

The stored rollout logits serve as the detached old-policy reference, while gradients are obtained by replaying the retained window through the current adapter. The path advantage $\hat{A}_{\text{DS}}^{(n,c)}$ and importance ratio $\rho_n^{(c)}$ are computed for each candidate path, and the clipped objective updates only ϕ. For DStreamer++, the corresponding advantages are computed from rewards evaluated under the pre-update target snapshot θ_t.

C Training and Implementation Details

C.1 DStreamer Initialization and RL Training

We construct the post-training corpus from a mixture of mathematical and coding data, including SimpleRL (Zeng et al., 2025), Dolci-Think (Olmo et al., 2025), Code Alpaca (Chaudhary,

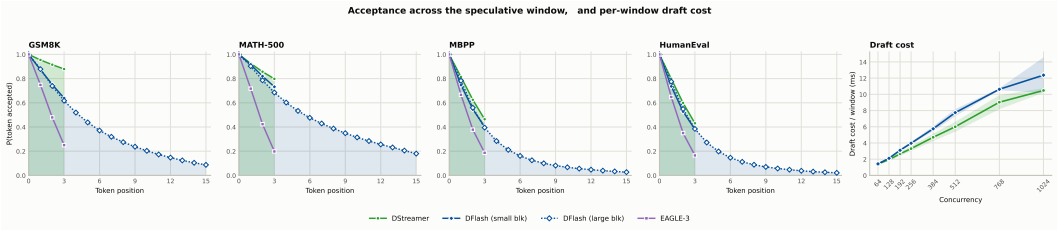


Figure 9: **Acceptance and drafting cost under batched inference.** Position-wise acceptance across GSM8K, MATH-500, MBPP, and HumanEval (left four panels), together with per-window drafting cost as serving concurrency increases (right).

2023), and NuminaMath (LI et al., 2024). A disjoint subset of this corpus is reserved for the acceptance-aligned DStreamer stage, while the remaining data are used for the preceding supervised and consistency-distillation initialization (Amin et al., 2026). The consistency-distillation loss (Eq. 1) uses $\lambda_{\text{sc}} = 1.0$ and $\lambda_{\text{ce}} = 0.1$. Each reserved example contains a prompt and a stored reference completion. To construct a training context, we tokenize their concatenation and sample a cut point uniformly between the end of the prompt and the latest valid position that leaves room for the speculative window, subject to the training-context limit. The resulting context therefore ranges from the prompt alone to the prompt followed by a random-length prefix of the stored completion. The reference completion is used only to construct the context; it does not define the acceptance reward, which is computed from the frozen target policy on the adapter's proposed continuations.

We train the acceptance-aligned adapter for 10,000 steps on eight NVIDIA H100 GPUs with a per-GPU context batch size of 64, while keeping the target model frozen. We use AdamW with learning rate 3×10^{-5}, weight decay 0.01, gradient-norm clipping at 1.0, and 100 linear warmup steps followed by cosine decay. The adapter objective uses PPO clipping with $\epsilon_{\text{clip}} = 0.2$. Unless otherwise stated, we use $w_{\text{train}} = 7$ speculative positions, corresponding to an implementation window size of $W_{\text{win,train}} = 8$, $C_\phi = 8$, $K_{\text{acc}} = 1$, $\gamma = 0.9$, and $\lambda_{\text{hard}} = \lambda_{\text{soft}} = 0.5$. The seven stochastic paths are sampled independently at each speculative position from the full-vocabulary adapter softmax at temperature 1, without top-K or top-p filtering.

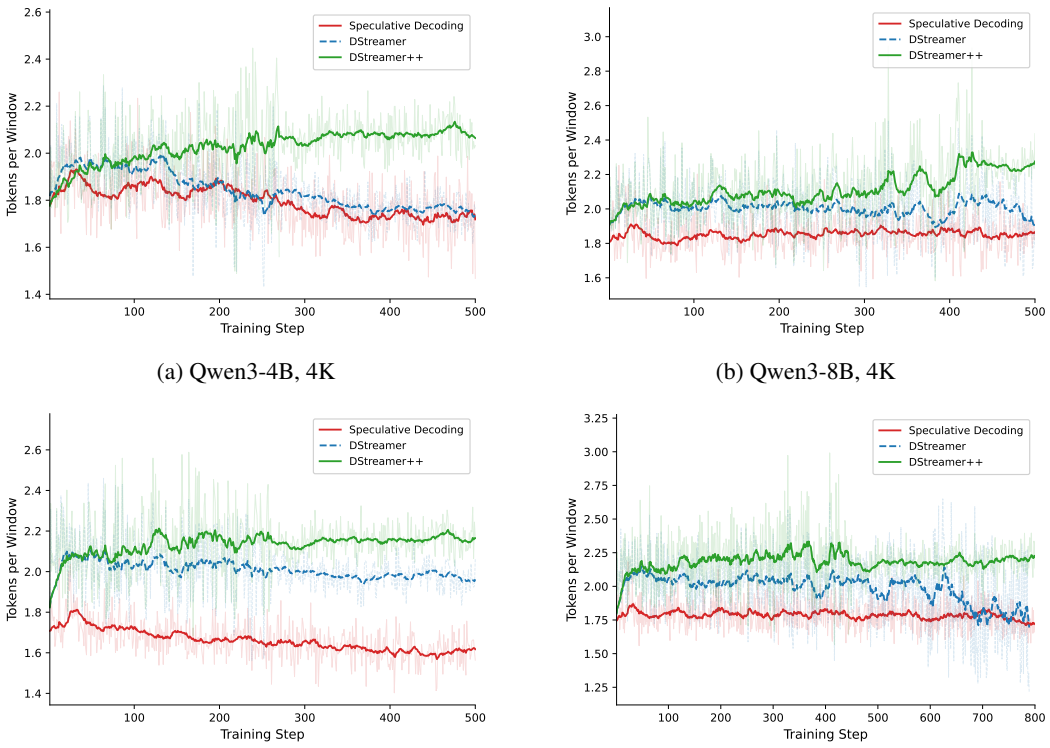
For reward computation, DStreamer conditions the frozen target on the most recent $L_{\text{local}} = 1024$ tokens preceding the speculative window, together with the preceding tokens of each candidate path. The same local target context is used for all C_ϕ paths associated with a given window, while the target distribution is evaluated autoregressively along each path's own proposed prefix. This local-context restriction is used only for the adapter-training reward and does not alter the full-context lossless verification procedure used at inference.

Unless otherwise noted, the GPT-OSS-20B adapter uses the same consistency-distillation warm-start and $10{,}000$-step acceptance-aligned RL procedure as the Qwen3 adapters; the two configurations differ only in the adapter-to-target layer mapping described in Appendix C.3.

The adapter stage is comparatively lightweight to train: the target model stays frozen, and the entire group of C_ϕ candidate paths is produced by a single adapter forward pass over the masked window rather than by autoregressive drafting; reward scoring then evaluates the frozen target only over the short candidate suffix appended to the cached L_{local} context, so no full-context re-prefill is incurred per step. Each optimization step is therefore dominated by a single adapter pass rather than by target generation, which keeps the wall-clock cost of the $10{,}000$-step schedule modest. For GPT-OSS-20B, the complete adapter pipeline, comprising diffusion supervised fine-tuning and consistency distillation ($\approx$ 3 hours) followed by acceptance-aligned RL ($\approx$ 1.5 hours), completes in roughly 4.5 hours on a single NVIDIA H100.

C.2 DStreamer++ Online Adaptation

During RLVR, DStreamer++ constructs the adapter-training batch from the same rollouts used to update the base policy. For each completion, we sample up to $N_{\text{ret}} = 64$ speculative windows uniformly over the generated trajectory, using all available windows when fewer than N_{ret} are present. Sampling is repeated independently at every RLVR iteration. For each selected window, we store a



(a) Qwen3-4B, 4K

(b) Qwen3-8B, 4K

(c) Qwen3-14B, 8K

(d) Qwen3-14B, 4K

Figure 10: **Acceptance dynamics during RLVR.** Mean accepted length $\tau = 1 + \alpha$ over training. Panels **(a)** through **(c)** correspond to the configurations reported in Figure 5; panel **(d)** adds Qwen3-14B at 4K context. DStreamer++ continually adapts the drafter as the target policy evolves, while frozen speculative predictors can lose acceptance over training. Autoregressive decoding is omitted because it emits exactly one token per forward pass by construction.

lightweight CPU-side record containing the local context token IDs, detached rollout-time adapter logits, rollout index, and context length. The rollout computation graph is not retained.

To avoid re-prefilling the target context during acceptance scoring and adapter replay, we also retain the target model's materialized generation KV cache on GPU together with a live-position mask identifying valid cache entries. For a selected window, its rollout index and context length identify the relevant prefix of the per-layer key and value tensors, which is sliced and reused for all candidate paths associated with that window. Only the short candidate suffix is evaluated on top of this cached prefix. No target hidden states are stored, and the retained KV tensors receive no gradient during the adapter update.

All acceptance rewards for iteration t are evaluated under the pre-update target snapshot θ_t using the most recent $L_{\text{local}} = 1024$ context tokens and the same C_ϕ, K_{acc}, γ, and reward weights used for fixed-target DStreamer. After reward computation, the base policy is updated with the RLVR task objective. Retained windows are then flattened across prompts and completions and processed in adapter minibatches of size 1024. The adapter is re-evaluated on each retained window with gradients enabled, while the stored rollout logits define the detached old-policy probabilities in the path-level replay objective of Section B.3. The number of adapter optimizer steps per RLVR iteration therefore depends on the number of retained windows.

C.3 Adapter-to-Target Layer Mapping

Each DStreamer adapter layer attends directly to the cached keys and values of a selected target-model layer. We use eight adapter layers and distribute their KV taps across target depth, keeping

Configuration	DStreamer	DStreamer++
Speculative positions w_{train}	7	7
Implementation window size $W_{\text{win,train}}$	8	8
Path-group size C_ϕ	8	8
Compatibility cutoff K_{acc}	1	1
Discount γ	0.9	0.9
λ_{hard}	0.5	0.5
λ_{soft}	0.5	0.5
Local target context L_{local}	1024	1024
Proposal sampling temperature	1.0	1.0
Proposal sampling truncation	None	None
PPO ratio	Path-level	Path-level
PPO clip ϵ_{clip}	0.2	0.2
Advantage-norm. ϵ_{norm}	10^{-8}	10^{-8}
Optimizer	AdamW	AdamW
Adapter learning rate	3×10^{-5}	3×10^{-5}
Weight decay	0.01	0.01
Gradient-norm clip	1.0	1.0
Warmup	100 steps	100 steps
Learning-rate schedule	Cosine	Cosine
Fixed-target RL steps	10,000	N/A
Context batch size	64 / GPU	N/A
Retained windows N_{ret}	N/A	64 / completion
Adapter replay minibatch M_ϕ	N/A	1024 windows
Training hardware	8×H100	8×H100

Table 2: **DStreamer adapter-training configuration.** The implementation window contains $W_{\text{win,train}} = 8$ slots, of which $w_{\text{train}} = 7$ are speculative proposal positions, following the general relationship $W_{\text{win}} = w + 1$. Stochastic paths are sampled independently at each position from the full-vocabulary adapter softmax without top-K or top-p truncation. For DStreamer++, N_{ret} denotes the maximum number of speculative windows retained from each completion, while M_ϕ denotes the number of retained window groups processed in each adapter replay minibatch.

Target model	Target layers used by adapter layers $0, \ldots, 7$
Qwen3-4B	$[0, 4, 8, 12, 16, 20, 24, 28]$
GPT-OSS-20B	$[1, 5, 7, 11, 13, 17, 19, 23]$

Table 3: **Adapter-to-target KV mapping.** Each adapter layer reads the cached KV state of the corresponding target layer. GPT-OSS taps are restricted to full-attention layers.

the mapping fixed between adapter training and inference. Table 3 reports the mappings used in our serving experiments.

For Qwen3-4B, whose 36 decoder layers use full attention, we select uniformly strided target layers. GPT-OSS-20B alternates sliding-window and full-attention layers; we restrict adapter taps to full-attention layers so that each adapter layer has access to the target's full cached context rather than a truncated sliding-window state.

C.4 Hyperparameters and Implementation Configuration

Table 2 summarizes the adapter-training configuration used for DStreamer and DStreamer++. Unless noted otherwise, the two settings use the same acceptance objective and adapter optimization hyperparameters.

D EXPERIMENTAL PROTOCOL

D.1 FIXED-TARGET HIGH-THROUGHPUT SERVING

We evaluate fixed-target serving using Qwen3-4B (Qwen Team, 2025) and GPT-OSS-20B (OpenAI, 2025) on a single NVIDIA H100 80GB GPU with vLLM (Kwon et al., 2023). All methods use BF16 inference, tensor parallelism $= 1$, FlashAttention 3 (Shah et al., 2024), CUDA graphs, prefix caching, and no quantization. We set the vLLM GPU-memory utilization target to 0.85 and use a maximum model length of 8192 tokens. To characterize the high-throughput operating regime, we sweep `max_num_seqs` over $\{64, 128, 192, 256, 384, 512, 768, 1024\}$ and report both aggregate output throughput and per-request output rate.

For the primary speculative-decoding comparison, DStreamer, EAGLE3 (Li et al., 2025), and DFlash (Chen et al., 2026) each propose three speculative tokens before target verification. DStreamer realizes this with $w_{\text{serve}} = 3$ speculative proposal positions, corresponding to an implementation window of $W_{\text{win,serve}} = 4$ slots, with one slot reserved for the target token. We additionally evaluate DFlash with its default 16-token block, while using the three-token configuration for the primary baseline because it gives the highest DFlash throughput in our serving sweep.

Throughput timing covers the complete `llm.generate` call, including prefill and decode, but excludes engine initialization, CUDA-graph capture, warmup, and tokenization. We therefore refer to this metric as *full-generation throughput* or *end-to-end generation speedup*, rather than request-level serving latency.

D.2 ONLINE RLVR

We evaluate online RLVR with Qwen3-4B, Qwen3-8B, and Qwen3-14B across 4K and 8K context budgets. Training uses synchronous GRPO (Shao et al., 2024) on eight NVIDIA H100 SXM5 GPUs with PyTorch FSDP2 (Zhao et al., 2023) full sharding and BF16 precision. For each prompt, we sample $G = 8$ completions from the current policy at temperature 1. The prompt batch size is set to the largest batch that fits for each model and context configuration. All rollout methods use the same base-policy training configuration.

The task reward is binary mathematical correctness evaluated with `math-verify`; no additional task-reward shaping is used. The base policy is optimized with AdamW using learning rate 1×10^{-5}, weight decay 0.01, gradient-norm clipping at 1.0, and PPO clipping parameter $\epsilon_{\text{clip}} = 0.2$. We use a KL regularization coefficient of 0.01. The policy loss uses token-level probability ratios with the group-relative completion advantage, masked over valid completion tokens. The learning rate uses a warmup phase followed by the configured decay schedule.

Rollout generation and policy optimization operate on the same evolving policy state. For DStreamer++, acceptance rewards are computed from the pre-update policy snapshot as described in Section B.2; this auxiliary adapter update does not change the task reward, base-policy objective, or rollout sampling configuration. The same RLVR protocol is used across autoregressive and speculative rollout methods so that differences in training throughput arise from rollout generation and speculative adaptation rather than changes to the underlying task-learning recipe.

D.3 BASELINES, DECODING CONFIGURATION, AND METRICS

For fixed-target serving, we compare against autoregressive decoding, EAGLE3, and DFlash. Autoregressive decoding uses the native vLLM execution path under the same serving configuration as DStreamer. For EAGLE3 and DFlash, we use their official released implementations and checkpoints without retraining baseline weights. In the primary matched-depth comparison, DStreamer, EAGLE3, and DFlash each use three speculative tokens per verification step. We also evaluate DFlash with its released 16-token default configuration, while using the three-token setting for the primary comparison because it provides the highest DFlash throughput in our sweep.

For online RLVR, our controlled runs compare autoregressive rollout, standard speculative decoding (Leviathan et al., 2023; Chen et al., 2023), DFlash, frozen DStreamer, and DStreamer++ under the common training protocol of Section D.2. The *Speculative Decoding* baseline, so labeled in Figures 5 and 10, uses a separate frozen autoregressive draft model of roughly 200M parameters,

initialized before RLVR and held fixed throughout policy training; it isolates speculative drift without online drafter adaptation. Where shown, results for online EAGLE3 training (Iso et al., 2026) and Efficient Rollout (Kim et al., 2026a) are taken from their respective papers rather than rerun in our system; we mark these results separately from measurements obtained under our implementation and hardware configuration.

Unless otherwise specified, generation uses temperature 1.0, top-$p = 1.0$, and minimum probability threshold 0.0. We report aggregate output throughput in generated tokens per second together with per-request output rate. For speculative decoding, the *mean accepted length* is $\tau = 1 + \alpha$, where α is the number of speculative tokens accepted in a verification step and the leading one is the target token emitted by the lossless verification procedure. For serving experiments, end-to-end generation speedup is computed from the complete `llm.generate` runtime defined in Section D.1.

For RLVR, end-to-end training speedup includes rollout generation, task-reward evaluation, base-policy optimization, acceptance-reward scoring, and the DStreamer++ adapter update. Final policy quality is evaluated on MATH500 (Lightman et al., 2023), AIME25 (Zhang & Math-AI, 2025), and AMC23 (Community, 2024) using the same decoding and evaluation configuration for every rollout method. Thus, throughput comparisons and accuracy comparisons differ only in the rollout mechanism and its associated speculative adaptation, while the underlying policy-training and evaluation protocols are held fixed.

E Additional DStreamer Analysis

E.1 Acceptance-Oriented RL from the Distillation Initialization

Figure 3 tracks DStreamer training from the consistency-distilled initialization across Qwen3-4B, Qwen3-8B, and Qwen3-14B. Acceptance-oriented RL consistently increases $r_{\mathrm{DS}} = 0.5\, r_{\mathrm{hard}} + 0.5\, r_{\mathrm{soft}}$ on both the training distribution and a fixed, disjoint validation set. The validation curves follow the same trend as training, indicating that the gain is not confined to the contexts used for adapter optimization. Together, these results show that the distilled adapter provides a useful initialization, while the subsequent path-level RL stage further improves alignment with the target verifier.

E.2 Reward Ablations

We ablate the acceptance-oriented reward on Qwen3-4B using the same consistency-distilled initialization and a fixed 10,000-step adapter-training budget. We compare training with the hard prefix reward only, the soft target-likelihood reward only, and their combined objective. To evaluate the resulting drafter independently of the training surrogate, we report the *mean accepted length* τ under the standard inference-time lossless verification procedure. The distilled initialization is included as the common pre-RL reference point.

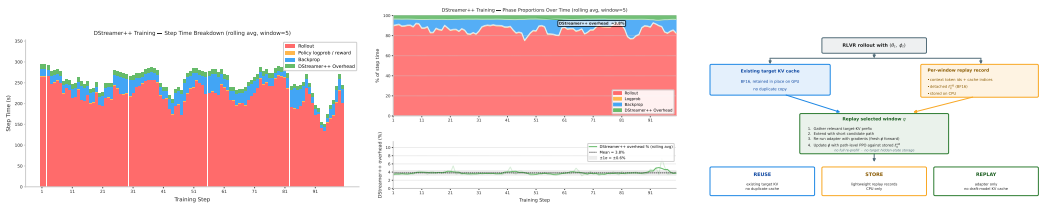
E.3 Speculative Drift and Online Adaptation

Figure 10 expands the acceptance analysis across model scale and context length. We report the mean accepted length τ throughout RLVR training for Qwen3-4B, Qwen3-8B, and Qwen3-14B. Across all configurations, DStreamer++ maintains or improves token progress as the target policy evolves, while frozen speculative predictors can lose acceptance over training. The relative degradation of the frozen methods depends on the operating point: a stronger initial drafter does not guarantee that its alignment will persist as θ changes. These results support the speculative-drift picture from Section 3: maintaining acceptance under an evolving verifier requires continued adaptation rather than relying solely on the initial drafter quality.

F Runtime and Memory Overhead

F.1 Online Adaptation Overhead

DStreamer++ adds two sources of work to each RLVR iteration: acceptance scoring for the retained speculative windows and the subsequent adapter replay update. As described in Section C.2, we



(a) Step-time decomposition. (b) Adaptation overhead. (c) State reuse.

Figure 11: **Runtime and state overhead of DStreamer++. (a)** Wall-clock decomposition of RLVR training steps. The additional acceptance-scoring and adapter-update work contributes approximately 3.8% of total step time in the profiled configuration (Qwen3-14B, 8k context, $8\times$H100). **(b)** DStreamer++ overhead remains stable over training despite variation in absolute step time. **(c)** DStreamer++ reuses the target generation KV cache, stores only lightweight per-window replay records, and replays only the adapter; no duplicate target KV cache or separate draft-model KV cache is required.

bound this cost by retaining at most N_{ret} windows per completion and processing them in large adapter minibatches, rather than training on every speculative window encountered during rollout.

We profile the step-time decomposition on Qwen3-14B at 8k context on $8\times$H100 under the RLVR configuration of Section D.2. Figure 11**(a)** reports the measured wall-clock decomposition over training. Rollout generation remains the dominant component, while acceptance scoring and adapter optimization together account for approximately 3.8% of total step time in the profiled configuration. Figure 11**(b)** shows the same overhead as a fraction of total iteration time. Despite variation in the absolute durations of rollout and policy optimization, the additional DStreamer++ cost remains nearly constant as a fraction of the training step. Thus, continual drafter adaptation preserves most of the rollout-time savings rather than introducing a comparable optimization cost.

F.2 Memory and State Retention

DStreamer++ similarly reuses state already produced during rollout instead of materializing a second copy for adapter training. Figure 11**(c)** summarizes the state flow. The target model's BF16 generation KV cache remains resident on GPU and is reused in place. Each retained speculative window stores only a lightweight CPU-side replay record containing context metadata, cache indices, and the detached rollout-time adapter logits used as the old-policy reference. No duplicate target KV tensors or target hidden states are retained.

During acceptance scoring and replay, the relevant prefix is gathered from the existing target KV cache and extended only by the short candidate continuation, avoiding a full context re-prefill. The rollout computation graph is discarded; a fresh adapter forward supplies gradients for the path-level PPO objective in Section B.3. Because the adapter attends directly to the target representations, DStreamer++ also maintains no separate draft-model KV cache. The online update therefore follows a simple state pattern: *reuse* the target KV cache, *store* lightweight per-window records, and *replay* only the adapter.

G Losslessness and On-Policy Correctness

G.1 Exactness of Lossless Verification

We formalize the losslessness property used by both DStreamer and DStreamer++. Let $x_{<t}$ denote the sequence already emitted by the decoder, and let $\pi_{\theta,T}$ denote the target sampling distribution under the same decoding transformation used by the autoregressive baseline, including sampling temperature T. Before target verification begins, the speculative adapter produces a candidate path

$$\mathbf{a} = (a_1, \ldots, a_w) \sim q_\phi(\mathbf{a} \mid x_{<t}).$$

The proposal distribution q_ϕ may be stochastic or deterministic, factorized across positions, conditioned on target-model representations, and updated throughout training. No assumption on its accuracy is required.

Given $\mathbf{a}$, verification proceeds from left to right (Leviathan et al., 2023; Chen et al., 2023). At speculative position i, provided all earlier proposals have been accepted, the target draws

$$z_i \sim \pi_{\theta,T}\left(\cdot \mid x_{<t} \oplus \mathbf{a}_{<i}\right). \tag{7}$$

If $a_i = z_i$, the proposal is accepted and verification continues. At the first mismatch, z_i is emitted and all remaining proposals are discarded. If all w proposals are accepted, an additional token

$$z_{w+1} \sim \pi_{\theta,T}\left(\cdot \mid x_{<t} \oplus \mathbf{a}_{1:w}\right) \tag{8}$$

is sampled directly from the target and emitted.

Proposition 1. For any proposal distribution q_ϕ generated before target verification, the procedure above produces exactly the same sequence distribution as ordinary autoregressive sampling from $\pi_{\theta,T}$.

Proof. Couple speculative verification to an autoregressive target sampler using the same target randomness. At the beginning of a verification window,

$$z_1 \sim \pi_{\theta,T}(\cdot \mid x_{<t}),$$

which is exactly the next token drawn by the coupled autoregressive sampler.

If $a_1 \neq z_1$, speculative verification emits z_1 directly and terminates the current speculative window. If $a_1 = z_1$, it emits a_1, which is identically the same token as z_1. Thus the first emitted token agrees pointwise with the corresponding autoregressive target sample.

More generally, suppose the first $i-1$ proposals have been accepted. Then

$$\mathbf{a}_{<i} = \mathbf{z}_{<i},$$

so

$$\pi_{\theta,T}\left(\cdot \mid x_{<t} \oplus \mathbf{a}_{<i}\right) = \pi_{\theta,T}\left(\cdot \mid x_{<t} \oplus \mathbf{z}_{<i}\right). \tag{9}$$

Consequently, z_i is drawn from exactly the conditional distribution used by ordinary autoregressive sampling after the sequence emitted so far. If $a_i = z_i$, verification emits $a_i = z_i$ and continues. Otherwise it emits z_i directly and terminates the speculative window. In either case, every token emitted before the window terminates is exactly the corresponding token of the coupled autoregressive trajectory.

If all w proposals are accepted, then $\mathbf{a}_{1:w} = \mathbf{z}_{1:w}$, and therefore the additional token z_{w+1} is sampled from the target distribution conditioned on exactly the same prefix as the autoregressive sampler. Applying the same argument inductively to every subsequent verification window establishes equality of the complete joint sequence distribution. □

The adapter therefore controls only how many tokens are advanced within a target verification step, not the distribution of the emitted sequence. Its initialization, factorized proposal architecture, acceptance-oriented training, and subsequent online updates may change acceptance efficiency but cannot change the generated distribution under the verification procedure above. Likewise, the top-K_{acc} compatibility reward and L_{local} context used during adapter training affect only the training signal for q_ϕ; they are not part of inference-time verification, which uses the target model's full decoding context.

G.2 On-Policy Rollouts Under an Evolving Target

We next apply Proposition 1 to online RLVR. At synchronous training iteration t, the target parameters θ_t remain fixed while the complete rollout batch is generated. The speculative adapter ϕ_t determines the proposal distribution and therefore the efficiency of generation, but Proposition 1 gives, for every prompt x,

$$y \sim \pi_{\theta_t,T}(\cdot \mid x), \tag{10}$$

independently of the proposal distribution induced by ϕ_t. The completion batch supplied to the task reward and base-policy GRPO update therefore has exactly the same distribution as a batch generated autoregressively from the current target policy.

After rollout collection, the task objective updates $\theta_t \to \theta_{t+1}$, while the acceptance objective separately updates $\phi_t \to \phi_{t+1}$. At the next iteration, the same losslessness argument applies to the new pair $(\theta_{t+1}, \phi_{t+1})$:

$$y' \sim \pi_{\theta_{t+1},T}(\cdot \mid x). \tag{11}$$

Thus, changing the speculative adapter affects the computational route used to obtain samples from the current policy, but not the policy from which those samples are drawn. DStreamer++ therefore preserves the on-policy rollout distribution required by synchronous RLVR at every training iteration.

For completeness, DStreamer++ evaluates its acceptance rewards under the same pre-update target snapshot θ_t that generated the rollout, as described in Section B.2. This ordering keeps the adapter's optimization signal aligned with the verifier it is tracking. It is not required for the losslessness result itself, which follows from Proposition 1 regardless of how ϕ is trained.

H Arithmetic-Intensity Analysis

H.1 Roofline Model and Assumptions

We use a first-order roofline model to characterize autoregressive decoding (Pope et al., 2023) in the configurations evaluated in this work. The analysis captures the dominant HBM traffic and compute cost of decoding rather than predicting kernel-level latency. In particular, it does not model cache residency, kernel-launch overhead, synchronization, or other implementation-specific effects.

For an operation with arithmetic intensity

$$\mathrm{AI} = \frac{\mathrm{FLOPs}}{\mathrm{HBM\ bytes}}, \tag{12}$$

the ideal roofline bound is

$$\mathrm{Perf} \le \min(C,\ \mathrm{BW} \cdot \mathrm{AI}), \tag{13}$$

where C is peak compute throughput and BW is peak HBM bandwidth. The corresponding ridge point is

$$\rho = \frac{C}{\mathrm{BW}}. \tag{14}$$

Arithmetic intensities below ρ lie on the bandwidth side of the ideal roofline, while intensities above ρ lie on the compute side.

Our experiments use NVIDIA H100 SXM GPUs with BF16 computation. NVIDIA reports 1,979 TFLOP/s of BF16 Tensor Core throughput with structured sparsity; because structured sparsity doubles the standard Tensor Core throughput, we use a nominal dense BF16 peak of $C \approx 989.5$ TFLOP/s. With HBM bandwidth $\mathrm{BW} = 3.35$ TB/s, this gives

$$\rho_{\mathrm{H100}} \approx \frac{989.5\ \mathrm{TFLOP/s}}{3.35\ \mathrm{TB/s}} \approx 295\ \mathrm{FLOP/byte}. \tag{15}$$

Throughout the derivation, B denotes the number of sequences decoded concurrently and L the context length. We retain the dominant HBM traffic from model weights and the attention KV cache and omit smaller traffic terms whose precise contribution depends on the implementation. The resulting model should therefore be interpreted as a first-order characterization of how batching and context length affect decoding arithmetic intensity.

H.2 Autoregressive Decoding

Consider one autoregressive decode step for a batch of B sequences, each with context length L. Let W denote the dominant model-weight bytes read from HBM during one target forward pass. Under ideal batching, this weight traffic is shared across the B sequences. In contrast, each sequence has its own attention KV state.

For a decoder with n_l layers, n_{kv} key/value heads, head dimension d_h, and KV element size s bytes, the stored KV state for a context of length L is

$$C_{\mathrm{KV}}(L) = 2 n_l n_{kv} d_h L s, \tag{16}$$

where the factor of two accounts for keys and values. For BF16 KV storage, $s = 2$ bytes.

Keeping only the dominant model-weight and past-KV traffic, the HBM traffic of one autoregressive decode step is approximated by

$$\mathrm{Bytes}_{\mathrm{AR}}(B, L) \approx W + B\, C_{\mathrm{KV}}(L). \tag{17}$$

The first term is shared across the batch, whereas the second scales with the number of independently decoded sequences.

Let $F_{\mathrm{tok}}(L)$ denote the FLOPs required to decode one token for one sequence at context length L, including the transformer computation and attention over the existing context. The corresponding batch-level compute is

$$\mathrm{FLOPs}_{\mathrm{AR}}(B, L) \approx B\, F_{\mathrm{tok}}(L). \tag{18}$$

Combining Eqs. 17 and 18 gives

$$\mathrm{AI}_{\mathrm{AR}}(B, L) \approx \frac{B\, F_{\mathrm{tok}}(L)}{W + B\, C_{\mathrm{KV}}(L)} = \frac{F_{\mathrm{tok}}(L)}{W/B + C_{\mathrm{KV}}(L)}. \tag{19}$$

Equation 19 makes the effect of batching explicit. Increasing B amortizes the shared model-weight traffic through the W/B term, but does not similarly amortize sequence-specific KV traffic. As the weight term shrinks, further gains in arithmetic intensity are increasingly governed by the ratio between per-token computation and per-sequence KV traffic.

H.3 Batching and the KV-Limited Ceiling

For fixed context length L, Eq. 19 increases monotonically with batch size:

$$\frac{\partial\, \mathrm{AI}_{\mathrm{AR}}}{\partial B} = \frac{F_{\mathrm{tok}}(L)\, W}{\left(W + B\, C_{\mathrm{KV}}(L)\right)^2} > 0. \tag{20}$$

Thus, within this first-order model, batching increases arithmetic intensity by amortizing the shared weight traffic.

The improvement has a finite large-batch limit:

$$\lim_{B \to \infty} \mathrm{AI}_{\mathrm{AR}}(B, L) = \frac{F_{\mathrm{tok}}(L)}{C_{\mathrm{KV}}(L)} \equiv \mathrm{AI}^{\infty}_{\mathrm{AR}}(L). \tag{21}$$

As B grows, the weight term is increasingly amortized, while the sequence-specific KV term remains. Batching therefore cannot raise arithmetic intensity beyond the ratio of per-token computation to per-sequence KV traffic at the same context length.

This ceiling does not imply that autoregressive decoding is bandwidth-bound at every operating point. Whether an operating point lies on the bandwidth or compute side of the roofline depends on its arithmetic intensity relative to the hardware ridge point. Figure 1a illustrates this comparison for the H100 autoregressive configurations considered in this work.

H.4 Implication for Multi-Token Verification

The analysis above motivates performing more useful token computation per target-model invocation. Autoregressive decoding advances one token per target decode step, whereas multi-token verification evaluates several candidate positions in a single target forward pass. At minimum, this amortizes target weight traffic across more query-token computation and, depending on the attention implementation, can also enable reuse of past-context KV state across query positions. Multi-token verification can therefore increase arithmetic intensity relative to one-token autoregressive decoding in regimes where these memory costs are significant.

This roofline argument provides a systems motivation for speculative execution, not a prediction of end-to-end speedup. Realized performance also depends on proposal cost, verification efficiency, acceptance, batching, and implementation overhead. We therefore measure serving and RLVR throughput directly in Section 4, rather than deriving speedup from the roofline model.