EntropyCache: Decoded Token Entropy Guided KV Caching for Diffusion Language Models

Minsoo Cheong [1] **Donghyun Son** [1] **Woosang Lim** [1] **Sungjoo Yoo** [1]

Abstract

Diffusion-based large language models (dLLMs) rely on bidirectional attention, which prevents lossless KV caching and requires a full forward pass at every denoising step. Existing approximate KV caching methods reduce this cost by selectively updating cached states, but their decision overhead scales with context length or model depth. We propose ENTROPYCACHE, a training-free KV caching method that uses the maximum entropy of newly decoded token distributions as a constant-cost signal for deciding *when* to recompute. Our design is grounded in two empirical observations: (1) decoded token entropy correlates with KV cache drift, providing a cheap proxy for cache staleness, and (2) feature volatility of decoded tokens persists for multiple steps after unmasking, motivating recomputation of the k most recently decoded tokens. The skip-or-recompute decision requires only $O(V)$ computation per step, independent of context length and model scale. Experiments on LLaDA-8B-Instruct and Dream-7B-Instruct show that ENTROPYCACHE achieves $15.2\times$–$26.4\times$ speedup on standard benchmarks and $22.4\times$–$24.1\times$ on chain-of-thought benchmarks against vanilla baselines, with competitive accuracy and decision overhead accounting for only 0.5% of inference time.

1. Introduction

Diffusion-based large language models (dLLMs) (Austin et al., 2021a; Sahoo et al., 2024; Nie et al., 2025; Ye et al., 2025) have emerged as a compelling alternative to autoregressive (AR) models (Touvron et al., 2023), generating tokens via iterative denoising over the full sequence. Combined with semi-autoregressive strategies such as blockwise (Wu et al., 2025b) or sliding-window decoding (Nguyen-Tri et al., 2025), they achieve throughput competitive with AR generation. However, these gains are constrained by a fundamental architectural limitation: dLLMs use non-causal (bidirectional) attention. Unlike AR models, where causal attention (Vaswani et al., 2017; Radford et al., 2018) makes KV caching (Pope et al., 2023) lossless, unmasking even a single token in a dLLM alters the representations of *all* positions. Vanilla dLLM inference therefore requires a full forward pass at every denoising step, with per-step cost proportional to the context length.

To reduce this cost, prior studies (Ma et al., 2025; Liu et al., 2025; Hu et al., 2025; Wu et al., 2025b) observe that the KV states of most tokens change only minimally between consecutive denoising steps, suggesting that selective reuse of cached states can closely approximate exact recomputation. Existing approximate KV caching strategies fall into two categories: *static* methods such as Fast-dLLM (Wu et al., 2025b) that apply a fixed reuse policy across all denoising steps, and *dynamic* methods that decide where or when to recompute based on intermediate states. Among dynamic methods, d^2Cache (Jiang et al., 2025) uses attention rollout to selectively update high-influence token positions (*where* to recompute), while Elastic-Cache (Nguyen-Tri et al., 2025) monitors per-layer attention-weight drift to detect cache staleness (*when* to recompute). However, their decision overhead scales with context length or model depth, limiting practical speedup.

We propose EntropyCache, which provides a cheaper signal for deciding *when* to recompute: the maximum entropy of the newly decoded token distributions. We show that this single scalar correlates with KV cache drift, enabling a skip-or-recompute decision with constant overhead independent of context length and model scale. The design is motivated by two empirical observations:

Observation 1: Unmasked token entropy predicts KV cache drift. We show that the maximum entropy (Shannon, 1948) of the token distributions obtained from the previous denoising step correlates with the magnitude of the resulting KV cache drift at the subsequent step. This single scalar therefore serves as a cheap yet effective proxy for cache staleness.

[1]Seoul National University, Seoul, South Korea. Correspondence to: Sungjoo Yoo <sungjoo.yoo@gmail.com>.

Observation 2: Feature volatility persists for multiple steps after unmasking. Tracking individual token trajectories reveals that some tokens remain unstable for several denoising steps after being unmasked, not just the single step assumed by prior dynamic methods (Ma et al., 2025; Jiang et al., 2025). This motivates recomputing the k most recently unmasked tokens rather than only those from the previous step.

Concretely, EntropyCache works as follows. After each unmasking step, it evaluates the maximum entropy of the newly unmasked token distributions. If this entropy exceeds a threshold τ, a full forward pass is triggered; otherwise, the method reuses cached KV pairs for most positions and recomputes KV pairs only for (a) the current mask tokens and (b) at most k recently unmasked tokens since the previous full recomputation.

In summary, we identify **maximum unmasked-token entropy as a lightweight proxy for KV cache staleness**—an $O(V)$ scalar independent of context length, model depth, and hidden dimension—and show that **feature volatility persists for multiple denoising steps after unmasking**, motivating recency-ranked local recomputation. Combining these, **EntropyCache** is a training-free caching method for dLLMs that delivers **15.2×–26.4×** speedup on standard benchmarks and **22.4×–24.1×** on chain-of-thought benchmarks over LLaDA-8B-Instruct and Dream-7B-Instruct, consistently outperforming prior methods in throughput while preserving accuracy.

2. Related work

2.1. Diffusion-based large language models

Unlike autoregressive (AR) models that generate tokens left-to-right with causal attention (Radford et al., 2018; Touvron et al., 2023), dLLMs denoise the entire sequence simultaneously using bidirectional attention. LLaDA (Nie et al., 2025) trains a masked diffusion model from scratch and achieves performance competitive with LLaMA3-8B. Dream (Ye et al., 2025) instead initializes from a pretrained AR checkpoint and introduces context-adaptive noise rescheduling. In both models, unmasking even a single token changes representations at all positions, making standard lossless KV caching inapplicable. Vanilla dLLM inference runs a full forward pass at every denoising step, so cost scales with the product of sequence length and step count. Semi-autoregressive strategies reduce this cost. Blockwise decoding (Wu et al., 2025b) denoises fixed-size blocks one at a time, while sliding-window decoding (Nguyen-Tri et al., 2025) advances a window over the sequence and keeps the number of denoised masked tokens constant at each step, better preserving diffusion-level parallelism. Orthogonally, confidence-based parallel decoding (Wu et al., 2025b) unmasks multiple tokens per step when predicted probabilities exceed a threshold. These strategies are complementary to KV caching methods.

2.2. Approximate KV caching for dLLMs

Prior approximate KV caching strategies for dLLMs fall into two categories: static methods that apply a fixed reuse policy across all denoising steps, and dynamic methods that determine where or when to refresh the cache based on intermediate states at each step.

Static caching strategy. Fast-dLLM (Wu et al., 2025b) freezes the KV pairs outside the current denoising block throughout its denoising steps. This is simple and effective but does not dynamically detect cache staleness. The entire cache is refreshed only at block boundaries.

Dynamic caching strategy. d^{2}Cache (Jiang et al., 2025) uses attention rollout to identify high-influence tokens and selectively updates their KV states at each step. Elastic-Cache (Nguyen-Tri et al., 2025) monitors attention-weight drift at the most-attended token and, upon detecting staleness, recomputes from a boundary layer onward while reusing shallow-layer caches. Both improve upon static caching in accuracy–throughput tradeoff, but their decision overhead scales with context length or model depth, limiting practical speedup.

3. Motivation

3.1. Decoded token entropy predicts KV cache drift

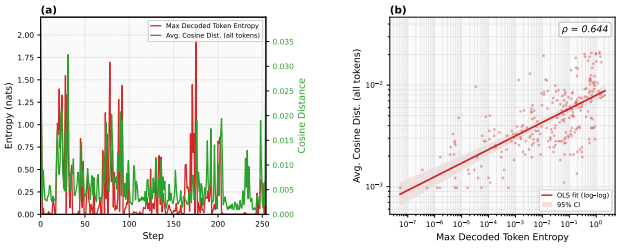

Figure 1. (a) Entropy and cosine distance metrics per decoding step experimented on single gsm8k sample using LLADA-8B-Instruct model. (b) Max decoded token entropy vs. avg. value vector cosine distance, plotted on log–log axes.

Entropy is a widely-used measure of predictive uncertainty in neural networks (Shannon, 1948; Gal & Ghahramani, 2016; Xu et al., 2025). We examine whether this signal can also serve as a predictor of KV cache drift in dLLMs, by measuring the relationship between prediction uncertainty at each denoising step and the resulting shift in the model's internal representations. Our central claim is that the **maximum entropy of the newly decoded token distributions** at each step serves as an effective, low-cost predictor of

how much the KV cache will drift at the subsequent step, providing an effective proxy for selective recomputation.

We run LLaDA-8B-Instruct on a single GSM8K sample with 256 denoising steps, recording two quantities at every step: (i) the maximum entropy among the newly decoded tokens and (ii) the average cosine distance between value vectors at consecutive denoising steps(step t and t+1) across all token positions, which we take as our ground-truth measure of KV cache drift. We present this single-sample analysis here to build intuition; an extended study aggregating 64 samples per benchmark across four datasets, which confirms and strengthens these findings, is provided in Appendix B.

Temporal co-occurrence of entropy spikes and cache drift. Figure 1(a) plots these two signals over the course of generation. The max decoded token entropy tracks the cosine distance spikes remarkably well: steps at which a high-entropy token is committed coincide with pronounced jumps in KV drift, whereas low-entropy decodings are followed by minimal KV drift. This suggests that cache staleness is driven specifically by the surprise introduced at the moment of token commitment.

Quantitative correlation. To move beyond visual inspection, we scatter-plot the max decoded token entropy against the average value-vector cosine distance for all 256 steps on log–log axes (Figure 1(b)). The two quantities exhibit a clear positive correlation in log–log space with a Spearman rank correlation of $\rho = 0.644$. Extending this single-sample analysis, we also present that this trend holds consistently across tasks and samples (Appendix B).

3.2. Feature volatility persists beyond the decoding step

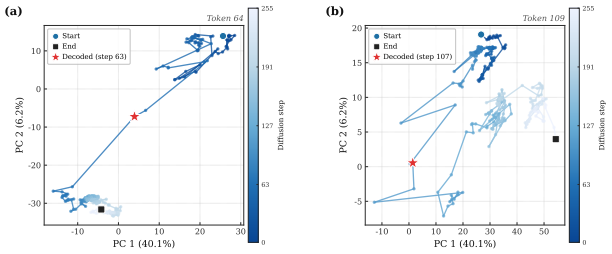

Figure 2. (a)–(b) PCA projections (PC 1 vs. PC 2) of the last-layer value vectors for two mask tokens over 256 denoising steps in LLADA-8B-Instruct on a single GSM8K sample. Color encodes step progression (dark→light); the red star marks the decoding step.

The previous section established *when* the KV cache drifts; we now examine *which tokens* are most affected and for *how long*. To this end, we track individual token trajectories in the value-vector space across all denoising steps.

PCA trajectories of individual tokens. Following d^{2}Cache (Jiang et al., 2025), we apply PCA to the last-layer value vectors of all mask-token positions across 256 denoising steps and visualize two tokens in Figure 2(a)–(b). Token 64 experiences sharp shift in trajectory directly after being decoded but converges rapidly in the following steps. By contrast, Token 109 continues to drift through the principal-component space for many steps after being decoded, exhibiting sustained volatility before eventual stabilization. These two cases illustrate a spectrum: while some tokens stabilize almost immediately upon unmasking, others remain volatile well beyond the single decode step.

4. Method

Motivated by the observations in Section 3, we propose EntropyCache, a KV caching method for dLLMs that ① uses a single entropy scalar to decide *when* to trigger full recomputation and ② recomputes the k most recently decoded tokens to account for multi-step feature volatility. Figure 3 illustrates the three phases executed at each denoising step; we describe each component below.[1]

4.1. Entropy-based skipping

Section 3.1 showed that the maximum entropy of newly decoded tokens is an effective predictor of KV cache drift. We exploit this by using it as a binary trigger for full recomputation.

At each step t, after decoding new tokens $\mathcal{D}^{t+1}$ from the model logits over all vocab indexes $j \in V$, we compute the maximum entropy over their predicted distributions:

$$E^{t+1} = \max_{i \in \mathcal{D}^{t+1}} \Big(-\sum_{j} p_{ij} \log p_{ij} \Big). \tag{1}$$

If $E^{t+1} > \tau$, the decoded tokens originate from uncertain distributions and a large cache drift is expected; therefore, we trigger a full forward pass at step $t+1$ to refresh the entire KV cache. If $E^{t+1} \leq \tau$, we reuse the existing KV cache, executing only a partial forward pass over the selected subset of tokens. This decision requires a single $O(V)$ computation per step, where V is the vocabulary size. Note that this overhead is entirely independent of context length, model depth, and hidden dimension.

4.2. Recent token recomputation

When the entropy-based decision skips full recomputation, we find that updating a small subset of recently decoded tokens alongside the current mask tokens is sufficient to retain model accuracy. Section 3.2 demonstrates that feature volatility persists for multiple steps after a token is

[1]Full pseudocode with layer-level details is provided in Algorithm 1 (Appendix A).

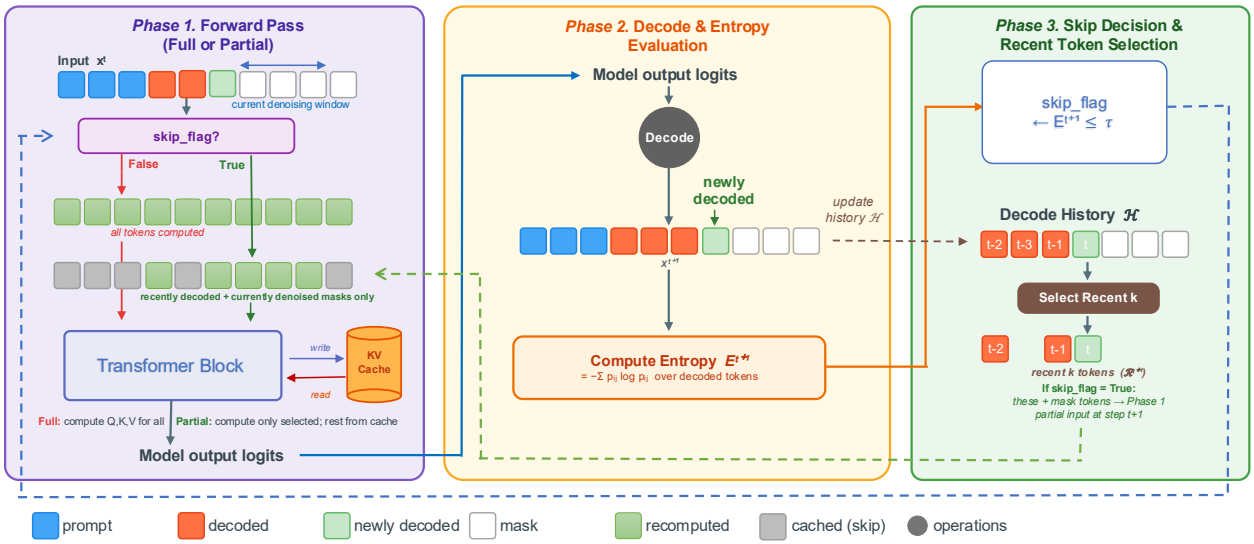

Figure 3. Overview of EntropyCache at a single denoising step t. **Phase 1**: a full or partial forward pass is executed depending on the recompute flag; in the partial case, only mask tokens and recently decoded tokens are recomputed while the rest are read from cache. **Phase 2**: new tokens are decoded from the model logits and the maximum entropy E^{t+1} of the decoded distributions is computed. **Phase 3**: the entropy is compared against threshold τ to set the recompute flag for the next step, and the k most recently decoded tokens $\mathcal{R}^{t+1}$ are selected for partial recomputation.

unmasked, not just the single step assumed by prior methods (Jiang et al., 2025; Ma et al., 2025). Accordingly, we maintain a decode history $\mathcal{H} \in \mathbb{R}^L$ and at each step select a fixed budget of k recently decoded tokens for recomputation.

Formally, each entry of the history vector records when a position was unmasked:

$$\mathcal{H}_i = \begin{cases} n & \text{if position } i \text{ was decoded at step } n, \\ -\infty & \text{otherwise.} \end{cases} \tag{2}$$

We then select the k positions with the largest history values:

$$\mathcal{K} = \underset{\mathcal{S} \subset \{1,\ldots,|\mathcal{H}|\},\ |\mathcal{S}|=k}{\arg\max} \sum_{j \in \mathcal{S}} \mathcal{H}_j, \tag{3}$$

and define a recency threshold of $\tau_{\mathcal{R}} = \max\big(\min_{j \in \mathcal{K}} \mathcal{H}_j,\ t - \Delta t_{recompute}\big)$ where $\Delta t_{recompute}$ is the number of steps elapsed since the last full recomputation and t is the current time step. Since a full forward pass resets all KV states to their exact values, subsequent drift is driven primarily by tokens decoded after that point; the threshold $\tau_{\mathcal{R}}$ therefore restricts the recomputation budget to these positions, where volatility is concentrated. The selected recent tokens mask is then $\mathcal{R}^{t+1} = \mathbb{I}(\mathcal{H}_i \geq \tau_{\mathcal{R}})$. In the partial forward pass, only the current mask tokens $\mathcal{M}^{t+1}$ and the selected recent tokens have their K, V recomputed; all other positions are reused from the cached KV states.

5. Experiment

5.1. Experiment setup

All experiments are conducted on NVIDIA RTX 3090 GPUs. To ensure a fair comparison of KV cache mechanisms, we re-implement all methods within a unified codebase to share the same decoding logic. We evaluate LLaDA-8B-Instruct and Dream-7B-Instruct using sliding window decoding (window size = 32), except for Fast-dLLM, which strictly requires blockwise decoding (block size = 32). All KV cache methods use confidence-based parallel decoding with a threshold of 0.9. For dynamic KV cache methods, we focus on d^{2}Cache and Elastic-Cache as baselines, as prior work (Jiang et al., 2025; Nguyen-Tri et al., 2025) has shown them to be more effective than other previous approaches (Ma et al., 2025; Liu et al., 2025) under comparable settings. We use the optimal parameters reported in their respective papers: rollout$_p = 0.2$ for d^{2}Cache and a cosine similarity threshold of 0.9 for Elastic-Cache. For EntropyCache, we set $\tau = 1.5$ and $k = 64$.

We evaluate on two benchmark suites: (1) standard benchmarks including GSM8K (Cobbe et al., 2021) 4-shot, MATH500 (Hendrycks et al., 2021) 4-shot, MBPP (Austin et al., 2021b) 3-shot, and HumanEval (Chen et al., 2021) zero-shot; and (2) chain-of-thought benchmarks including GSM8K 8-shot, MMLU-Pro (Wang et al., 2024), BBH 3-shot (Suzgun et al., 2023), and GPQA zero-shot (Rein et al.,

2024).

5.2. Main results

Standard benchmarks. Table 1 compares all methods on four standard benchmarks. EntropyCache achieves the best accuracy and throughput among all caching methods: on LLaDA-Instruct it attains 52.28% average accuracy at 15.2× speedup, and on Dream-Instruct it reaches 56.21% at 26.4× speedup. No other caching method preserves baseline-level average accuracy while achieving comparable throughput.

The advantage is clearest on HumanEval, where all other methods degrade accuracy on at least one model. EntropyCache is the only method that achieves lossless performance compared to baseline on both models, while still delivering 8.9×–30.7× speedup.

Chain-of-thought benchmarks. Table 2 evaluates on longer-context CoT tasks. EntropyCache achieves 22.4× and 24.1× average speedup on LLaDA-Instruct and Dream-Instruct, roughly 1.4× the next-best method (d^{2}Cache). On BBH 3-shot the gap widens to 1.7×, with EntropyCache reaching 107.3× speedup on LLaDA and 96.4× on Dream, as long generations with many low-entropy steps maximize cache reuse. Accuracy is highest among caching methods on Dream-Instruct (50.97%) and within 0.5pp of d^{2}Cache on LLaDA-Instruct (48.57% vs. 49.02%).

5.3. Ablation study

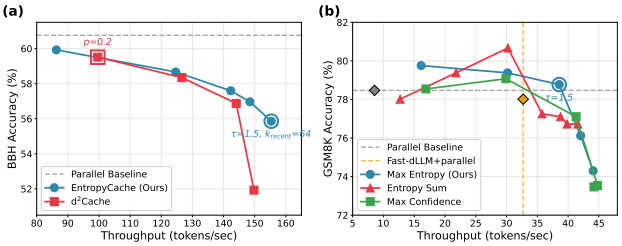

Figure 4. **(a)** Speed-accuracy tradeoff comparison on BBH between EntropyCache and d^{2}Cache (Dream-7B-Instruct). Hollow markers denote default configurations. **(b)** Accuracy–throughput tradeoff on GSM8K (LLaDA-8B-Instruct) for three candidate skipping metrics across varying thresholds τ.

Speed-accuracy tradeoff comparison on BBH. We further analyze EntropyCache's behavior on long-generation tasks by tracing out the speed-accuracy tradeoff on BBH with Dream-7B-Instruct. We sweep τ for EntropyCache and $\mathrm{rollout}_p$ for d^{2}Cache (Figure 4(a)); the default configurations of the two methods land at different points on the tradeoff curve, which makes a single-configuration comparison misleading. Across the full sweep, EntropyCache consistently outperforms d^{2}Cache: at matched throughput, EntropyCache achieves higher accuracy. EntropyCache also maintains stable accuracy at high throughput, whereas d^{2}Cache degrades sharply as $\mathrm{rollout}_p$ decreases.

Choice of thresholding variable We compare three candidate metrics for adaptive skipping: (1) max entropy, which takes the maximum entropy of decoded tokens; (2) entropy sum, which uses the summation of entropy over all decoded tokens; and (3) max confidence, which uses the maximum token-level probability across decoded tokens. For each metric, we plot the accuracy–throughput tradeoff on GSM8K (Figure 4(b)). Max entropy produces the most favorable tradeoff among the three: at $\tau = 1.5$ it achieves approximately 78.8% accuracy at ∼38.6 tok/s, surpassing both the parallel baseline's accuracy and Fast-dLLM DualCache's throughput. Entropy sum peaks sharply at $\tau = 1.5$ but degrades beyond it, making threshold selection sensitive to miscalibration. Max confidence is dominated by max entropy across most of the throughput range. Based on these results, we adopt max entropy as the default thresholding variable for all experiments.

Additional ablations. We provide additional ablation studies in Appendix C, including the effect of the sliding-window size and per-benchmark sweeps over τ and k_{recent} on the four standard benchmarks.

5.4. Overhead analysis

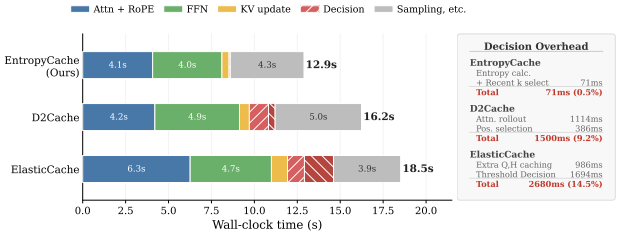

Figure 5. Wall-clock time breakdown on a single GSM8K sample (LLaDA-Inst, $w = 32$, generation length $= 256$). Left: stacked bar chart of inference components. Right: decision overhead decomposition per method. EntropyCache's decision cost is 71 ms (0.5%), roughly 20× and 38× cheaper than d^{2}Cache and Elastic-Cache respectively.

Dynamic KV caching methods introduce extra computation at each step to decide where or when to refresh the cache. A key practical advantage of EntropyCache is that its decision overhead is decoupled from both context length and model scale. As summarized in Table 3, d^{2}Cache and Elastic-Cache both scale with context length L and model depth ℓ, while EntropyCache requires only an $O(V)$ entropy evaluation over the vocabulary distribution—independent of L, ℓ, and hidden dimension d. Wall-clock profiling confirms this gap: EntropyCache's decision logic accounts for only 0.5% of inference time, versus 9.2% for d^{2}Cache and 14.5%

Table 1. Comprehensive dLLM KV cache method comparison. Speedup (×) relative to Baseline. Underlined scores meet or exceed Baseline accuracy. Acc.: Accuracy (%). T-put: Throughput (tok/s). Superscripts indicate max generation length.

Method	GSM8K^{256}		MATH500^{512}		MBPP^{256}		HumanEval^{512}		Average	
	Acc.	T-put	Acc.	T-put	Acc.	T-put	Acc.	T-put	Acc.	T-put
LLADA-8B-Instruct										
Baseline	77.86	2.28	42.40	2.47	42.20	2.89	46.34	4.32	52.20	2.99
Baseline+Parallel	78.47	8.56 (3.8×)	42.40	9.50 (3.9×)	42.40	16.37 (5.7×)	48.17	15.40 (3.6×)	52.86	12.46 (4.2×)
Fast-DLLM (Dual)	78.01	32.54 (14.3×)	40.00	32.81 (13.3×)	39.20	45.58 (15.8×)	44.51	37.71 (8.7×)	50.43	37.16 (12.4×)
Elastic-Cache	79.15	23.49 (10.3×)	43.80	24.43 (9.9×)	40.60	32.24 (11.2×)	44.51	27.73 (6.4×)	52.02	26.97 (9.0×)
d^{2}Cache	78.01	29.96 (13.2×)	43.60	32.18 (13.1×)	40.80	52.77 (18.3×)	41.46	41.36 (9.6×)	50.97	39.06 (13.1×)
EntropyCache (Ours)	78.77	38.49 (16.9×)	42.80	39.64 (16.1×)	38.80	64.91 (22.5×)	48.78	38.41 (8.9×)	**52.28**	**45.36 (15.2×)**
Dream-v0-Instruct-7B										
Baseline	74.30	2.60	44.60	2.60	50.60	3.31	53.05	3.87	55.64	3.10
Baseline+Parallel	74.91	12.14 (4.7×)	44.00	15.86 (6.1×)	48.60	29.66 (9.0×)	53.05	38.74 (10.0×)	55.14	24.10 (7.8×)
Fast-DLLM (Dual)	72.25	42.17 (16.2×)	40.20	43.52 (16.7×)	51.40	69.36 (21.0×)	55.49	71.71 (18.5×)	54.83	56.69 (18.3×)
Elastic-Cache	69.60	32.15 (12.4×)	38.80	41.90 (16.1×)	45.80	54.84 (16.6×)	50.61	72.30 (18.7×)	51.20	50.30 (16.2×)
d^{2}Cache	75.44	41.68 (16.0×)	40.60	55.80 (21.4×)	48.40	87.63 (26.5×)	53.66	110.83 (28.6×)	54.52	73.98 (23.9×)
EntropyCache (Ours)	74.30	48.32 (18.6×)	43.20	62.48 (24.0×)	48.80	97.79 (29.6×)	58.54	119.02 (30.7×)	**56.21**	**81.90 (26.4×)**

Table 2. Chain-of-Thought (CoT) benchmark comparison. Speedup (×) relative to Baseline. Acc.: Accuracy (%). T-put: Throughput (tok/s).

Method	GSM8K 8-shot		MMLU-Pro		BBH 3-shot		GPQA Zero-shot		Average	
	Acc.	T-put	Acc.	T-put	Acc.	T-put	Acc.	T-put	Acc.	T-put
LLADA-8B-Instruct										
Baseline	81.12	2.07	37.60	1.68	59.58	1.35	27.68	5.76	51.49	2.71
Baseline+Parallel	81.43	12.44 (6.0×)	37.57	4.77 (2.8×)	57.90	21.45 (15.9×)	27.68	11.22 (1.9×)	51.15	12.47 (4.6×)
Fast-DLLM (Dual)	80.44	41.68 (20.1×)	37.15	21.56 (12.8×)	52.48	36.89 (27.3×)	24.78	25.34 (4.4×)	48.71	31.37 (11.6×)
Elastic-Cache	81.20	32.89 (15.9×)	36.69	14.78 (8.8×)	52.57	42.39 (31.4×)	24.78	18.08 (3.1×)	48.81	27.04 (10.0×)
d^{2}Cache	81.35	47.03 (22.7×)	36.95	16.80 (10.0×)	55.68	83.94 (62.2×)	22.10	23.22 (4.0×)	**49.02**	42.75 (15.8×)
EntropyCache (Ours)	81.20	62.31 (30.1×)	36.06	15.21 (9.1×)	52.48	144.90 (107.3×)	24.55	20.98 (3.6×)	48.57	**60.85** (22.4×)
Dream-v0-Instruct-7B										
Baseline	78.01	2.51	47.12	2.01	65.69	1.61	27.01	6.14	54.46	3.07
Baseline+Parallel	77.41	13.95 (5.6×)	46.64	10.93 (5.4×)	60.76	25.42 (15.8×)	24.78	19.12 (3.1×)	52.40	17.36 (5.7×)
Fast-DLLM (Dual)	74.22	55.50 (22.1×)	41.86	35.01 (17.4×)	58.09	40.96 (25.4×)	25.22	34.69 (5.6×)	49.85	41.54 (13.5×)
Elastic-Cache	69.37	36.95 (14.7×)	42.50	26.50 (13.2×)	31.52	52.61 (32.7×)	27.68	26.90 (4.4×)	42.77	35.74 (11.6×)
d^{2}Cache	73.69	49.59 (19.8×)	43.33	40.95 (20.4×)	59.51	99.60 (61.8×)	27.01	35.23 (5.7×)	50.89	56.34 (18.4×)
EntropyCache (Ours)	75.74	60.01 (23.9×)	45.49	42.14 (21.0×)	55.86	155.30 (96.4×)	26.79	38.73 (6.3×)	**50.97**	**74.05 (24.1×)**

for Elastic-Cache (Figure 5).

Table 3. Complexity comparison of dynamic KV cache methods. EntropyCache achieves constant overhead, decoupling decision overhead from both context length and model scale.
Notation: ℓ = number of layers; L = context length; H = number of attention heads; d = hidden dimension; V = vocabulary size. d^{2}Cache and Elastic-Cache overhead grows with L and model scale (ℓ, d), respectively, whereas EntropyCache remains constant regardless of both.

Method	Computation	Memory	Bottleneck
d^{2}Cache (Jiang et al., 2025)	$\mathcal{O}(\ell \cdot L \cdot H)$	$\mathcal{O}(H \cdot L^2)$	Quadratic in L
Elastic-Cache (Nguyen-Tri et al., 2025)	$\mathcal{O}(\ell \cdot L \cdot d)$	$\mathcal{O}(\ell \cdot L \cdot d)$	Scales with ℓ, L, d
EntropyCache (Ours)	$\mathcal{O}(V)$	$\mathcal{O}(k)$	**Constant**

6. Conclusion

We presented EntropyCache, a KV caching method for diffusion-based large language models that provides a cheap proxy for deciding *when* recomputation is necessary. By exploiting the empirical correlation between decoded token entropy and KV cache drift, EntropyCache uses a constant $O(V)$ entropy check to decide whether to reuse cached states or trigger a full forward pass, while recomputing the k most recently decoded tokens to account for multi-step feature volatility. EntropyCache achieves 15.2× to 26.4× average speedup over vanilla dLLM inference on standard benchmarks with competitive accuracy, and maintains consistent gains on chain-of-thought benchmarks. Additional analyses are provided in the appendix.

References

Austin, J., Johnson, D. D., Ho, J., Tarlow, D., and Van Den Berg, R. Structured denoising diffusion models in discrete state-spaces. *Advances in Neural Information Processing Systems*, 34:17981–17993, 2021a.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021b.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Gal, Y. and Ghahramani, Z. Dropout as a Bayesian approximation: Representing model uncertainty in deep learning. In *International Conference on Machine Learning*, pp. 1050–1059. PMLR, 2016.

Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., and Steinhardt, J. Measuring mathematical problem solving with the MATH dataset. *arXiv preprint arXiv:2103.03874*, 2021.

Hu, Z., Meng, J., Akhauri, Y., Abdelfattah, M. S., Seo, J.-s., Zhang, Z., and Gupta, U. Accelerating diffusion language model inference via efficient kv caching and guided diffusion. *arXiv preprint arXiv:2505.21467*, 2025.

Jiang, Y., Cai, Y., Luo, X., Fu, J., Wang, J., Liu, C., and Yang, X. d^2cache: Accelerating diffusion-based llms via dual adaptive caching. *arXiv preprint arXiv:2509.23094*, 2025.

Liu, Z., Yang, Y., Zhang, Y., Chen, J., Zou, C., Wei, Q., Wang, S., and Zhang, L. dllm-cache: Accelerating diffusion large language models with adaptive caching. *arXiv preprint arXiv:2506.06295*, 2025.

Ma, X., Yu, R., Fang, G., and Wang, X. dkv-cache: The cache for diffusion language models. *arXiv preprint arXiv:2505.15781*, 2025.

Nguyen-Tri, Q., Ranjan, M., and Shen, Z. Attention is all you need for kv cache in diffusion llms. *arXiv preprint arXiv:2510.14973*, 2025.

Nie, S., Zhu, F., You, Z., Zhang, X., Ou, J., Hu, J., Zhou, J., Lin, Y., Wen, J.-R., and Li, C. Large language diffusion models. *arXiv preprint arXiv:2502.09992*, 2025.

Pope, R., Douglas, S., Chowdhery, A., Devlin, J., Bradbury, J., Heek, J., Xiao, K., Agrawal, S., and Dean, J. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5:606–624, 2023.

Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. Improving language understanding by generative pre-training. 2018.

Rein, D., Hou, B. L., Stickland, A. C., Petty, J., Pang, R. Y., Dirani, J., Michael, J., and Bowman, S. R. GPQA: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*, 2024.

Sahoo, S. S., Arriola, M., Gokaslan, A., Marroquin, E. M., Rush, A. M., Schiff, Y., Chiu, J. T., and Kuleshov, V. Simple and effective masked diffusion language models. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=L4uaAR4ArM.

Shannon, C. E. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

Suzgun, M., Scales, N., Schärli, N., Gehrmann, S., Tay, Y., Chung, H. W., Chowdhery, A., Le, Q. V., Chi, E. H., Zhou, D., and Wei, J. Challenging BIG-Bench tasks and whether chain-of-thought can solve them. In *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 13003–13051, 2023.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 2017.

Wang, Y., Ma, X., Zhang, G., Ni, Y., Chandra, A., Guo, S., Ren, W., Arulraj, A., He, X., Jiang, Z., et al. MMLU-Pro: A more robust and challenging multi-task language understanding benchmark. *Advances in Neural Information Processing Systems*, 37:95266–95290, 2024.

Wu, C., Zhang, H., Xue, S., Diao, S., Fu, Y., Liu, Z., Molchanov, P., Luo, P., Han, S., and Xie, E. Fast-dllm v2: Efficient block-diffusion llm. *arXiv preprint arXiv:2509.26328*, 2025a.

Wu, C., Zhang, H., Xue, S., Liu, Z., Diao, S., Zhu, L., Luo, P., Han, S., and Xie, E. Fast-dllm: Training-free acceleration of diffusion llm by enabling kv cache and parallel decoding. *arXiv preprint arXiv:2505.22618*, 2025b.

Xu, G., Xu, W., Zhao, J., and Ma, K. Gift: Guided importance-aware fine-tuning for diffusion language models. *arXiv preprint arXiv:2509.20863*, 2025.

Ye, J., Xie, Z., Zheng, L., Gao, J., Wu, Z., Jiang, X., Li, Z., and Kong, L. Dream 7b: Diffusion large language models. *arXiv preprint arXiv:2508.15487*, 2025.

A. Full EntropyCache algorithm

Algorithm 1 EntropyCache algorithm

Input: Prompt $\mathbf{x}_{\text{prompt}}$, generation length N, window size w, entropy threshold τ, recent-token window k
Initialize: $\mathbf{x}^0 \leftarrow \{\mathbf{x}_{\text{prompt}}; [\text{MASK}], \ldots, [\text{MASK}]\}$; $P \leftarrow \text{length}(\mathbf{x}_{\text{prompt}})$
$t \leftarrow 1$; $\mathcal{D}^1 \leftarrow \{1, \ldots, P\}$; $\mathcal{M}^1 \leftarrow \{P+1, \ldots, P+N\}$; $\mathcal{R}^1 \leftarrow \emptyset$
$\mathcal{A}^{0,l} \leftarrow \emptyset$ for all layers l; $\mathcal{D}_{\text{prev}} \leftarrow \emptyset$; skip_flag $\leftarrow$ False
$\mathcal{H} \leftarrow \{-\infty\}^N$
while $\mathcal{M}^t \neq \emptyset$ **do**
  if $t = 1$ **or** skip_flag $=$ False **then**
    // Prefill
    for $l = 1, \ldots, L$ **do**
      $\mathbf{Q}^{t,l}_{[\mathcal{I}]}, \mathbf{K}^{t,l}_{[\mathcal{I}]}, \mathbf{V}^{t,l}_{[\mathcal{I}]} \leftarrow \text{W}(\mathbf{H}^{t,l-1}_{[\mathcal{I}]})$
      $\mathbf{H}^{t,l}_{[\mathcal{I}]}, \mathbf{S}_{[\mathcal{I}]} \leftarrow \text{TransformerBlock}(\mathbf{Q}^{t,l}_{[\mathcal{I}]}, \mathbf{K}^{t,l}_{[\mathcal{I}]}, \mathbf{V}^{t,l}_{[\mathcal{I}]})$
    end for
  else
    for $l = 1, \ldots, L$ **do**
      $\mathbf{H}^{t,0}_{[\mathcal{M}^t]} \leftarrow \mathbf{x}^t_{[\mathcal{M}^t]}$
      $\mathbf{Q}^{t,l}_{[\mathcal{M}^t \cup \mathcal{R}^t]}, \mathbf{K}^{t,l}_{[\mathcal{M}^t \cup \mathcal{R}^t]}, \mathbf{V}^{t,l}_{[\mathcal{M}^t \cup \mathcal{R}^t]} \leftarrow \text{W}(\mathbf{H}^{t,l-1}_{[\mathcal{M}^t \cup \mathcal{R}^t]})$
      $\mathbf{H}^{t,l}_{[\mathcal{M}^t \cup \mathcal{R}^t]}, \mathbf{S}^{t,l}_{[\mathcal{M}^t \cup \mathcal{R}^t]} \leftarrow \text{TransformerBlock}(\mathbf{Q}^{t,l}_{[\mathcal{M}^t \cup \mathcal{R}^t]}, \mathbf{K}^{t,l}_{[\mathcal{I}]}, \mathbf{V}^{t,l}_{[\mathcal{I}]})$
    end for
  end if
  $\mathbf{x}^{t+1}, \mathcal{D}^{t+1} \leftarrow \text{decode}(\mathbf{x}^t, \mathcal{M}^t)$ *// Decode new tokens*
  $\mathcal{H}[\mathcal{D}^{t+1}] \leftarrow t$ *// Record decode history*
  // Select recent tokens
  $\mathcal{K} \leftarrow \underset{S \subset \{1,\ldots,|\mathcal{H}|\},\, |S|=k}{\arg\max} \sum_{j \in S} \mathcal{H}_j$
  $\tau_{\mathcal{R}} \leftarrow \max(\min_{j \in \mathcal{K}} \mathcal{H}_j,\ t - \Delta t_{\text{recompute}})$
  $\mathcal{R}^{t+1} \leftarrow \{i \mid \mathcal{H}_i \geq \tau_{\mathcal{R}}\}$
  $E^{t+1} \leftarrow \max_{i \in \mathcal{D}^{t+1}} e(p_i)$ where $e(p_i) = -\sum_j p_{ij} \log p_{ij}$ *// Compute entropy for next step*
  if $E^{t+1} \leq \tau$ **then**
    skip_flag $\leftarrow$ True *// Set flag for next iteration*
    $\Delta t_{\text{recompute}} \leftarrow \Delta t_{\text{recompute}} + 1$
  else
    skip_flag $\leftarrow$ False
    $\Delta t_{\text{recompute}} \leftarrow 0$
  end if
  $\mathcal{M}^{t+1} \leftarrow \mathcal{M}^t \setminus \mathcal{D}^{t+1}$ *// Update mask positions*
  $t \leftarrow t + 1$
end while
return $\mathbf{x}^{t-1}$

B. Extended empirical analysis on entropy and KV cache drift

The correlation between decoded token entropy and KV cache drift presented in Section 3 was derived from a single generation trajectory. To validate that this relationship holds broadly, we extend the analysis across all four standard benchmarks (GSM8K, MATH500, MBPP, HumanEval), generating 64 samples per dataset using LLaDA-8B-Instruct under the same sliding-window decoding configuration used in our main experiments. For each denoising step of every sample, we record the maximum decoded token entropy and the average value-vector cosine distance across all token positions, yielding between 12K and 28K step-level observations per dataset.

Per-dataset scatter analysis. Figure 6 plots the per-step entropy–drift pairs on log–log axes for each benchmark. Across all four datasets, the positive monotonic trend observed in Section 3.1 is clearly reproduced: higher decoded entropy at step t is consistently associated with larger KV cache drift at step $t{+}1$. The aggregate Spearman correlations on non-EOS steps are $\rho = 0.616$ (GSM8K), $\rho = 0.558$ (MATH500), $\rho = 0.606$ (MBPP), and $\rho = 0.377$ (HumanEval), all with $p \approx 0$.

EOS token outliers. A notable cluster of outlier points appears in the low-entropy, high-drift region of each scatter plot (shown in gray). These correspond to denoising steps in which the model has already generated its complete response and is filling the remaining positions with EOS tokens. Because EOS tokens are produced with near-zero entropy yet cause a non-trivial representational shift as the sequence transitions from active generation to padding, they decouple from the entropy–drift trend that governs content tokens. Crucially, these outliers have no practical impact on EntropyCache: since their entropy falls well below any reasonable threshold τ, they are automatically classified as skip steps, and the method correctly reuses the cached KV states. We therefore report all Spearman correlations on non-EOS steps to reflect the regime in which the entropy trigger is operative.

Per-sample Spearman distribution. To assess the consistency of the entropy–drift relationship at the individual sample level, Figure 7 shows the distribution of per-sample Spearman ρ values (non-EOS, log–log) across the 64 samples for each dataset. GSM8K exhibits the tightest distribution (mean $\rho = 0.62$, IQR ≈ 0.59–0.68), while MATH500 (mean $\rho = 0.57$) and MBPP (mean $\rho = 0.59$) show similarly concentrated positive correlations. HumanEval yields a lower and more dispersed distribution (mean $\rho = 0.40$, IQR ≈ 0.33–0.50), indicating that the entropy signal is a weaker predictor of raw KV drift for code generation sequences.

Weak correlation does not imply weak performance. Interestingly, the weaker entropy–drift correlation on HumanEval does not translate into weaker downstream performance. As shown in Table 1, EntropyCache achieves its highest accuracy improvement over the baseline on HumanEval (+2.44%p for LLaDA, +5.49%p for Dream), despite the comparatively modest ρ. We note that Spearman correlation measures *synchronous* alignment between entropy and drift at each step: a recomputation triggered 1–2 steps after a drift event would appear as a mismatch in this metric, yet may still refresh the KV cache before it affects downstream decoding. Point-wise correlation may therefore underestimate the practical utility of the entropy signal as a recomputation trigger, and we leave a deeper mechanistic analysis of this relationship to future work.

C. Extended ablation study

C.1. Effect of entropy threshold and k_{recent}

We conduct a grid search over $\tau \in \{0.5, 1.0, 1.5, 2.0, 2.5\}$ and $k_{\text{recent}} \in \{16, 32, 64, 128, 256\}$ on both LLaDA and Dream, evaluated on GSM8K (Figure 8). The entropy threshold τ governs the primary accuracy–throughput trade-off: higher thresholds aggressively skip remasking steps, yielding faster generation at the cost of accuracy. Meanwhile, k_{recent} plays a complementary role. By remasking a small window of recent tokens, it recovers errors introduced by aggressive thresholding at only a marginal throughput cost. For example, on Dream at $\tau = 1.5$, increasing k_{recent} from 16 to 64 improves accuracy from 72.9% to 74.3% while reducing throughput by less than 2 tok/s. This allows our method to maintain near-lossless accuracy even at operating points where τ alone would incur noticeable degradation. We select $\tau = 1.5$ and $k_{\text{recent}} = 64$ as our default configuration: on LLaDA it achieves 38.6 tok/s with 78.8% accuracy, and on Dream it reaches 48.4 tok/s with 74.3% accuracy, delivering a substantial throughput gain over the baseline while preserving accuracy.

C.2. Window size ablation

Table 4 reports accuracy and throughput for EntropyCache ($\tau = 1.5$, $k = 64$) across sliding-window sizes $w \in \{8, 16, 32, 64, 128\}$ on all four standard benchmarks.

For LLaDA-Inst, accuracy remains stable across $w = 8$ to $w = 64$ on most tasks, with $w = 32$ offering the best accuracy–throughput balance: it achieves the highest or near-highest accuracy on GSM8K (78.6%), MATH500 (42.8%), and HumanEval (47.6%), while delivering competitive throughput across all benchmarks. Smaller windows ($w = 8, 16$) preserve accuracy but sacrifice throughput due to the increased number of autoregressive blocks required to cover the generation length.

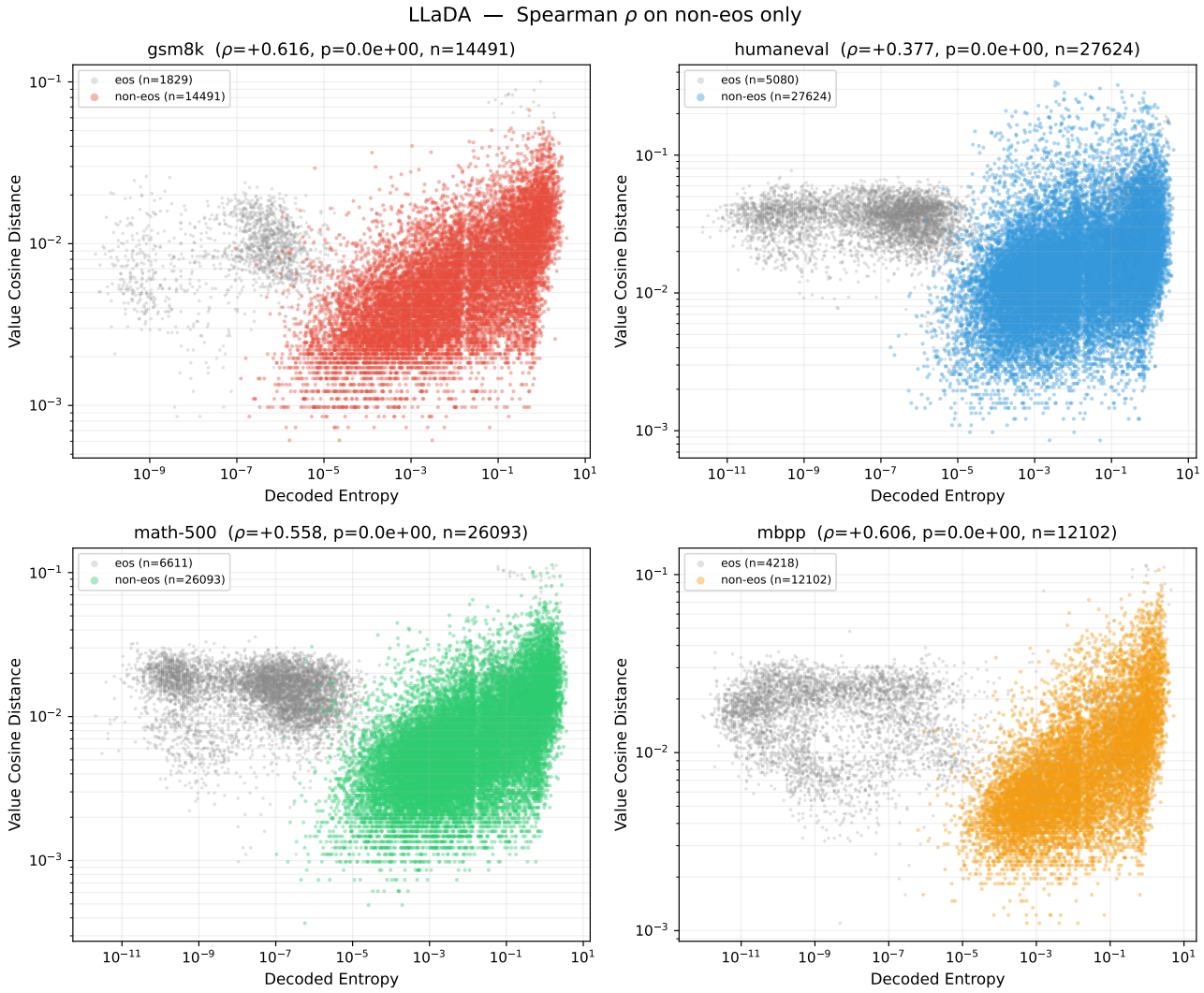

Figure 6. Per-step decoded token entropy vs. average value-vector cosine distance for each benchmark (LLaDA-8B-Instruct, 64 samples per dataset). Gray points denote EOS-filling steps; colored points denote non-EOS content steps. Spearman ρ is computed on non-EOS steps only.

For Dream-Inst, the sensitivity to window size is more pronounced. Accuracy degrades substantially at $w = 64$ and $w = 128$—for instance, GSM8K drops from 74.3% ($w = 32$) to 61.9% ($w = 64$) and 49.7% ($w = 128$), and MATH500 collapses from 43.2% to 10.6% at $w = 128$. This aligns with findings in prior work on semi-autoregressive decoding for dLLMs (Ye et al., 2025; Wu et al., 2025b): when the denoising window is large, the model must resolve many tokens simultaneously without the left-to-right bias that smaller windows naturally impose, leading to degraded generation quality. The $w = 32$ configuration strikes the best trade-off between parallel efficiency and the sequential denoising structure that these models benefit from, and we adopt it as the default for all main experiments.

C.3. EntropyCache parameter ablation on extended datasets

The main text reports the accuracy–throughput grid for GSM8K and averages across benchmarks. Here we present the full per-benchmark ablation over $\tau \in \{0.5, 1.0, 1.5, 2.0, 2.5\}$ and $k \in \{16, 32, 64, 128, 256\}$ on LLaDA-Inst ($w = 32$) in Tables 5–8.

Entropy threshold τ. Across all four benchmarks, accuracy is largely preserved at $\tau \leq 1.5$ and begins to degrade at $\tau \geq 2.0$, while throughput increases monotonically with τ. On GSM8K, accuracy remains within 1%p of the $\tau = 0.5$ baseline up to $\tau = 1.5$ (e.g., 78.8% vs. 79.8% at $k = 64$), but drops by $\sim$3%p at $\tau = 2.0$. A similar pattern holds for MATH500 and HumanEval, where $\tau = 1.5$ maintains

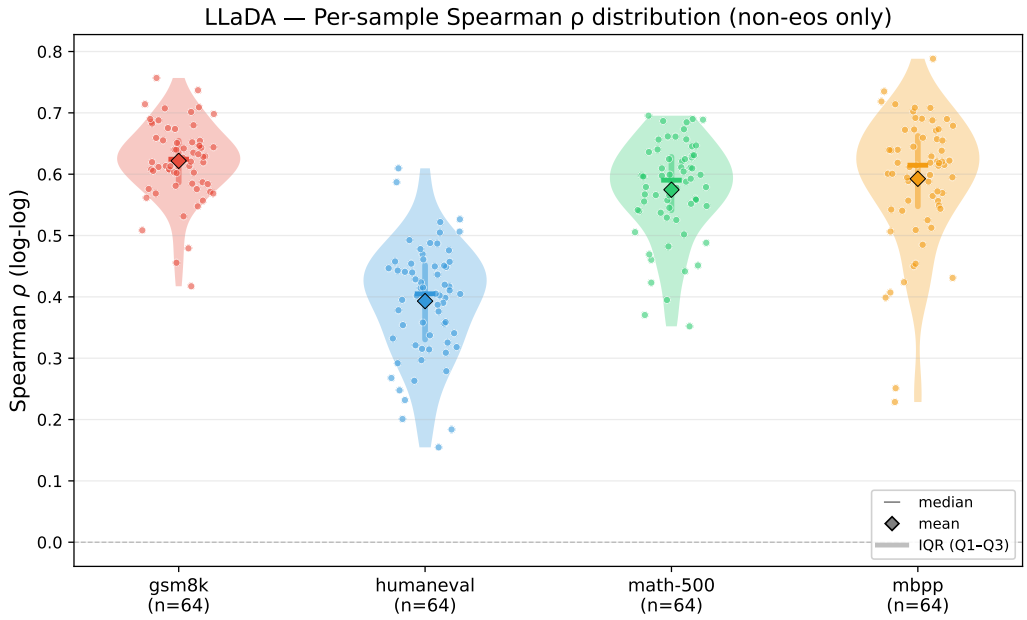

Figure 7. Distribution of per-sample Spearman ρ (log–log, non-EOS only) across 64 samples for each benchmark (LLaDA-8B-Instruct). Diamonds indicate means; horizontal bars indicate medians; shaded regions show kernel density estimates.

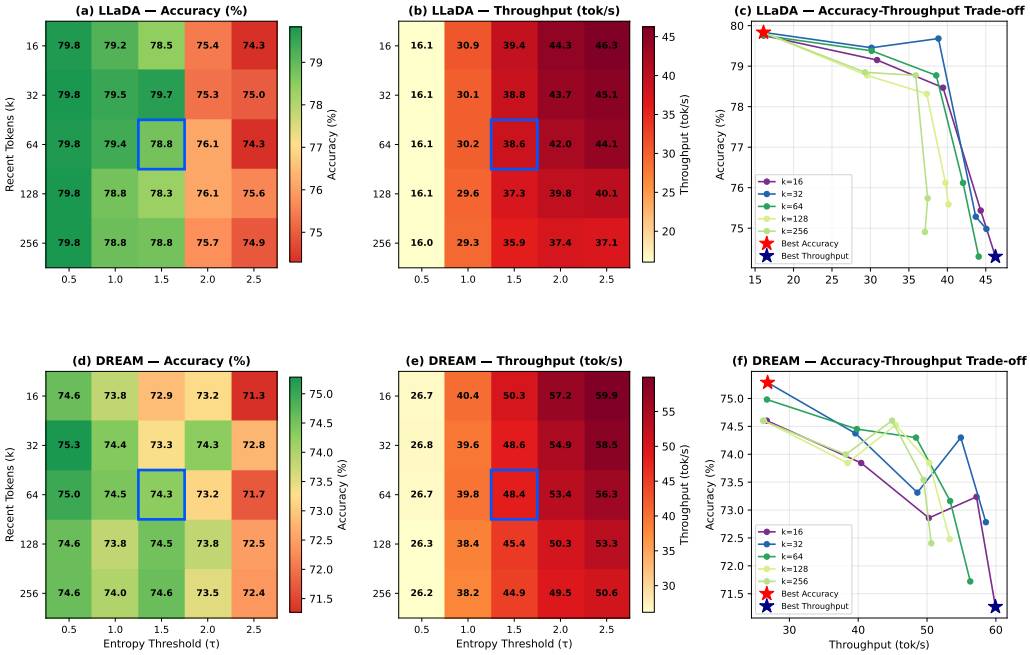

Figure 8. Grid search over entropy threshold τ and recent-token budget k_{recent} on GSM8K. (a, d) Accuracy. (b, e) Throughput. (c, f) Accuracy vs. throughput. Top: LLaDA; bottom: Dream.

competitive accuracy while roughly doubling throughput relative to $\tau = 0.5$. The threshold $\tau = 1.5$ thus consistently sits at the knee of the accuracy–throughput curve across tasks.

Recent token budget k. The effect of k is most visible in throughput: larger k increases the per-step recomputation cost, reducing throughput by 10–15% when moving from $k = 16$ to $k = 256$ at a fixed τ. On accuracy, the impact is more nuanced. For reasoning-heavy tasks (GSM8K, MATH500), moderate values of k (32–64) slightly outper-

Table 4. Window size ablation for EntropyCache ($\tau = 1.5$, $k = 64$). Acc.: Accuracy (%). T-put: Throughput (tok/s). Bold indicates the selected configuration ($w = 32$).

	$w = 8$		$w = 16$		$\mathbf{w = 32}$		$w = 64$		$w = 128$	
Benchmark	Acc.	T-put	Acc.	T-put	Acc.	T-put	Acc.	T-put	Acc.	T-put
LLaDA-Inst										
GSM8K	79.08	31.66	78.92	36.68	**78.62**	**38.62**	77.33	35.82	68.84	28.99
MATH500	41.60	31.34	40.80	37.45	**42.80**	**39.52**	40.60	37.51	38.20	29.77
MBPP	42.00	40.42	40.00	55.52	**38.40**	**64.53**	39.20	65.71	37.80	63.93
HumanEval	47.56	31.64	43.90	35.32	**47.56**	**37.86**	47.56	36.25	50.00	30.72
Dream-Inst										
GSM8K	74.83	37.52	75.36	47.42	**74.30**	**48.06**	61.87	44.67	49.66	38.16
MATH500	43.80	39.10	42.20	54.85	**43.20**	**62.41**	33.60	55.74	10.60	41.87
MBPP	53.00	52.36	56.60	79.68	**48.80**	**97.96**	36.80	103.77	28.20	92.71
HumanEval	54.27	52.96	60.98	86.71	**58.54**	**118.23**	53.66	130.57	43.90	134.79

form very small ($k = 16$) or very large ($k = 256$) budgets, suggesting that too few recent tokens miss ongoing feature volatility while too many dilute the recomputation with already-stable positions. For code generation (MBPP, HumanEval), accuracy is relatively insensitive to k, likely because code tokens stabilize quickly after commitment. The setting $k = 64$ provides the best average accuracy across all four benchmarks without significantly sacrificing throughput relative to smaller values.

Joint selection. Taking both dimensions together, the configuration $\tau = 1.5$, $k = 64$ achieves the highest average accuracy while delivering throughput within 5% of more aggressive settings. We adopt this as the default for all main experiments.

Table 5. EntropyCache ablation on GSM8K (LLaDA-Inst, $w = 32$). Each cell: Accuracy (%) / Throughput (tok/s).

	$\tau = 0.5$	$\tau = 1.0$	$\tau = 1.5$	$\tau = 2.0$	$\tau = 2.5$
$k = 16$	79.76 / 16.12	79.15 / 30.87	78.47 / 39.44	75.44 / 44.34	74.30 / 46.26
$k = 32$	79.83 / 16.10	79.45 / 30.14	79.68 / 38.84	75.28 / 43.69	74.98 / 45.08
$k = 64$	79.76 / 16.13	79.38 / 30.16	**78.77 / 38.56**	76.12 / 42.04	74.30 / 44.07
$k = 128$	79.83 / 16.06	78.77 / 29.56	78.32 / 37.32	76.12 / 39.76	75.59 / 40.14
$k = 256$	79.83 / 16.04	78.85 / 29.31	78.77 / 35.88	75.74 / 37.44	74.91 / 37.09

Table 6. EntropyCache ablation on MATH500 (LLaDA-Inst, $w = 32$). Each cell: Accuracy (%) / Throughput (tok/s).

	$\tau = 0.5$	$\tau = 1.0$	$\tau = 1.5$	$\tau = 2.0$	$\tau = 2.5$
$k = 16$	41.20 / 17.66	41.40 / 30.81	41.00 / 42.03	39.60 / 47.40	39.20 / 50.49
$k = 32$	40.80 / 17.65	41.40 / 30.44	42.20 / 40.40	43.00 / 46.13	40.60 / 47.90
$k = 64$	41.40 / 17.65	41.20 / 30.64	**42.80 / 39.83**	43.40 / 45.04	39.80 / 46.96
$k = 128$	41.20 / 17.48	41.20 / 30.18	43.00 / 37.78	43.20 / 40.82	41.00 / 42.44
$k = 256$	41.20 / 17.26	41.00 / 29.61	42.80 / 35.45	43.60 / 36.18	40.20 / 35.06

Table 7. EntropyCache ablation on MBPP (LLaDA-Inst, $w = 32$). Each cell: Accuracy (%) / Throughput (tok/s).

	$\tau = 0.5$	$\tau = 1.0$	$\tau = 1.5$	$\tau = 2.0$	$\tau = 2.5$
$k = 16$	42.40 / 33.16	41.80 / 53.91	38.80 / 65.32	40.60 / 70.36	38.60 / 72.36
$k = 32$	42.60 / 32.79	41.80 / 53.75	38.80 / 65.19	41.20 / 68.88	40.60 / 71.97
$k = 64$	42.60 / 32.80	41.40 / 53.03	**38.80 / 63.78**	40.00 / 68.68	40.00 / 69.34
$k = 128$	42.60 / 32.20	41.60 / 50.95	38.80 / 60.85	40.60 / 65.73	40.80 / 65.63
$k = 256$	42.60 / 31.75	41.60 / 50.00	38.60 / 59.14	40.40 / 61.82	41.20 / 61.60

Table 8. EntropyCache ablation on HumanEval (LLaDA-Inst, $w = 32$). Each cell: Accuracy (%) / Throughput (tok/s).

	$\tau = 0.5$	$\tau = 1.0$	$\tau = 1.5$	$\tau = 2.0$	$\tau = 2.5$
$k = 16$	45.73 / 24.55	45.73 / 33.87	47.56 / 38.97	46.34 / 41.73	42.07 / 43.91
$k = 32$	45.73 / 24.32	48.78 / 33.13	46.34 / 37.91	44.51 / 40.64	41.46 / 41.00
$k = 64$	45.73 / 24.34	48.17 / 33.54	**48.78 / 38.10**	49.39 / 39.45	42.07 / 39.46
$k = 128$	45.73 / 24.30	46.95 / 32.95	48.78 / 36.85	45.12 / 37.41	43.90 / 36.29
$k = 256$	45.73 / 24.02	47.56 / 31.85	49.39 / 35.73	47.56 / 33.45	46.34 / 29.19

D. Recompute ratio and throughput analysis

We measure the *recompute ratio*—the fraction of KV states recomputed from scratch per step (baseline = 1.0)—to disentangle cache reuse from decision overhead. Table 9 reports results for the three dynamic methods.

On GSM8K and MATH500, EntropyCache achieves both the lowest recompute ratio and the highest throughput, indicating that the entropy trigger skips more steps without sacrificing cache freshness. On MBPP (LLaDA), EntropyCache recomputes slightly more than d^{2}Cache (0.178 vs. 0.164) yet delivers 23% higher throughput (64.9 vs. 52.8 tok/s), demonstrating the wall-clock benefit of its constant-overhead decision rule over d^{2}Cache's context-length-dependent attention rollout. On HumanEval (LLaDA), EntropyCache has a marginally higher recompute ratio and lower throughput than d^{2}Cache, but achieves the highest

accuracy among all methods (48.78% vs. d^{2}Cache 41.46%, Table 1), suggesting that entropy-triggered recomputation allocates compute to generation-critical steps rather than minimizing recomputation indiscriminately.

Table 9. Recompute ratio (RR) and throughput (T-put, tok/s) for dynamic KV caching methods. Lower RR indicates more cache reuse. Bold: best per benchmark.

	Elastic-Cache		d^{2}Cache		Ours	
Benchmark	RR	T-put	RR	T-put	RR	T-put
LLaDA-Inst						
GSM8K	0.140	23.49	0.147	29.96	**0.114**	**38.49**
MATH500	0.148	24.43	0.146	32.18	**0.134**	**39.64**
MBPP	0.243	32.24	**0.164**	52.77	0.178	**64.91**
HumanEval	0.168	27.73	**0.160**	41.36	0.172	38.41
Average	0.175	26.97	0.154	39.06	**0.149**	**45.36**
Dream-Inst						
GSM8K	0.152	32.15	0.145	41.68	**0.123**	**48.32**
MATH500	0.148	41.90	0.147	55.80	**0.125**	**62.48**
MBPP	0.348	54.84	0.166	87.63	**0.145**	**97.79**
HumanEval	0.281	72.30	0.163	110.83	**0.160**	**119.02**
Average	0.232	50.30	0.155	73.98	**0.138**	**81.90**

E. Memory usage analysis

Figure 9 profiles GPU memory usage over time for the three dynamic KV caching methods on a single GSM8K sample (LLaDA-Inst, $w = 32$, generation length $= 256$). EntropyCache consumes the most memory (peak 19.85 GB) because it caches not only key–value pairs but also query projections and hidden states for its per-layer drift comparison. d^{2}Cache and EntropyCache are nearly identical at $\sim$18.85 GB, as neither requires substantial auxiliary storage beyond the KV cache itself. While the absolute differences are modest—model weights and the KV cache dominate the footprint—EntropyCache's auxiliary cost is $O(N)$ (one integer per generated token), remaining constant regardless of prompt length, unlike the context- or depth-dependent buffers of the other methods.

F. Limitations and broader impacts

Limitations. As with any approximate inference acceleration method, EntropyCache exhibits accuracy variations across task–model combinations under a single fixed configuration. Our analyses (Section 5.3, Appendix C.3) demonstrate that EntropyCache maintains favorable accuracy–throughput trade-offs across the operating range and outperforms prior caching methods once τ and k_{recent} are appropriately set. We recommend that practitioners calibrate these hyperparameters to their target workload to obtain the best operating point, particularly for tasks such as code generation where token stability dynamics may differ from natural language reasoning.

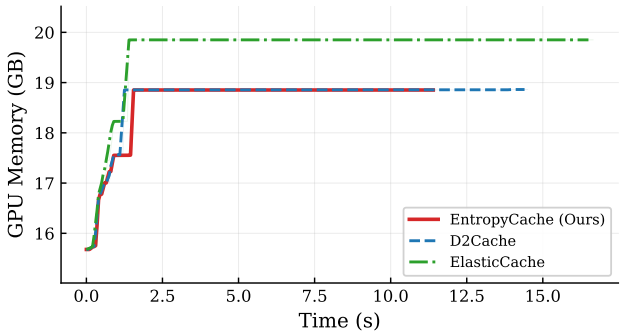

Figure 9. GPU memory usage over time on a single GSM8K sample (LLaDA-Inst, $w = 32$, generation length $= 256$). ElasticCache requires $\sim$1 GB more due to extra query and hidden-state caching; d^{2}Cache and EntropyCache are nearly identical.

Additionally, our evaluation covers two dLLM architectures (LLaDA-8B and Dream-7B), the most widely adopted discrete masked diffusion language models in the current research landscape. All prior KV caching works in this space (Jiang et al., 2025; Nguyen-Tri et al., 2025; Wu et al., 2025b; Ma et al., 2025) evaluate on the same two models. As larger or more diverse dLLMs become available, validating EntropyCache at greater scale would be a natural next step.

Broader impacts. By substantially reducing per-request inference cost, EntropyCache can lower the energy and hardware requirements for deploying dLLMs, making diffusion-based generation more accessible in resource-constrained settings. As with any inference acceleration technique, practitioners should verify that approximate caching does not introduce subtle quality regressions in safety-critical applications.

Looking ahead, the entropy-based skipping principle is not limited to the sliding-window decoding setting studied here. Recent block diffusion language models such as Fast-dLLMv2 (Wu et al., 2025a) generate text in coarse blocks that are internally refined through sub-block denoising steps. Since each sub-block undergoes iterative denoising with its own KV dynamics, the entropy trigger proposed in this work could serve as a lightweight recomputation signal within sub-blocks, complementing the block-level KV reuse that these architectures already employ. We leave this extension to future work.

G. Sample responses

G.1. Math

Figure 10 shows responses from each method on a GSM8K example requiring multi-step cumulative reasoning (ID: 546,

answer: 130). Highlighted tokens in the EntropyCache output mark positions where decoded entropy exceeded τ, triggering full KV recomputation. These tend to coincide with transition words and numerically significant tokens. The remaining methods all arrive at incorrect answers.

G.2. Coding

Figure 11 shows responses from each method on a HumanEval example requiring interval intersection logic (ID: 127). The task asks whether the length of the intersection of two closed integer intervals is a prime number. Highlighted tokens in the EntropyCache output mark positions where decoded entropy exceeded τ, triggering full KV recomputation. These tend to coincide with logical connectives in the reasoning preamble and the assignment operator in the critical formula. Only EntropyCache produces the correct implementation, while the remaining methods all fail the test cases.

Sample Responses on GSM8K (ID: 546) | Ground Truth Answer: 130

Question

Elise is learning to write and decides to keep re-writing the alphabet until she knows it. She writes it in full twice, writes half of it once, then re-writes everything she has already written. How many letters has Elise written in total?

Ground Truth — 130

```
Elise has written the alphabet twice which is a total of 26 * 2 = 52 letters.
She then writes half the alphabet, which is 26 / 2 = 13 letters.
So far, this is a total of 52 + 13 = 65 letters.
Writing this again means she has doubled the number of letters she has written,
so she has written a total of 65 * 2 = 130 letters.
```

EntropyCache (Ours) — ✓ Correct (130)

```
Elise writes the full alphabet twice, which is 26 letters x 2 = <<26*2=52>>52 letters
.She then writes half of the alphabet once, which is 26 letters / 2 = <<26/2=13>>13 l
etters.After writing the full alphabet twice and half of it once, she has written a t
otal of 52 + 13 letters = <<52+13=65>>65 letters.Elise then re-writes everything she
has already written. Since she has already written 65 letters, she will write 65 lett
ers again.Therefore, the total number of letters Elise has written is 65 + 65 = <<65+
65=130>>130 letters.#### 130
```

token = decoded token with entropy $\geq \tau$, triggering full KV recomputation at next step

Fast-dLLM — ✗ Incorrect (91)

```
Elise writes the full alphabet twice, which is 26 * 2 = 52 letters.
She then writes half of the alphabet once, which is 26 / 2 = 13 letters.
Elise re-writes everything she has already written, which means she writes
the full alphabet again, which is 26 letters.
Adding these together, Elise writes a total of 52 + 13 + 26 = 91 letters.
```

⚠ *Misinterprets "re-writes everything" as writing only the alphabet (26) instead of all 65 letters.*

D2Cache — ✗ Incorrect (195)

```
Elise writes the full alphabet twice, so she writes 2 * 26 = <<2*2*2=52>>52 letters.
She then writes half of the alphabet once, which is 26 / 2 = <<26/2=13>>13 letters.
After writing half of the alphabet, she has written 52 + 13 = <<52+13=65>>65 letters.
Elise then re-writes everything she has already written, which means she writes 65 *
2 = <<65*2=130>>130 letters.
Therefore, the total number of letters Elise has written is 52 + 13 + 130 =
<<52+13+130=195>>195 letters.
#### 195
```

⚠ *Computes re-write as 65 * 2 = 130, then adds all three (52+13+130) instead of recognizing 65 * 2 = 130.*

ElasticCache — ✗ Incorrect (195)

```
1. Writing the full alphabet twice: 26 * 2 = 52 letters.
2. Writing half of the alphabet once: 26 / 2 = 13 letters.
3. Re-writing everything she has already written:
   She has already written 52 + 13 = 65 letters.
   Re-writing everything means she writes 65 * 2 = 130 letters.
Total: 52 + 13 + 130 = 195 letters.
```

⚠ *Same error as D2Cache: double-counts by adding original (65) and rewritten (130).*

Figure 10. Sample responses on GSM8K (ID: 546, ground truth: 130). Highlighted tokens in the EntropyCache response denote decoded tokens with entropy $\geq \tau$ that triggered full KV recomputation at the next denoising step.

Sample Responses on HumanEval (ID: 127) | Task: intersection(interval1, interval2)

Prompt

```
def intersection(interval1, interval2):
    """You are given two intervals, each a pair of integers (start, end).
    Intervals are closed (inclusive). Determine whether the length of
    intersection of the two intervals is a prime number.
    intersection((1,3),(2,4))    -> length 1 -> NO
    intersection((-1,2),(2,4))   -> length 0 -> NO
    intersection((-3,-1),(-5,5)) -> length 2 -> YES
    """
```

EntropyCache (Ours) — ✓ Correct (PASS)

```
To solve this problem, we need to follow these steps:

1. Determine if the two intervals intersect.
2. Calculate the length of the intersection.
3. Check if the length is a prime number.

Let's break down the steps:

1. Check if the intervals intersect:
   - Two intervals intersect if the start of the first
   interval is less than or equal to the end of the second
   interval and the start of the second interval is less than
   or equal to the end of the first interval.

2. Calculate the length of the intersection:
   - The length of the intersection is the difference between
   the start and end of the intersection.

3. Check if the length is a prime number:
   - A prime number is a number greater than 1 that has no
   positive divisors other than 1 and itself.

Let's implement this in Python:

def is_prime(n):
    if n <= 1:
        return False
    if n == 2:
        return True
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True

def intersection(interval1, interval2):
    start1, end1 = interval1
    start2, end2 = interval2

    if start1 > end2 or start2 > end1:
        return "NO"

    intersection_length = min(end1, end2) - max(start1, start2)

    if is_prime(intersection_length):
        return "YES"
    else:
        return "NO"
This code defines a helper function is_prime to check if a
number is prime. The intersection function checks if the
intervals intersect, calculates the length of the
intersection, and then checks if this length is a prime
number. If the intersection length is prime, it returns "YES";
otherwise, it returns "NO".
```

token = token with entropy ≥ τ -> full KV recompute

Fast-dLLM — ✗ Incorrect (FAIL)

```
To solve this problem, we need to follow these steps:

1. Determine if the two intervals intersect.
2. Calculate the length of the intersection.
3. Check if the length is a prime number.
4. Return the appropriate result based on the check.

Here's the implementation of the intersection function:

def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

def intersection(interval1, interval2):
    start1, end1 = interval1
    start2, end2 = interval2
    # Check if the intervals intersect
    if start1 > end2 or start2 > end1:
        return "NO"
    # Calculate the length of the intersection
    intersection_length = min(end1, end2) - max(start1, start2) + 1
    # Check if the length is a prime number
    if is_prime(intersection_length):
        return "YES"
    else:
        return "NO"

Test cases
print(intersection((1, 2), (2, 3)))   # Output: "NO"
print(intersection((-1, 1), (0, 4)))  # Output: "NO"
print(intersection((-3, -1), (-5, 5)))# Output: "YES"
Explanation:

1. is_prime function: This helper function checks if a number
is prime.
2. intersection function:
   - We first check if the intervals intersect. If they don't,
   we return "NO".
   - If they intersect, we calculate the length of the
   intersection.
   - We then check if this length is prime using the is_prime
   function.
   - Finally, we return "YES" if the length is prime,
   otherwise "NO".

This implementation should work correctly for the given
problem.
```

⚠ *Off-by-one: adds "+ 1" to intersection_length; correct is min(end1,end2) - max(start1,start2).*

D2Cache — ✗ Incorrect (FAIL)

```
To solve this problem, we need to follow these steps:

1. Check if the two intervals intersect.
2. Calculate the length of the intersection.
3. Check if the length is a prime number.

Here's the implementation of the intersection function:

def is_prime(n):
    """Check if a number is prime."""
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

def intersection(interval1, interval2):
    """Check if the intervals intersect and if the length is prime."""
    start1, end1 = interval1
    start2, end2 = interval2
    # Check if the intervals intersect
    if end1 < start2 or end2 < start1:
        return "NO"
    # Calculate the length of the intersection
    intersection_length = min(end1, end2) - max(start1, start2) + 1
    # Check if the length is a prime number
    if is_prime(intersection_length):
        return "YES"
    else:
        return "NO"

Test cases
print(intersection((1, 2), (2, 3)))   # "NO"
print(intersection((-1, 1), (0, 4)))  # "NO"
print(intersection((-3, -1), (-5, 5)))# "YES"
Explanation:
1. is_prime function: This helper function checks if a number
is prime.
2. intersection function:
   - It first checks if the two intervals intersect.
   - If they intersect, it calculates the length of the
   intersection.
   - It then checks if this length is prime using the is_prime
   function.
   - Finally, it returns "YES" if prime, otherwise "NO".

This implementation should work correctly for the given
problem.
```

⚠ *Same off-by-one: "+ 1" inflates length; e.g. non-prime length 1 becomes 2 (prime) → wrong YES.*

ElasticCache — ✗ Incorrect (FAIL)

```
To solve this problem, we need to follow these steps:

1. Check if the two intervals intersect.
2. If they intersect, calculate the length of the
intersection.
3. Check if the length is a prime number.
4. Return "YES" if the length is prime, otherwise return "NO".

Here's the implementation of the intersection function:

def is_prime(n):
    """Check if a number is prime."""
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

def intersection(interval1, interval2):
    # Check if the intervals intersect
    start1, end1 = interval1
    start2, end2 = interval2
    if start1 > end2 or end1 < start2:
        return "NO"
    # Calculate the length of the intersection
    intersection_length = min(end1, start2) - max(start1, end2) + 1
    # Check if the length is a prime number
    if is_prime(intersection_length):
        return "YES"
    else:
        return "NO"

Test cases
print(intersection((1, 2), (2, 3)))   # Output: "NO"
print(intersection((-1, 1), (0, 4)))  # Output: "NO"
print(intersection((-3, -1), (-5, 5)))# Output: "YES"
Explanation:

1. is_prime function: This helper function checks if a number
is prime.
2. intersection function:
   - It first checks if the two intervals intersect.
   - If they intersect, it calculates the length of the
   intersection.
   - It checks if this length is prime using the is_prime
   function.
   - If it is, it returns "YES", otherwise it returns "NO".
```

⚠ *Wrong variables: uses min(end1,start2) - max(start1,end2) instead of min(end1,end2) - max(start1,start2).*

Figure 11. Sample responses on HumanEval (ID: 127). Highlighted tokens in the EntropyCache response denote decoded tokens with entropy $\geq \tau$ that triggered full KV recomputation at the next denoising step.