TRACEQUERY: LEARNING WHAT TO RECOMPUTE FROM FULL-DENOISING TRAJECTORY IN DIFFUSION LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Diffusion large language models (dLLMs) enable flexible generation through iterative masked denoising, but incur substantial inference cost because token representations are repeatedly recomputed across denoising steps. Existing cache-based methods reduce this cost by reusing intermediate results across denoising steps and recomputing only a subset of positions at each step, chosen either on a fixed schedule or by heuristic signals of change. Our trajectory experiments instead point to decoding trajectory as the more relevant quantity. Cached generation remains accurate when it stays close to the full-denoising trajectory, with errors concentrated around unmask-order deviations and reduced as longer reference prefixes are recovered. This suggests a more direct principle: **limiting representation error helps, but keeping generation on the full-denoising trajectory is a more direct route to accurate decoding**. We therefore formulate efficient dLLM inference as *trajectory-guided querying*. Rather than broadly repairing representation error, we use full-denoising trajectories to supervise which unresolved positions should receive limited fresh computation. Based on this formulation, we introduce TRACEQUERY, a lightweight sparse decoding framework that predicts a small set of near-future trajectory-relevant positions, queries only those candidates, and lets the backbone select which one to commit from their fresh predictions. TRACEQUERY delivers the strongest quality–computation trade-off among cache-based approaches, sustaining high accuracy with substantially fewer fresh token computations, and generalizes across tasks, model variants, and families.

1 INTRODUCTION

Diffusion large language models (dLLMs) have emerged as a promising alternative to autoregressive language models by replacing fixed left-to-right generation with iterative masked denoising (Sahoo et al., 2024; Ou et al., 2025; Nie et al., 2026). By repeatedly predicting unresolved tokens under bidirectional attention, dLLMs support flexible generation orders and expose opportunities for parallel decoding (Ye et al., 2025; Song et al., 2025; Labs et al., 2025). However, these advantages come with substantial inference cost. At each denoising step, standard dLLMs repeatedly recompute token representations over large portions of the sequence, even though only one or a few tokens may ultimately be committed (Hu et al., 2026; Cheong et al., 2026). When the number of denoising steps grows with generation length, this redundant recomputation can lead to cubic decoding cost (Ma et al., 2026; Liu et al., 2026b). To reduce this redundancy, recent work maintains an approximate KV cache across denoising steps, reusing stable representations and selectively refreshing only a subset of positions (Ma et al., 2026; Wu et al., 2026; Liu et al., 2026b; Jiang et al., 2026). Despite increasingly fine-grained refresh policies, these methods share a common objective: *limiting the representation error introduced by cache reuse*.

We revisit this objective by examining how cached decoding behaves relative to full denoising. Our trajectory analysis (Figure 1) shows that cached decoding deviates from the full-denoising unmasking order even when token-level predictions agree. Accuracy also drops as the injected full-denoising prefix shortens, tying quality to keeping generation anchored to the reference trajectory. These observations motivate the following causal view:

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

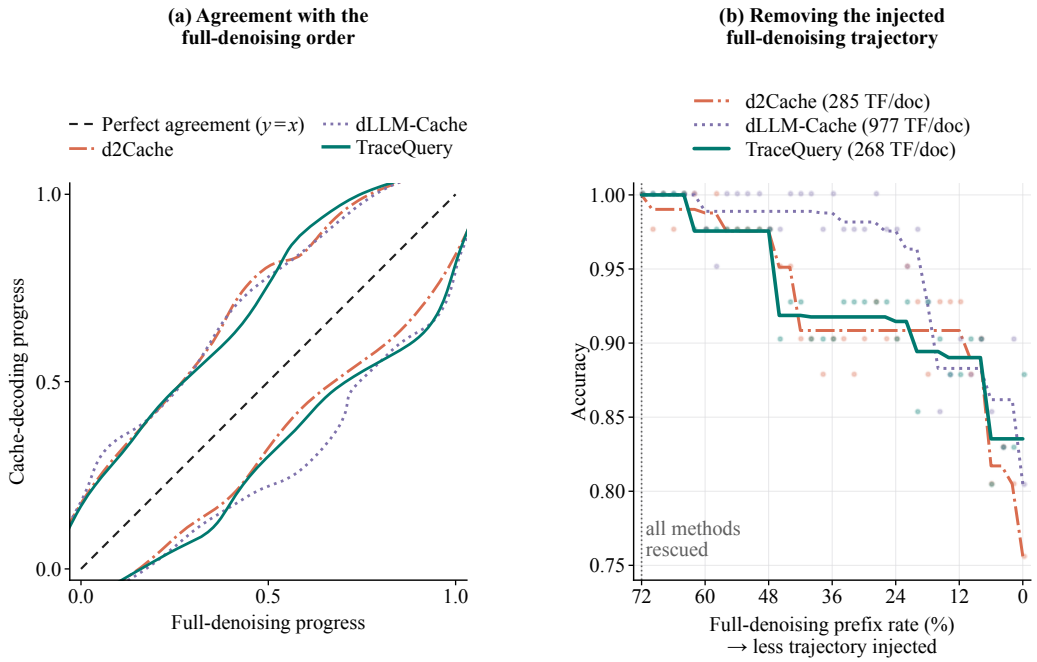


Figure 1: **Trajectory agreement and dependence on trajectory guidance.** **(a)** We compare decoding progress under full denoising and cached decoding. Each underlying observation corresponds to a token position, paired by the step at which it is unmasked under full denoising and its corresponding step under cached decoding. Rather than plotting individual observations, we show the 80% density contours. Concentration along the diagonal indicates agreement with the full-denoising unmask order, while broader deviations reflect trajectory divergence. **(b)** Accuracy on examples solved by full denoising as the injected full-denoising prefix is progressively removed.

representation error → *decision perturbation* → *trajectory divergence* → *quality degradation*.

Refreshing cache entries intervenes at the first link of this chain by reducing representation error, but it controls decision perturbation and trajectory divergence only indirectly. The link from representation error to decoding decisions is highly non-uniform. Many errors are decision-irrelevant and leave the next commitment unchanged, while a small subset can alter a consequential decision and redirect the subsequent trajectory. This makes representation repair inefficient, because computation may be spent correcting harmless errors, while even useful corrections must propagate through later representations and decisions before improving the decoding trajectory. This suggests a more direct principle: **limiting representation error helps, but keeping generation on the full-denoising trajectory is a more direct route to accurate decoding.**

This view gives the full-denoising trajectory a new role. A full-denoising run provides a sequence of decisions made by the original backbone from fully recomputed representations, without cache-induced error. Although multiple decoding trajectories may lead to equally good generations, this trajectory is immediately available from the model itself and requires neither additional trajectory search nor external supervision. It therefore provides a natural, unapproximated reference for anticipating the decoding decisions that lie in the near future.

Based on this view, we formulate efficient dLLM inference as *trajectory-guided querying*. Rather than searching for an optimal trajectory, we use the already available full-denoising trajectory to guide where a limited query budget should be spent. At each denoising step, the objective is to identify unresolved positions that appear soon along the full-denoising reference trajectory and include them in a small fresh-query set. Importantly, the trajectory does not determine the final decoding decision. It supervises candidate selection, while the backbone queries the selected candidates with fresh computation and retains control over which position to commit.

We introduce **TRACEQUERY**, a lightweight trajectory-guided sparse decoding framework illustrated in Figure 2. During training, TRACEQUERY learns from full-denoising trajectories to rank unresolved positions according to their near-future relevance. During inference, it selects a small candidate set from inexpensive signals available in the current decoding state. Only these candidates receive fresh computation, while the remaining positions continue to reuse cached representations. The backbone dLLM then selects the final commitment from the freshly evaluated candidates. TRACEQUERY therefore splits the two decisions: the router decides where fresh computation goes, and the backbone decides what to commit.

Our main contributions are summarized as follows:

- We propose *trajectory-guided querying*. Representation error degrades generation through perturbed consequential decisions, so rather than spending limited computation to repair the error, we let the available full-denoising trajectory guide it directly toward the decisions that lie ahead.
- We build **TRACEQUERY**, a lightweight sparse decoding framework that implements this formulation with a router trained on full-denoising trajectories. From inexpensive decoding-state signals, the router decides which few positions receive fresh computation, and the backbone decides what to commit from the freshly evaluated candidates.
- A single learned router approaches full-denoising quality with substantially fewer fresh token computations across tasks, model variants, and dLLM families, improving the quality–computation trade-off over cache-based baselines.

2 RELATED WORK

Diffusion language models generate text through iterative masked denoising rather than fixed left-to-right factorization, enabling bidirectional context and flexible decoding orders (Austin et al., 2021; Li et al., 2022; Lou et al., 2023; Sahoo et al., 2024; Ou et al., 2025). Recent large-scale models such as LLaDA and Dream demonstrate that this paradigm can scale to competitive language modeling and reasoning performance (Nie et al., 2026; Ye et al., 2025), while systems such as Seed and Mercury further highlight its potential for parallel generation (Song et al., 2025; Labs et al., 2025).

Learned and Trajectory-Aware Decoding. Prior methods learn, search, guide, or verify unmasking decisions and denoising trajectories (Jazbec et al., 2025; Liu et al., 2026a; Hu et al., 2026; Pan et al., 2026). TRACEQUERY instead learns which masked positions to *query*: its router allocates fresh computation, while the backbone determines the committed position and token. Full-denoising trajectories provide offline supervision, with no external guider, online search, or draft verification at inference.

KV Reuse and Adaptive Refresh. Exact KV reuse is difficult in dLLMs because bidirectional attention changes token representations as decoding proceeds. Early methods reuse states across completed blocks (Arriola et al., 2025) or stabilized regions, with periodic recomputation to limit staleness (Hu et al., 2026; Wu et al., 2026). dKV (Ma et al., 2026) identifies *local fidelity*: each decoding decision changes representations most strongly at a limited set of positions, making uniform refresh unnecessary. Building on this principle, dLLM-Cache (Liu et al., 2026b) uses V-state verification to select refresh positions, while d2Cache (Jiang et al., 2026) combines confidence and local mask density for unresolved positions with attention-based selection for prompt and decoded positions. These methods adapt which cached states to refresh; TRACEQUERY focuses on which unresolved positions receive fresh queries before the next commitment.

Different from the above works, our analysis suggests that the downstream effect of cache-induced error is more closely reflected by trajectory deviation than by representation error alone. We use the available full-denoising trajectory to guide fresh computation toward positions that are likely to matter for upcoming decoding decisions.

3 PRELIMINARIES

This section introduces masked diffusion decoding, the full-denoising reference trajectory, and the cache setting used by TRACEQUERY. These definitions apply to both single-block and block-wise diffusion decoding.

3.1 Masked Diffusion Decoding

A masked diffusion language model operates on a partially masked sequence $x^{(t)}$ at denoising step t. Let $\mathcal{M}_t$ and $\mathcal{U}_t$ denote the unresolved and already-resolved positions, respectively. The model predicts token distributions for positions in $\mathcal{M}_t$ in parallel, and a decoding policy selects a subset $\mathcal{A}_t \subseteq \mathcal{M}_t$ to commit:

$$p_{\mathrm{DLLM}}(x_{1:N} \mid \tau) = \prod_{t=0}^{T-1} \prod_{i \in \mathcal{A}_t} p_\theta(x_i \mid x^{(t)}), \qquad \tau = (\mathcal{A}_0, \ldots, \mathcal{A}_{T-1}), \tag{1}$$

where N is the sequence length, T is the number of denoising steps, and τ denotes the decoding trajectory induced by the sequence of commitments. After committing $\mathcal{A}_t$, the resolved and unresolved sets evolve as

$$\mathcal{U}_{t+1} = \mathcal{U}_t \cup \mathcal{A}_t, \qquad \mathcal{M}_{t+1} = \mathcal{M}_t \setminus \mathcal{A}_t. \tag{2}$$

Under single-token decoding, $\mathcal{A}_t = \{i_t\}$, where i_t is the position selected at step t. Unlike autoregressive decoding, t specifies only temporal progression and is not tied to a fixed sequence position. Consequently, the ordering of committed positions itself becomes part of the decoding process.

3.2 Full-Denoising Reference Trajectory

For a given prompt, we fix the backbone model, decoding policy, denoising schedule, and token-sampling configuration. At each denoising step, the full-denoising baseline performs a forward pass over the entire active generation region and recomputes the representations used to make the next commitment. In particular, it does not reuse approximate representations from earlier denoising steps. The decoding policy then selects the next position from predictions obtained using these fully recomputed representations.

Let $\mathcal{A}_t^{\mathrm{FD}}$ denote the set of positions committed at step t by full denoising. Under single-token decoding, $\mathcal{A}_t^{\mathrm{FD}} = \{i_t^{\mathrm{FD}}\}$, and we define the full-denoising reference trajectory as

$$\tau^{\mathrm{FD}} = \left(\mathcal{A}_0^{\mathrm{FD}}, \ldots, \mathcal{A}_{T-1}^{\mathrm{FD}}\right) = \left(i_0^{\mathrm{FD}}, \ldots, i_{T-1}^{\mathrm{FD}}\right). \tag{3}$$

The trajectory records only the order in which positions are unmasked; it does not include the identities of the committed tokens. The superscript FD indicates that the corresponding decisions are produced by full denoising. We use τ^{FD} as an operational reference for the behavior of the unmodified backbone and decoding policy in the absence of cache-induced approximation. It is neither assumed to be the unique decoding trajectory nor guaranteed to produce a correct or optimal output. Nevertheless, it can be generated offline directly from the frozen backbone, without external annotations or additional trajectory search, and therefore provides a well-defined target for supervising and evaluating trajectory agreement under cached decoding.

3.3 Cached Diffusion Inference

We partition the sequence into three regions: the prefix $\mathcal{P}$, the active generation region $\mathcal{G}$, and the postfix $\mathcal{S}$. Previously completed blocks belong to $\mathcal{P}$, while subsequent blocks belong to $\mathcal{S}$. At denoising step t, the active generation region is partitioned into

$$\mathcal{G} = \mathcal{U}_t \cup \mathcal{M}_t, \qquad \mathcal{U}_t \cap \mathcal{M}_t = \emptyset. \tag{4}$$

Decoding starts with a full forward pass to initialize the persistent KV cache. Following existing region-wise caching strategies (Wu et al., 2026; Liu et al., 2026b), the active generation region is periodically recomputed to control accumulated approximation error, while the prefix and postfix are reused over longer intervals. TraceQuery operates only within the active generation region $\mathcal{G}$.

4 Methodology

4.1 Trajectory Recovery under a Query Budget

Let τ^{FD} be the full-denoising reference trajectory defined in Section 3.2. After step t, its remaining ordered suffix is

$$\pi_t^{\mathrm{FD}} = \left(i_{t+1}^{\mathrm{FD}}, \ldots, i_{T-1}^{\mathrm{FD}}\right). \tag{5}$$

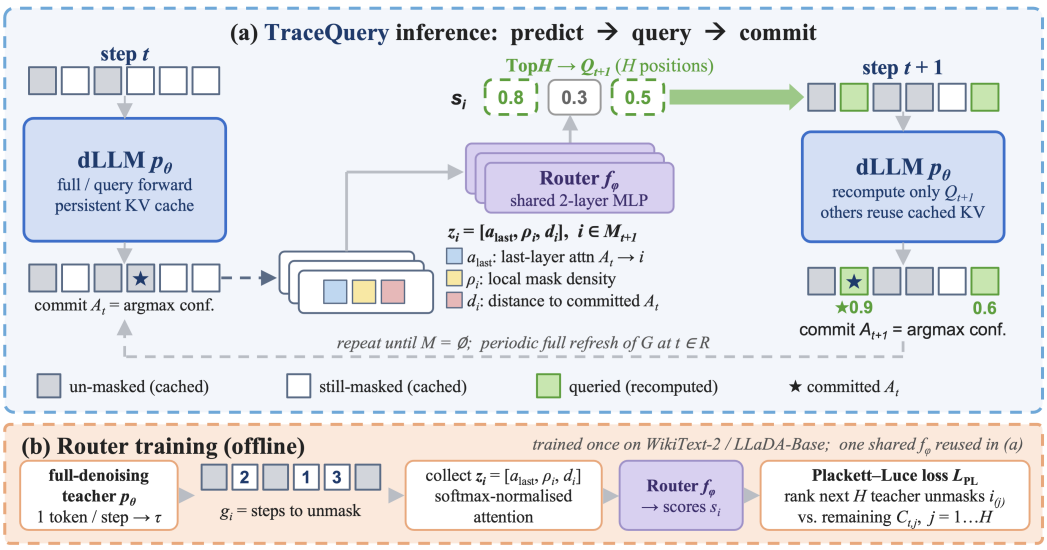


Figure 2: Overview of TRACEQUERY. Full denoising first provides an offline reference trajectory describing the future order of decoding decisions. The router is trained to recover the near-future portion of this trajectory from inexpensive signals available during decoding. At inference time, after step t commits $\mathcal{A}_t$, the router ranks the remaining unresolved positions and selects a small query set $\mathcal{Q}_{t+1}$. Only these positions receive fresh computation, while other positions reuse cached representations. The backbone dLLM then makes the final commitment from the freshly queried candidates. Periodic broad recomputation is retained to control accumulated approximation error.

Given a maximum query horizon H, we define

$$\begin{aligned} H_t &= \min\left(H, \left|\mathcal{M}_{t+1}^{\mathrm{FD}}\right|\right), \\ \pi_{t,H_t}^{\mathrm{FD}} &= \left(i_{t+1}^{\mathrm{FD}}, \ldots, i_{t+H_t}^{\mathrm{FD}}\right), \\ \mathcal{Q}_{t+1}^{\mathrm{oracle}} &= \left\{i_{t+1}^{\mathrm{FD}}, \ldots, i_{t+H_t}^{\mathrm{FD}}\right\}, \end{aligned} \tag{6}$$

where $\mathcal{M}_{t+1}^{\mathrm{FD}}$ is the masked set in the corresponding full-denoising state. The ordered prefix $\pi_{t,H_t}^{\mathrm{FD}}$ specifies the next H_t reference decisions, while $\mathcal{Q}_{t+1}^{\mathrm{oracle}}$ is the query set that would be available if the reference trajectory were known at inference time. Here, "oracle" denotes access to τ^{FD}, rather than global optimality. We formulate computation allocation as predicting this near-future reference set from the current decoding state. During training, $\pi_{t,H_t}^{\mathrm{FD}}$ provides ordered supervision; during inference, TRACEQUERY selects at most H positions using only signals available from the cached decoding state. Thus, the target is determined by future decoding behavior rather than by cached-state magnitude, age, or representation error.

4.2 TRAJECTORY-SUPERVISED ROUTER TRAINING

We train the TRACEQUERY router offline on intermediate states collected from full-denoising trajectories. For each reference state, let $i_{t,j}^{\mathrm{FD}}$ denote the j-th element of $\pi_{t,H_t}^{\mathrm{FD}}$.

For every unresolved position $i \in \mathcal{M}_{t+1}^{\mathrm{FD}}$, we construct

$$\begin{aligned} \mathbf{z}_i^{(t)} &= \left[a_{i,\mathrm{last}}^{(t)}, \rho_i^{(t)}, d_i^{(t)}\right], \\ s_i^{(t)} &= f_\phi\left(\mathbf{z}_i^{(t)}\right), \end{aligned} \tag{7}$$

where the features are normalized last-layer attention from recently committed positions, local mask density, and relative distance to recently committed positions, respectively. The scorer f_ϕ is a two-layer MLP shared across positions and denoising steps. Detailed feature definitions are provided in Appendix B.

At ranking stage j, previously selected reference positions are removed:

$$\mathcal{C}_{t,j} = \mathcal{M}_{t+1}^{\mathrm{FD}} \setminus \left\{ i_{t,1}^{\mathrm{FD}}, \ldots, i_{t,j-1}^{\mathrm{FD}} \right\}. \tag{8}$$

We train the router with a truncated Plackett–Luce objective:

$$\mathcal{L}_{\mathrm{PL}} = -\frac{1}{|\mathcal{D}|} \sum_{(x,t)\in\mathcal{D}} \sum_{j=1}^{H_t} \log \frac{\exp\left(s_{i_{t,j}^{\mathrm{FD}}}^{(t)}\right)}{\sum_{i\in\mathcal{C}_{t,j}} \exp\left(s_i^{(t)}\right)}, \tag{9}$$

where $\mathcal{D}$ is the collection of intermediate reference states. The objective assigns higher scores to positions that occur earlier in the future full-denoising order. The reference trajectory is used only during offline training.

4.3 TraceQuery Inference

At inference time, the trained router scores every unresolved position $i \in \mathcal{M}_{t+1}$ and selects the H_t highest-scoring positions:

$$\mathcal{Q}_{t+1} = \operatorname*{arg\,max}_{\substack{\mathcal{Q}\subseteq\mathcal{M}_{t+1} \\ |\mathcal{Q}|=H_t}} \sum_{i\in\mathcal{Q}} s_i^{(t)}. \tag{10}$$

The router scores determine only which positions are queried. For each $i \in \mathcal{Q}_{t+1}$, the backbone recomputes its representation against the persistent KV cache and obtains a query-updated token distribution $\widetilde{p}_i^{(t+1)}$. Because the remaining representations are cached, this sparse query is not equivalent to full recomputation.

Under confidence-based greedy single-token decoding, the backbone computes

$$\begin{aligned} \widehat{v}_i^{(t+1)} &= \operatorname*{arg\,max}_{v\in\mathcal{V}} \widetilde{p}_i^{(t+1)}(v), \\ c_i^{(t+1)} &= \widetilde{p}_i^{(t+1)}\left(\widehat{v}_i^{(t+1)}\right), \\ i_{t+1} &= \operatorname*{arg\,max}_{i\in\mathcal{Q}_{t+1}} c_i^{(t+1)}, \qquad \mathcal{A}_{t+1} = \{i_{t+1}\}. \end{aligned} \tag{11}$$

It replaces the mask at i_{t+1} with $\widehat{v}_{i_{t+1}}^{(t+1)}$ and updates

$$\mathcal{M}_{t+2} = \mathcal{M}_{t+1} \setminus \{i_{t+1}\}. \tag{12}$$

All other positions in $\mathcal{Q}_{t+1} \setminus \{i_{t+1}\}$ remain masked, while positions outside $\mathcal{Q}_{t+1}$ continue to reuse their cached representations. The loop repeats between scheduled broad-recomputation steps until the active generation region is resolved. No reference trajectory or oracle computation is used at inference time.

5 Experiments

We have shown in Figure 1 that trajectory fidelity is strongly associated with downstream performance. Based on this observation, we ask three questions: whether trajectory-guided querying yields end-to-end gains across models and tasks, how efficiently it allocates computation relative to existing cache-based methods, and which router design choices account for its performance.

5.1 Experimental Setup

Models. We evaluate four representative diffusion language models: LLaDA-Base, LLaDA-Instruct, Dream-Base, and Dream-Instruct. For the instruction-tuned models, we do not apply chat templates. In our preliminary evaluation, chat templates improve performance on some benchmarks but degrade it on others, introducing an additional source of variation unrelated to the inference method itself. We therefore use a unified prompt format across all models.

Table 1: **End-to-end quality and inference efficiency.** We evaluate four dLLM backbones on five downstream benchmarks with batch size 16. Each method uses its adopted operating configuration. TPS denotes tokens per second.

Model	Method	MATH-500	BBH-125	GSM8K	MBPP	HumanEval	Avg.↑	TFLOPs ↓	TPS ↑	Speedup ↑
LLaDA Base	Full Denoising	30.1	44.7	70.7	39.6	32.9	100.0	5931	53	1.0×
	dLLM-Cache	29.5	43.5	70.2	37.8	32.9	98.1	1605	65	1.23×
	d2Cache	29.0	44.9	68.5	38.8	29.3	96.6	804	33	0.6×
	TRACEQUERY	28.1	39.7	67.0	34.2	29.3	91.0	415	112	2.1×
LLaDA Instruct	Full Denoising	34.9	58.0	76.3	36.8	35.4	100.0	5828	53	1.0×
	dLLM-Cache	34.3	56.4	73.4	28.4	36.6	94.9	1590	65	1.23×
	d2Cache	34.9	40.6	52.8	35.5	36.6	83.0	794	33	0.6×
	TRACEQUERY	33.7	49.1	75.1	35.4	35.4	94.7	409	112	2.1×
Dream Base	Full Denoising	39.8	43.2	71.6	55.0	37.8	100.0	5554	56	1.0×
	dLLM-Cache	39.1	39.2	74.1	54.6	34.8	97.7	1252	84	1.5×
	d2Cache	38.2	41.1	72.4	55.6	38.4	99.3	753	36	0.6×
	TRACEQUERY	37.2	40.3	72.0	52.0	30.5	93.8	388	119	2.1×
Dream Instruct	Full Denoising	41.3	49.2	79.2	52.8	49.4	100.0	5457	56	1.0×
	dLLM-Cache	41.8	58.3	83.7	54.6	45.7	104.5	1243	84	1.5×
	d2Cache	40.0	47.2	75.3	53.0	46.6	96.4	743	36	0.6×
	TRACEQUERY	38.6	50.1	80.6	50.0	47.6	98.1	387	119	2.1×

Benchmarks. We evaluate reasoning, knowledge, and code generation using five benchmarks: MATH-500(512), BBH-125(256), GSM8K(256), HumanEval(512), and MBPP(512). All benchmarks use the same precision, decoding configuration, and hardware environment.

Baselines. We compare TRACEQUERY against full denoising and two selective cache-based acceleration methods: dLLM-Cache, and d2Cache. The controlled comparison varies the operating point of each method to obtain matched-quality and matched-computation settings, while the final end-to-end evaluation uses the adopted configuration of each method.

Efficiency metrics. We report downstream task score together with total computation in TFLOPs, an wall-clock inference time. For batch experiments, we additionally report the number of samples that can be processed under the same resource budget. TFLOPs measure the algorithmic computation footprint, while wall-clock time captures the practical execution cost of each method.

5.2 END-TO-END EVALUATION

We begin with the main result: end-to-end performance across four backbone models and five benchmarks. Each method uses its adopted operating configuration. We compare full denoising, dLLM-Cache, d2Cache, and TRACEQUERY for each backbone.

Table 1 reports downstream task performance together with total computation, and wall-clock inference time. Full denoising provides the quality reference, while the three cache-based methods represent different strategies for reducing repeated computation through approximate reuse and selective refresh.

Across the four backbones, TraceQuery trades a modest drop in accuracy for substantially lower compute: it retains 90-98% of full-denoising accuracy while using roughly 14× fewer TFLOPs per document, the lowest among the caching methods compared. Compared with the selective cache-based baselines, the advantage is particularly pronounced in TFLOPs and wall-clock time, consistent with the small query footprint observed in the controlled comparison. The improvement is also maintained across both base and instruction-tuned models, suggesting that the learned querying strategy is not specific to a single model variant or task family.

The controlled comparison in Section 5.3 examines how computation allocation contributes to these results.

5.3 CONTROLLED EFFICIENCY COMPARISON

To examine this allocation directly, we compare TRACEQUERY against dLLM-Cache and d2Cache under controlled resource budgets. This experiment varies the operating point of each method to

isolate how efficiently fresh computation is allocated. We consider two complementary settings: matched quality and matched computation.

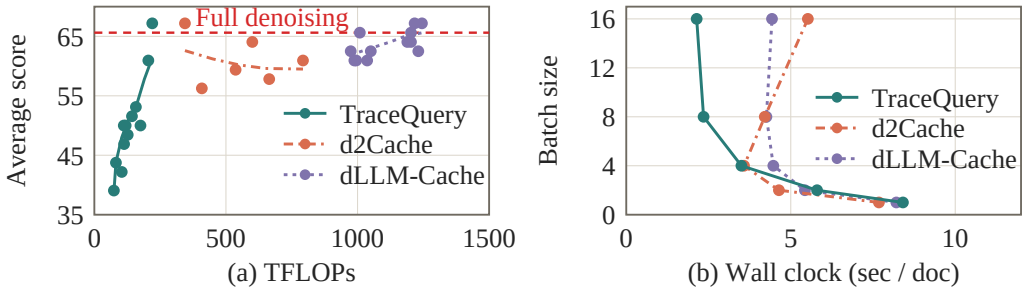


Figure 3: **Quality–compute and batch-scaling comparison. (a)** Average score across benchmarks as a function of total computation, measured in TFLOPs. The red dashed line marks the score of full denoising without caching. TRACEQUERY reaches higher scores under comparable compute budgets. **(b)** Supported batch size as a function of wall-clock time per document. TRACEQUERY supports larger batches at comparable latency, demonstrating improved batch-scaling efficiency.

Matched-quality computation. Figure 3(a) compares the amount of computation required to reach the same downstream quality. The x-axis reports total computation in TFLOPs. For each method, we adjust its computation budget and evaluate the resulting average performance across the all benchmarks.

At the same quality level, TRACEQUERY requires substantially less fresh computation than other selective cache-based baselines. This difference follows from the allocation objective. dLLM-Cache, and d2Cache spend computation maintaining or refreshing a comparatively large set of cached representations, whereas TRACEQUERY uses a small query horizon and concentrates fresh computation on candidates expected to matter for upcoming decoding decisions. As a result, TRACEQUERY reaches the same quality with a smaller computation footprint.

Matched-compute batch efficiency. We next study how the per-sample query footprint affects batched inference. For sample b, let $\mathcal{Q}_t^{(b)}$ denote the positions requiring fresh computation at denoising step t. Batched execution combines these positions into a shared query set: $\mathcal{U}_t(B) = \bigcup_{b=1}^{B} \mathcal{Q}_t^{(b)}$. As batch size increases, different samples contribute different queried positions, causing this union to grow. Methods with a large per-sample refresh budget can therefore quickly cover most of the generation region, making sparse batched inference increasingly close to dense recomputation.

In contrast, TRACEQUERY uses a much smaller query horizon, so the union grows substantially more slowly and sparse computation remains effective at larger batch sizes. Under the same computation budget, TRACEQUERY therefore supports substantially larger batches than the competing cache methods. The advantage is modest in the single-sample setting but grows rapidly with batch size, showing that reducing the per-sample query footprint provides an additional systems benefit beyond reducing FLOPs alone.

Together, the two controlled comparisons show that trajectory-guided querying improves both *quality efficiency*, by achieving the same quality with less computation, and *batch efficiency*, by accommodating more samples under the same resource budget.

5.4 ABLATION STUDY

We then examine the router design underlying these results through the two-stage ablation shown in Figure 4. All results are averaged across LLaDA-Base, LLaDA-Instruct, Dream-Base, and Dream-Instruct.

Stage I: Router design. We tune the router on full-denoising trajectories, varying one design choice at a time and measuring Recall@5 over 17,670 held-out positions. Features matter most: adding geometry and confidence raises Recall@5 from 0.51 (attention alone) to 0.61 and 0.78, re-

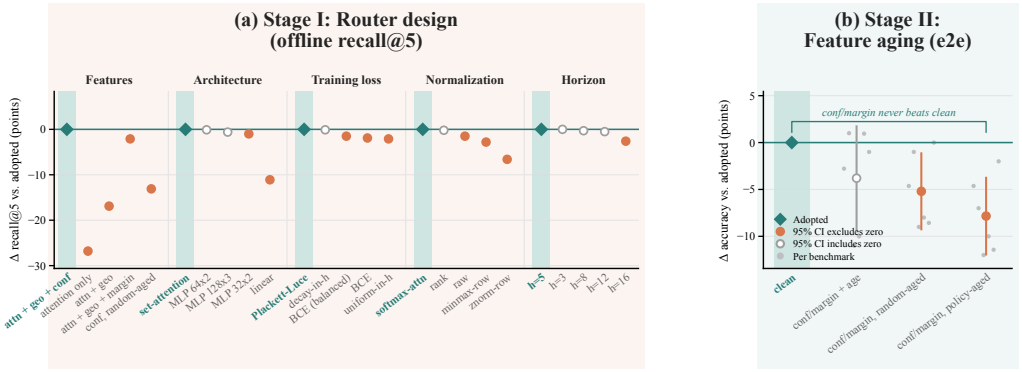


Figure 4: **Two-stage ablation of TRACEQUERY.** (a) Stage I selects the router configuration by varying the feature combination, router architecture, ranking loss, normalization, and query horizon during router training. (b) Stage II fixes the selected configuration and evaluates alternative feature groups under closed-loop cached decoding. Results are averaged across four models and reported as accuracy change relative to the adopted configuration. Bars show paired 95% bootstrap confidence intervals; filled markers indicate intervals excluding zero, and gray dots denote per-benchmark changes.

spectively. Stale confidence loses 13 points, while nonlinear architectures differ by less than 0.6 points. Loss and normalization choices are similarly stable, and performance holds for $H = 3$–12; we adopt $H = 5$. We train on mixed tasks and retrain for each backbone. Stage II tests whether confidence remains useful under cached inference.

Stage II: Closed-loop feature evaluation. Stage I evaluates the router under offline trajectory supervision, but some features may behave differently once the model operates with cached states. We therefore fix all design choices selected in Stage I and evaluate alternative feature groups under closed-loop cached decoding.

For each benchmark, we sample 100 prompts and measure accuracy relative to the adopted router. We test whether confidence and margin remain useful when cached, since training observes fresh values but inference updates them only on re-query. We compare four feature sets: **Clean** uses only attention and geometry; **Policy-aged** carries values forward according to the deployment refresh schedule; **Random-aged** draws values from earlier steps; and **Conf/margin + age** adds the time since each position's last refresh. Closed-loop results reverse the offline ranking. We adopt Clean: conf/margin + age changes accuracy by -3.8 points (CI includes zero), while random-aged and policy-aged significantly reduce it by 5.2 and 7.8 points, respectively. Confidence is highly predictive offline, yet every cached variant underperforms without it, making closed-loop evaluation essential for feature selection.

6 CONCLUSION

We formulate efficient dLLM inference as *trajectory-guided querying*: allocating limited computation to positions likely to matter for upcoming decoding decisions. TRACEQUERY learns this allocation from full-denoising trajectories, then uses inexpensive decoding-state signals to select a small query set at inference time. The backbone's updated predictions determine which position and token to commit. Across LLaDA and Dream models, TRACEQUERY improves the quality–computation trade-off over cache-based baselines, with further gains in batched inference from its small per-sample query footprint.

AI USE STATEMENT

In this work, we used generative AI tools only for language polishing, grammar correction, improving the clarity of author-written text, and identifying potentially relevant related work. We did not

use generative AI tools to generate research ideas, formulate mathematical claims, design experiments, implement methods, or analyze results. All AI-assisted content, including suggested related work, was reviewed, verified, and edited by the authors. We take responsibility for the final content of this work, including all text, claims, and artifacts.

REFERENCES

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

Jacob Austin, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. Structured denoising diffusion models in discrete state-spaces. *CoRR*, abs/2107.03006, 2021. URL https://arxiv.org/abs/2107.03006.

Minsoo Cheong, Donghyun Son, Woosang Lim, and Sungjoo Yoo. EntropyCache: Decoded token entropy guided KV caching for diffusion language models. *arXiv preprint arXiv:2603.18489*, 2026. URL https://arxiv.org/abs/2603.18489.

Zhanqiu Hu, Jian Meng, Yash Akhauri, Mohamed Abdelfattah, Jae-sun Seo, Zhiru Zhang, and Udit Gupta. Flashdlm: Accelerating diffusion language model inference via efficient kv caching and guided diffusion. In *International Conference on Learning Representations*, volume 2026, pp. 153529–153544, 2026.

Metod Jazbec, Theo X Olausson, Louis Béthune, Pierre Ablin, Michael Kirchhof, João Monteiro, Victor Turrisi, Jason Ramapuram, and Marco Cuturi. Learning unmasking policies for diffusion language models. *arXiv preprint arXiv:2512.09106*, 2025.

Yuchu Jiang, Yue Cai, Xiangzhong Luo, Jiale Fu, Jiarui Wang, Chonghan Liu, and Xu Yang. d^2Cache: Accelerating diffusion-based LLMs via dual adaptive caching. In *International Conference on Learning Representations*, 2026. URL https://arxiv.org/abs/2509.23094.

Inception Labs, Samar Khanna, Siddhant Kharbanda, Shufan Li, Harshit Varma, Eric Wang, Sawyer Birnbaum, Ziyang Luo, Yanis Miraoui, Akash Palrecha, et al. Mercury: Ultra-fast language models based on diffusion. *arXiv preprint arXiv:2506.17298*, 2025.

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

Xuejie Liu, Yap Vit Chun, Yitao Liang, and Anji Liu. Lookahead path likelihood optimization for diffusion llms. *arXiv preprint arXiv:2602.03496*, 2026a.

Zhiyuan Liu, Yicun Yang, Yaojie Zhang, Junjie Chen, Chang Zou, Qingyan Wei, Shaobo Wang, Yichen Zhu, and Linfeng Zhang. dLLM-Cache: Accelerating diffusion large language models with adaptive caching. In *International Conference on Machine Learning*, 2026b. URL https://arxiv.org/abs/2506.06295.

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

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

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

Jingyang Ou, Shen Nie, Kaiwen Xue, Fengqi Zhu, Jiacheng Sun, Zhenguo Li, and Chongxuan Li. Your absorbing discrete diffusion secretly models the conditional distributions of clean data. In *International Conference on Learning Representations*, volume 2025, pp. 64972–65009, 2025.

Tianxiang Pan, Baitao Gong, Mo Guang, Hongwei Yong, Tianpeng Jiang, Yaqian Li, Zheng Cao, and Kaiwen Long. Trajectory-level speculative decoding for diffusion language models. *arXiv preprint arXiv:2608.27514*, 2026.

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

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

Chengyue Wu, Hao Zhang, Shuchen Xue, Zhijian Liu, Shizhe Diao, Ligeng Zhu, Ping Luo, Song Han, and Enze Xie. Fast-dllm: Training-free acceleration of diffusion llm by enabling kv cache and parallel decoding. In *International Conference on Learning Representations*, volume 2026, pp. 57027–57051, 2026.

Jiacheng Ye, Zhihui Xie, Lin Zheng, Jiahui Gao, Zirui Wu, Xin Jiang, Zhenguo Li, and Lingpeng Kong. Dream 7B: Diffusion large language models. *arXiv preprint arXiv:2508.15487*, 2025. URL https://arxiv.org/abs/2508.15487.

A Detailed Computation and Memory Analysis

Computation Complexity Let S denote the total sequence length visible to attention, B the size of the active generation region, T the number of denoising steps, H the query horizon, R the number of full refresh steps, and L the number of Transformer layers.

Under single-token decoding, a regular sparse step recomputes at most

$$q = H + 1 \tag{13}$$

positions, consisting of H future-query candidates and one recently committed position required by delayed cache admission. Near the end of a generation block, q may be smaller when fewer than H masked positions remain.

Querying q positions against a KV cache of length S requires $\mathcal{O}(LqS)$ attention computation, whereas refreshing all B positions in the active generation region requires $\mathcal{O}(LBS)$. Ignoring the one-time cache initialization, the total attention cost is therefore

$$C_{\textsc{TraceQuery}} = \mathcal{O}\left(LS\left[RB + (T - R)(H + 1)\right]\right). \tag{14}$$

By comparison, full recomputation at every denoising step requires

$$C_{\text{Full}} = \mathcal{O}(LSTB). \tag{15}$$

For single-block diffusion, $B = T = N$, where N denotes the generation length. Assuming $S = \mathcal{O}(N)$ and treating H and R as bounded constants,

$$C_{\textsc{TraceQuery}} = \mathcal{O}(LN^2), \qquad C_{\text{Full}} = \mathcal{O}(LN^3). \tag{16}$$

Thus, selective querying reduces the dominant attention-computation scaling from cubic to quadratic in generation length.

Concrete Decoding Configurations Our default experiments use a query horizon of $H = 5$. For single-block generation with $N = 256$, we perform $R = 16$ global refreshes. A regular sparse step therefore recomputes at most six positions: five future-query candidates and one recently committed position. Using the number of attention query positions as a computation proxy, the relative attention workload is approximately

$$\frac{16 \times 256 + (256 - 16) \times 6}{256 \times 256} \approx 8.45\%. \tag{17}$$

This quantity measures the reduction in attention-query computation and should not be interpreted as wall-clock speedup. Practical runtime additionally depends on projection and MLP computation, cache operations, router execution, kernel efficiency, and hardware utilization.

For a length-256 generation divided into 16 blocks, each active block contains 16 positions. We perform no additional periodic full refresh within each block and use the same query horizon $H = 5$. The effect of query horizon and refresh schedule is evaluated empirically in Section 5.

Table 2: Persistent KV-cache memory for the illustrative BF16 LLaDA-8B configuration under batch size 1. Relative memory denotes model weights plus KV cache divided by model-weight memory alone.

Cached Length S	KV Cache	Relative Memory
256	128 MiB	$\sim 1.008\times$
512	256 MiB	$\sim 1.016\times$
1024	512 MiB	$\sim 1.034\times$
2048	1 GiB	$\sim 1.067\times$

Persistent Memory Consumption TRACEQUERY maintains a persistent KV cache for the complete sequence. Let d_{KV} denote the dimensionality of the stored KV representation per token, b the number of bytes per element, and B_{batch} the batch size. The persistent KV-cache memory is

$$M_{\mathrm{KV}} = 2B_{\mathrm{batch}} L S d_{\mathrm{KV}} b, \tag{18}$$

where the factor of two accounts for keys and values. Thus,

$$M_{\mathrm{KV}} = \mathcal{O}(L S d_{\mathrm{KV}}), \tag{19}$$

which scales linearly with sequence length.

For illustration, assuming a BF16 LLaDA-8B configuration with $L = 32$, $d_{\mathrm{KV}} = 4096$, and batch size 1, the model parameters occupy approximately

$$M_{\mathrm{model}} \approx 8 \times 10^9 \times 2 = 16 \text{ GB}. \tag{20}$$

For a cached sequence of length $S = 256$, the corresponding KV cache requires

$$M_{\mathrm{KV}} = 2 \times 32 \times 256 \times 4096 \times 2 \approx 128 \text{ MiB}, \tag{21}$$

corresponding to less than 1% additional persistent storage relative to the BF16 model weights.

B DECODING SIGNAL DEFINITIONS

This appendix provides the detailed definitions of the decoding signals introduced in Section 4.3. Let $i \in \mathcal{M}_t$ denote a still-masked candidate position and let $\mathcal{R}_t \subseteq \mathcal{U}_t$ denote the set of recently unmasked positions.

Confidence. Given the token probability distribution at position i, confidence is defined as

$$c_i^{(t)} = \max_v p_\theta \left(v \mid x^{(t)} \right)_i. \tag{22}$$

Probability margin. Let $p_{i,(1)}^{(t)}$ and $p_{i,(2)}^{(t)}$ denote the largest and second-largest token probabilities at position i. The probability margin is

$$m_i^{(t)} = p_{i,(1)}^{(t)} - p_{i,(2)}^{(t)}. \tag{23}$$

Mask density. For a local neighborhood $\mathcal{W}_i$ centered at candidate i, mask density is defined as

$$\rho_i^{(t)} = \frac{1}{|\mathcal{W}i|} \sum j \in \mathcal{W}_i \mathbf{1} \left[j \in \mathcal{M}_t \right]. \tag{24}$$

Relative distance. Let L be the generation block size. The distance from candidate i to recently unmasked positions is defined as

$$d_i^{(t)} = \frac{\min_{j \in \mathcal{R}_t} |i - j|}{L}, \tag{25}$$

and is normalized by the active generation length when required.

Last-layer attention. Let $A_{\ell,h}^{(t)}(j,i)$ denote the attention from position j to position i at layer ℓ and head h. The last-layer attention signal is

$$a_{i,\text{last}}^{(t)} = \text{Agg}_{j \in \mathcal{R}t,\, h}\, AL, h^{(t)}(j,i), \tag{26}$$

where L denotes the final Transformer layer.

Full-layer attention. The full-layer attention signal aggregates the same dependency across Transformer layers:

$$a_{i,\text{full}}^{(t)} = \text{Agg}_{\ell,\, j \in \mathcal{R}t,\, h}\, A\ell, h^{(t)}(j,i). \tag{27}$$

C Inference and Training algorithm

Algorithm 1 TraceQuery Router Training

Require: Training samples $\mathcal{X}$, diffusion model p_θ, router f_ϕ, query horizon H

$\mathcal{D} \leftarrow \emptyset$
for all $x \in \mathcal{X}$ **do**
  Run full-denoising decoding with one token unmasked per step
  Record teacher trajectory $\tau = (i_0, i_1, \ldots, i_{T-1})$
  for $t = 0, \ldots, T-2$ **do**
    Collect signals for all $i \in \mathcal{M}_{t+1}$
    Normalize last-layer attention with SOFTMAX
    Store $\{\mathbf{z}_i^{(t)}, g_i^{(t)}\}_{i \in \mathcal{M}_{t+1}}$ in $\mathcal{D}$
  end for
end for
repeat
  Sample a decoding state t from $\mathcal{D}$
  for all $i \in \mathcal{M}_{t+1}$ **do**
    $s_i^{(t)} \leftarrow f_\phi(\mathbf{z}_i^{(t)})$
  end for
  $\mathcal{L} \leftarrow 0$
  for $j = 1, \ldots, H$ **do**
    $\mathcal{C}_{t,j} \leftarrow \{i \in \mathcal{M}_{t+1} : g_i^{(t)} \geq j\}$
    $i_{(j)} \leftarrow$ teacher's j-th future unmask
    $\mathcal{L} \leftarrow \mathcal{L} - \log \dfrac{\exp(s_{i_{(j)}}^{(t)})}{\sum_{i \in \mathcal{C}_{t,j}} \exp(s_i^{(t)})}$
  end for
  Update ϕ using $\nabla_\phi \mathcal{L}$
until convergence
return trained router f_ϕ

Algorithm 2 TRACEQUERY Inference

Require: Diffusion model p_θ, router f_ϕ, query horizon H, refresh schedule $\mathcal{R}$

Initialize masked set $\mathcal{M}_0$ and perform a full forward pass
Cache all KV states; set $\mathcal{Q}_0 \leftarrow \mathcal{M}_0$
for $t = 0, 1, \ldots$ **while** $\mathcal{M}_t \neq \emptyset$ **do**
    if $t \in \mathcal{R}$ **then**
        $(P_t, A_t, \mathrm{KV}) \leftarrow \textsc{FullForward}(\mathcal{M}_t, \mathrm{KV})$
        $\mathcal{E}_t \leftarrow \mathcal{M}_t$
    else
        $(P_t, A_t, \mathrm{KV}) \leftarrow \textsc{QueryForward}(\mathcal{Q}_t, \mathrm{KV})$
        $\mathcal{E}_t \leftarrow \mathcal{Q}_t$
    end if
    $i_t \leftarrow \arg\max_{i \in \mathcal{E}_t} \textsc{Confidence}(P_t(i))$
    Commit position i_t and update $\mathcal{M}_{t+1} \leftarrow \mathcal{M}_t \setminus \{i_t\}$
    if $\mathcal{M}_{t+1} = \emptyset$ **then**
        break
    end if
    for all $i \in \mathcal{M}_{t+1}$ **do**
        $\mathbf{z}_i^{(t)} \leftarrow [a_{i,\mathrm{last}}^{(t)}, \rho_i^{(t)}, d_i^{(t)}]$
        $s_i^{(t)} \leftarrow f_\phi(\mathbf{z}_i^{(t)})$
    end for
    $\mathcal{Q}_{t+1} \leftarrow \textsc{TopH}(\{s_i^{(t)}\}_{i \in \mathcal{M}_{t+1}}, H)$
end for
return generated sequence