DENOISING SURFACE: MODELING AND PREDICTING INFERENCE COST FOR DIFFUSION LLM SERVING

Anonymous authors
Paper under double-blind review

ABSTRACT

As diffusion large language models (dLLMs) become more capable, they are moving from research settings to real-world *serving*, where request management (such as scheduling and resource allocation) relies on accurate estimation of per-request inference cost. However, common cost proxies fall short for dLLMs: output length ignores that one forward pass can unmask multiple tokens, and denoising-step count ignores the *heterogeneous* per-step costs. We observe that the block-autoregressive generation mechanism induces a two-dimensional execution structure over output blocks and within-block denoising steps, whereas these proxies collapse it into a scalar, discarding information essential for characterizing the cost. Motivated by this insight, we propose the Denoising Workload Surface (DWS), which preserves this two-dimensional block-step structure as a probability surface to weight the heterogeneous per-step costs. We then design a coarse-to-fine training scheme that enables a lightweight prompt-only predictor to accurately predict the complex DWS. This predictor runs efficiently even on a single CPU core, avoiding GPU contention with the serving model. Since DWS decouples request-dependent execution behavior from deployment-specific cost factors, the predictor transfers across hardware configurations without retraining. In *real-world* serving experiments, DWS reduces cost-prediction error by up to $2.50\times$ over scalar-based predictors, while the DWS-guided shortest-job-first scheduler reduces end-to-end latency by up to $1.92\times$ for online chatbots. Code at: `https://anonymous.4open.science/r/DS1`.

1 INTRODUCTION

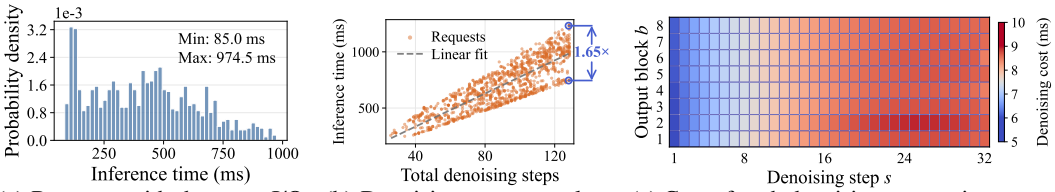
Diffusion Large Language Models (dLLMs) (Nie et al., 2025; Gong et al., 2025; Li et al., 2026) have attracted growing attention in recent years for their parallel decoding mechanism. Unlike autoregressive (AR) LLMs, which iteratively predict the next token and generate only *one* token per decoding forward pass, dLLMs can jointly predict multiple masked positions and therefore unmask *multiple* tokens within a single forward pass (i.e., a *denoising step*) (Wu et al., 2026b; Wang et al., 2026b). Importantly, recent dLLMs are competitive with small- and mid-scale autoregressive LLMs across various tasks (Nie et al., 2025; Gong et al., 2025; Bie et al., 2025). Together with parallel decoding, these developments make dLLMs a promising alternative to AR LLMs for efficient serving (Wang et al., 2026b). Recently, popular LLM serving frameworks such as SGLang and vLLM have begun to support dLLM inference, signaling a shift from research settings to *real-world serving* (SGLang, 2025; vLLM, 2026).

Efficient LLM serving relies on effective request management, such as request scheduling, load balancing, and resource allocation. For example, prior work on AR LLM serving commonly adopts shortest-job-first (SJF) scheduling to prevent long requests from blocking short ones, thereby reducing average latency and time-to-first-token (TTFT) (Qiu et al., 2024; Shahout et al., 2025; Zheng et al., 2026). Such request management relies on estimating how much computation each request will consume, making accurate cost estimation an important signal. This raises our first question:

***Q1 (cost modeling)**: How can we accurately quantify the inference cost of a dLLM request?*

Output length? The question *Q1* is well studied for AR LLMs, where each generated token corresponds to one decoding forward pass, making output length a natural proxy for request workload. Accordingly, the SJF-based studies discussed above predict output length to estimate request cost. However, this simple abstraction fundamentally **breaks down** for dLLMs, where a single denoising step can unmask a *variable* number of tokens. To empirically demonstrate this breakdown, we use LLaDA2.0-mini (Bie et al., 2025) and serve each request individually to avoid batching effects. As shown in Figure 1a, even requests with the same input and output lengths exhibit substantially different (up to $11.47\times$) inference times, so output length is *no longer a reliable proxy* for dLLM workload.

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



(a) Requests with the same I/O lengths exhibit various costs.

(b) Denoising steps correlate with cost, but still inaccurate.

(c) Cost of each denoising step varies across its positions, rather than being homogeneous.

Figure 1: **Motivation: common scalar proxies cannot faithfully characterize dLLM inference cost.** Setting: LLaDA2.0-mini serves *equal-length* prompts *individually*.

Denoising steps? A natural alternative is the number of denoising steps, which directly counts the decoding forward passes. As shown in Figure 1b, although it correlates with the inference time, requests with the same number of steps still vary considerably in inference cost. Figure 1c reveals the reason: denoising steps at different positions can incur different computational costs[1], especially across within-block step indices. However, simply counting the denoising steps treats *heterogeneous* steps equally, making the total step count a reasonable but still *inaccurate* proxy for dLLM workload.

Our analysis and solution to Q1. Popular serving-oriented dLLMs typically generate in a block-autoregressive manner[2], where each output block undergoes multiple denoising steps (detailed in Section 2). This process naturally forms a **two-dimensional** execution structure indexed by the output block and the within-block denoising step. However, both output length and total denoising-step count compress this structure into a *scalar*, discarding information essential for cost estimation. Motivated by this insight, we introduce the **D**enoising **W**orkload **S**urface (DWS), which preserves this structure by decomposing the denoising process into *fine-grained* denoising cells. Then we weight the costs of each cell, thereby providing a more faithful characterization of dLLM inference cost.

How to predict? A further challenge is to predict DWS before the request inference. A straightforward approach is to adapt existing AR-LLM workload predictors, which typically use a lightweight encoder (e.g., BERT) for prompt encoding, followed by a small regression head (e.g., an MLP) to predict the workload. However, this design has two drawbacks for dLLM serving. *First*, DWS is a complex two-dimensional target rather than a scalar, so a small regression head can struggle to learn it directly from the prompt. *Second*, although the encoder is lightweight compared with the serving model, it still requires GPU acceleration for efficient prediction, *competing* with the serving model for GPU resources. Moving it to CPUs can avoid GPU contention but incurs high prediction latency, delaying cost estimates and potentially leading to suboptimal serving decisions. This motivates our second question:

> ***Q2 (cost prediction)**: How can we design a CPU-efficient predictor that enables accurate cost estimation for dLLM requests by capturing their structured inference workload?*

Our solution to Q2. We prove that DWS can be factorized along its two dimensions *without* loss of expressiveness, and design a prompt-only predictor that separately predicts the corresponding distributions and combines them to *reconstruct* the two-dimensional DWS. Since learning the fine-grained DWS structure from prompts remains challenging, we further design a *progressive* training scheme with *multi-horizon* workload supervision to gradually guide the encoder from coarse workload patterns to the fine-grained DWS. Finally, we carefully quantize the predictor to trade a *negligible* loss in accuracy for substantially higher efficiency even deployed on a single CPU core.

Importantly, our formulation *decouples* request-dependent execution behavior from deployment-specific cost factors. The former is captured by our accurate prompt-only DWS predictor, while the latter are automatically collected and refreshed through online profiling. This separation allows the predictor to **transfer** across hardware deployments *without* collecting new training data or retraining.

We evaluate both the predictor and its downstream serving performance. *For the predictor*, DWS achieves up to $2.50\times$ lower cost MAE than common scalar-based predictors. Its quantization further improves the prediction throughput by $2.32\times$ with only $< 1\%$ accuracy degradation. *For serving*, our DWS-guided SJF scheduler delivers substantial gains *across* different scenarios, datasets, model scales, and hardware without retraining. For online chatbots, it reduces end-to-end latency and TTFT

[1]We find that this heterogeneity stems from multiple sources, including the model architecture and context length (detailed in Appendix C). The main text focuses on how to model and predict the resulting complex cost.

[2]We find that *all* dLLMs currently natively supported by SGLang and vLLM follow this manner.

by up to $1.92\times$ and $2.22\times$, respectively, over the default first-come-first-served (FCFS) scheduler. For offline data generation, it accelerates sample collection by up to $2.73\times$ over FCFS.

Contributions. In summary, we make the following contributions:

- We conduct a fine-grained attribution of dLLM inference cost and reveal the heterogeneity of per-step costs. Based on these insights, we introduce DWS and a corresponding cost model that preserves the two-dimensional structure for *accurate* estimation of inference cost.
- We design a CPU-efficient prompt-only DWS predictor that accurately predicts the structured workload of dLLM inference. Moreover, our *decoupled* design enables this predictor to transfer across deployments without collecting new training data or retraining.
- We show that DWS improves *both* cost prediction accuracy and serving efficiency, enabling the DWS-guided scheduler to reduce online latency and accelerate offline data generation.

2 Preliminaries

In this section, we introduce the key mechanisms widely adopted by modern serving-oriented dLLMs.

Block-autoregressive generation. Modern dLLMs commonly generate output blocks autoregressively (DiffusionGemma Team et al., 2026; Bie et al., 2025; Song et al., 2025), as shown in Figure 2. Within each output block, the model iteratively denoises masked tokens in parallel, where each denoising step performs one forward pass and reveals one or more tokens (Arriola et al., 2025). The block size B is typically configured at deployment time and *kept fixed* during inference, e.g., $B = 32$ for LLaDA2.0. Once a block is fully denoised, generation terminates if an `[EOS]` token has been produced; otherwise, a new masked block is appended and denoised. Compared with early fixed-canvas diffusion, this design supports variable-length generation and avoids wasted computation on padded positions (Arriola et al., 2025). Thus, the generation workload substantially varies across requests, making cost-aware request scheduling beneficial.

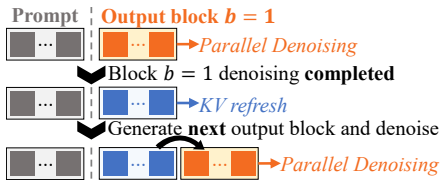


Figure 2: Block-autoregressive generation with KV refresh.

KV refresh. DLLMs can cache the key/value (KV) states of completed context to avoid redundant computation. However, bidirectional attention within a block keeps changing its KV states during denoising. Therefore, once a block is fully denoised, its *own* KV states need to be refreshed and cached.

3 Cost Modeling

3.1 Anatomy of dLLM Inference Cost

We begin with an empirical study of where dLLM inference time is actually spent.

Setting. We sample 20K requests from LMSYS-Chat-1M (Zheng et al., 2024), a real-world dataset. Each is served by LLaDA2.0-mini *individually* to isolate its *own* cost from batching effects.

Cost components. We analyze their latency breakdown to determine which stages to include in our cost model. We retain a stage if it satisfies *either* of two criteria: (i) it dominates the total latency; or (ii) it varies substantially across requests, and thus affects their relative ordering. Figure 3 summarizes the latency shares across these requests. Overall, denoising dominates the total latency (avg: 80.8%), whereas prefill and KV refresh take up smaller average shares but vary substantially across requests ($5.8 \pm 14.4\%$ and $6.7 \pm 4.7\%$). In contrast, other stages are both small and stable, and thus have limited impact. Therefore, our cost model focuses on these three key components.

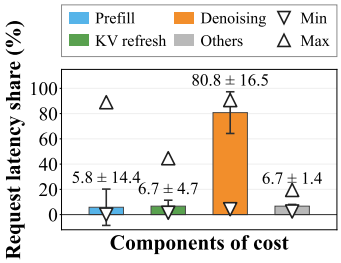


Figure 3: Per-request latency shares of inference stages (bars: mean ± std; markers: min/max).

We first model the *simpler* prefill and KV-refresh costs, deferring the *complex* denoising to Section 3.2.

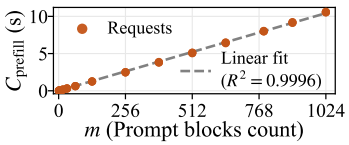
Prefill and KV refresh. These two components follow simple execution patterns: prefill processes *each* prompt block once, and KV refresh is performed once for *each* newly completed output block. These patterns naturally motivate *linear* cost models in the prompt-block count m and output-block count n, respectively. As shown in Figure 4, both linear models fit the measured costs *closely*

throughout the context range supported by LLaDA2.0 (up to 32K tokens, i.e., 1K blocks) with $R^2 \geq 0.999$ (detailed in Appendix B). We further study the higher-order terms introduced by attention computation under very long contexts in Appendix D.

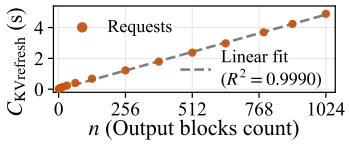
We therefore model these two components as $C_{\text{prefill}} \triangleq \alpha m$ and $C_{\text{KVrefresh}} \triangleq \beta n$, so the total inference cost can be modeled as:

$$C \triangleq \alpha m + \beta n + C_{\text{denoising}}. \tag{1}$$

The term $C_{\text{denoising}}$ dominates the total cost and is considerably harder to model, as we address below.



(a) C_{prefill} scales linearly with m.



(b) $C_{\text{KVrefresh}}$ scales linearly with n.

Figure 4: Linear fits for prefill and KV-refresh costs.

3.2 Modeling the Denoising Cost

Recall that Figure 1c shows that denoising steps at different execution positions differ in cost. Thus, accurately modeling $C_{\text{denoising}}$ requires preserving its *fine-grained* denoising structure.

A structured view of denoising. In dLLM inference, the number of output blocks n is bounded by the configured maximum output length, so $n \in \{1, \cdots, N_{\max}\}$. Let s_b denote the number of denoising steps required by the b-th output block b. Since each step unmasks at least one token, $s_b \in \{1, \cdots, B\}$, where B denotes the block size. Writing $c^\star_{b,s}$ for the *actual* cost of the denoising step s in output block b, the *actual* denoising cost of a request can be decomposed as $C^\star_{\text{denoising}} = \sum_{b=1}^{n} \sum_{s=1}^{s_b} c^\star_{b,s}$.

Here, although n and s_b vary across requests, their upper bounds ($N_{\max}$ and B, respectively) are fixed by the serving configuration. We therefore embed the denoising execution of each request into a fixed-size *denoising activity matrix* $A \in \{0,1\}^{N_{\max} \times B}$, where $A_{b,s} = 1$ iff the output *reaches* block b and performs *at least* s denoising steps within it, i.e., $A_{b,s} = \mathbb{1}\{b \leq n \wedge s \leq s_b\}$. This yields

$$C^\star_{\text{denoising}} = \sum_{b=1}^{N_{\max}} \sum_{s=1}^{B} c^\star_{b,s} \cdot A_{b,s}, \tag{2}$$

which clearly separates these two *orthogonal* factors: (i) the denoising activity carried by $A_{b,s}$, and (ii) the cost of each step carried by $c^\star_{b,s}$. Next, we turn to the cost of a single denoising step.

Modeling cell costs. We first examine how the cost of the same denoising cell varies across requests. Across the 20K traces detailed above, the 99th percentile (P99) coefficient of variation (CV) of $c^\star_{b,s}$ is only 11.08%, indicating that cell costs are fairly *stable* across requests. This suggests that a *shared* cell-cost matrix is a reasonable starting point. We further account for the residual variation induced by request-dependent factors. Mechanically, a denoising step at block b attends to the KV states of the preceding context, whose length is approximately $m + b - 1$ blocks, where m varies across requests. We therefore decompose the effects of the execution position (b, s) and the prompt length m:

$$K_{b,s}(m) \triangleq W_{b,s} + \delta(m - \bar{m}), \tag{3}$$

where $K(m) \in \mathbb{R}^{N_{\max} \times B}$ denotes the modeled cost of all cells for a request with m prompt blocks, and $W \in \mathbb{R}^{N_{\max} \times B}$ is the request-shared baseline cost matrix. For the effects of context, the output-side context growth is already indexed by b and thus absorbed into $W_{b,s}$, leaving only the prompt-side dependence, which is modeled by $\delta(m - \bar{m})$, a request-dependent prompt-length correction.

We jointly calibrate W and δ from request-level denoising latency (detailed in Appendix E). Across 5-fold validation on the 20K real traces, this model achieves $R^2 = 0.9956$ and MAPE $= 5.01\%$, showing an accurate approximation of denoising cost. We further study its residual in Appendix D.4.

Putting it together. Substituting $K_{b,s}(m)$ into Equation 2, the denoising cost can be modeled as:

$$C_{\text{denoising}} \triangleq \sum_{b=1}^{N_{\max}} \sum_{s=1}^{B} K_{b,s}(m) \cdot A_{b,s} = \langle K(m), A \rangle. \tag{4}$$

Moreover, the output blocks n is determined by A as the number of its active blocks, so $n = \sum_{b=1}^{N_{\max}} A_{b,1}$. Finally, combining the prefill, KV-refresh, and denoising components, the total cost is:

$$C = \alpha m + \beta \sum_{b=1}^{N_{\max}} A_{b,1} + \langle K(m), A \rangle. \tag{5}$$

Here, m is directly observed from the prompt, while α, β, and $K(\cdot)$ depend on the deployment configuration and can be obtained by profiling. Next, we turn to the *only* remaining unknown term A.

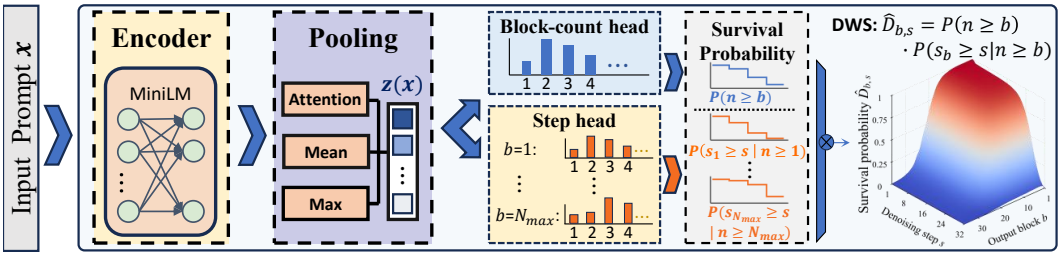


Figure 5: **Architecture of the DWS predictor.** An encoder extracts the representations of prompt x and aggregates into $z(x)$, followed by two separate heads for the output-block and denoising-step distributions. Then the corresponding survival probabilities are combined to reconstruct the DWS $\widehat{D}$. DWS shown on the right is a smoothed visualization of $\widehat{D}$ for the first prompt in LMSYS-Chat-1M.

4 Prediction of the Denoising Activity

4.1 Prediction Target

Equation 5 *naturally suggests* the prediction target, as derived below. First, the expected cost of x is:

$$\mathbb{E}[C \mid x] = \alpha m(x) + \beta \sum_{b=1}^{N_{\max}} D_{b,1}(x) + \langle K(m(x)), D(x) \rangle, \tag{6}$$

where $D(x) \triangleq \mathbb{E}[A \mid x] \in [0,1]^{N_{\max} \times B}$ is the only unknown term in $\mathbb{E}[C \mid x]$, which is *clearly decoupled* from the deployment-specific cost factors α, β, and $K(\cdot)$. Next, we focus on $D(x)$.

Since $A_{b,s}$ is binary, $D_{b,s}(x) = P(A_{b,s} = 1 \mid x)$ is the activity probability of denoising cell (b, s). Geometrically, $D(x)$ forms a *surface* over the block-step space, whose height at (b, s) is $D_{b,s}(x)$, with an example shown on the right of Figure 5. We call $D(x)$ the **D**enoising **W**orkload **S**urface (**DWS**).

Structure-preserving factorization. The decoding process imposes a *survival structure*: $A_{b,s+1} \leq A_{b,s}$ and $A_{b+1,1} \leq A_{b,1}$ hold point-wise. DWS inherits these constraints, and so *should* its prediction. However, directly predicting each cell independently may violate these constraints and produce structurally invalid activity patterns. A straightforward fix is to explicitly enforce them on the predictor outputs, but the predictor itself still does not capture the underlying structure.

Instead, we build this decoding process into the parameterization, making the constraints hold *by construction*. Specifically, cell (b, s) is active *if and only if* decoding reaches block b and this block runs at least s steps, i.e., $\{A_{b,s} = 1\} = \{n \geq b\} \cap \{s_b \geq s\}$. By the chain rule, DWS factorizes exactly as

$$D_{b,s}(x) = P(A_{b,s} = 1 \mid x) = P(n \geq b \mid x) \cdot P(s_b \geq s \mid n \geq b, x), \tag{7}$$

where both factors are survival probabilities and are thus non-increasing, satisfying these constraints *naturally*. We further prove that this factorization loses *no* expressiveness (Appendix L.3). Thus, DWS prediction decomposes into predicting two families of *unconstrained* distributions, as described below.

4.2 DWS Predictor

Following Equation 7, we predict the two survival factors separately. As shown in Figure 5, given a prompt x, we use a pretrained all-MiniLM-L12-v2 (Wang et al., 2020) encoder to encode its semantics and pool the token embeddings into a representation $\mathbf{z}(x)$. On top of $\mathbf{z}(x)$, a block-count head predicts $p_k^{\text{block}}(x) \triangleq P(n = k \mid x)$ for $k \in \{1, \ldots, N_{\max}\}$, while a step head predicts $p_{b,t}^{\text{step}}(x) \triangleq P(s_b = t \mid n \geq b, x)$ for each potential block b and step $t \in \{1, \ldots, B\}$. Given these distributions, we convert them into their corresponding survival probabilities and then reconstruct the predicted DWS as

$$\widehat{D}_{b,s}(x) = \left(\sum_{k=b}^{N_{\max}} p_k^{\text{block}}(x) \right) \cdot \left(\sum_{t=s}^{B} p_{b,t}^{\text{step}}(x) \right). \tag{8}$$

The next question is how to train this predictor, facing two challenges: (i) the pretrained encoder is optimized for *generic* semantic representations, whereas we ask it to represent the denoising process; and (ii) DWS is a two-dimensional structured target and is inherently *harder* to learn than a scalar workload. We therefore adopt a *two-stage* training scheme with *multi-horizon* workload supervision.

Coarse-to-fine two-stage training. *In Stage A*, we introduce multi-horizon workload supervision to adapt the encoder. Given a step horizon h, we truncate the denoising steps of every block at h, yielding the truncated workload $T^{(h)} \triangleq \sum_{b=1}^{n} \min(s_b, h) = \sum_{b=1}^{N_{max}} \sum_{s=1}^{h} A_{b,s}$. At $h = 1$,

$T^{(1)} = n$ focuses on the number of output blocks, while increasing h *progressively* exposes deeper within-block denoising behavior, until $T^{(B)}$ recovers the total denoising-step count. To illustrate the benefits, consider two denoising traces with the same total step count: τ_1 generates more blocks but requires fewer steps per block, whereas τ_2 does the opposite. At a small h, $T^{(h)}(\tau_1) > T^{(h)}(\tau_2)$, since the deep denoising steps of τ_2 are truncated. As h grows, these deeper steps are gradually exposed, and $T^{(h)}(\tau_2)$ approaches $T^{(h)}(\tau_1)$. Supervising the encoder with $h \in \{1, 4, 8, 12, \ldots, B\}$ thus yields a multi-scale signal that a single scalar target cannot provide, teaching it to represent the *coarse* denoising structure, which Stage B then resolves into the fine-grained DWS.

In Stage B, we train the block-count head and the denoising-step head on top of the adapted encoder by directly supervising the reconstructed DWS, the final workload target, rather than the two heads separately. Specifically, we minimize the Brier loss $\mathcal{L}_B = \frac{1}{N_{\max} B} \sum_{b,s} (\widehat{D}_{b,s}(x) - A_{b,s})^2$. Minimizing this objective *provably* controls the cost-estimation error under *all* deployments simultaneously (detailed in Proposition 1 in Section 5). Further training details are provided in Appendix H and our code.

Quantization. To run the predictor efficiently on CPUs, we quantize it to INT8, achieving $2.32\times$ speedup on a *single CPU core*, with only a 0.54% degradation in total-cost MAE (Appendix F).

Pipeline. Upon request arrival, the CPU-side predictor takes its prompt x and predicts its DWS $\widehat{D}(x)$, while $m(x)$ is obtained directly from the prompt length. Equation 6 then combines them with the profiled cost factors α, β, and $K(\cdot)$, yielding the predicted inference cost $\widehat{C}(x)$.

5 Discussion

Why do dLLMs need DWS? We divide the answer into three aspects. *First*, DWS is the right prediction target. By Equation 6, given the deployment-specific cost factors, $D(x)$ and the observed $m(x)$ can model the expected cost of *any* prompt x under *any* hardware deployment. In contrast, scalar targets are essentially *lossy* compressions of DWS: (i) output length keeps only $\mathbb{E}[n \mid x] = \sum_b D_{b,1}(x)$, losing the denoising structure in each block; (ii) total steps keeps only $\sum_{b,s} D_{b,s}(x)$, losing the heterogeneous cost of each step; (iii) direct cost relies on its training deployment and must be retrained once the cost factors change. Appendix L.1 further theoretically quantifies their bias. Moreover, a single DWS predictor can control the cost-estimation error for *all* deployments (proved in Appendix L.2):

Proposition 1 (Error control). *For any predictor $\widehat{D}$ and deployment θ, the predicted cost satisfies*

$$\mathbb{E}_x \big| \widehat{C}_\theta(x) - \mathbb{E}[C_\theta \mid x] \big| \le \kappa_\theta \sqrt{N_{\max} B \left(\mathcal{R}(\widehat{D}) - \mathcal{R}(D) \right)}, \tag{9}$$

where $\mathcal{R} \triangleq \mathbb{E}[\mathcal{L}_B]$ is the Brier risk and κ_θ depends only on the profiled cost factors.

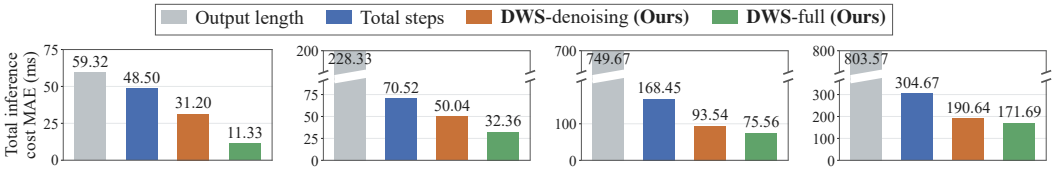
This bound depends on the predictor *only* through $\mathcal{R}(\widehat{D})$, which is why we minimize it in Stage B.

Second, beyond more accurate cost estimation, the *structured* representation of DWS also supports serving decisions at different *granularities*. (i) At the request level, it enables cost-aware scheduling such as SJF, which we evaluate in Section 6.4. (ii) At the block level, DWS can predict the denoising steps within a block, thus supporting grouping requests with similar predicted next-block workloads to mitigate intra-batch stragglers, or estimating remaining workload for preemptive scheduling. (iii) Moreover, the predicted output blocks can support KV-Cache estimation and memory allocation.

Third, the decoupled design allows DWS to *transfer* across deployments, which is crucial for serving, whereas a direct cost predictor must be retrained. The transferring experiments in Appendix K show that the unchanged DWS predictor can improve serving performance across deployments.

Effect of individual profiling. We measure requests *individually* to accurately isolate and attribute per-request costs. Batching and other runtime factors (such as KV-Cache reuse) may alter the cell costs, similarly to changes in hardware, but leave the DWS and its predictor *unchanged*. Moreover, using such request-level cost estimates to guide batch-level decisions is common practice in LLM serving (Zheng et al., 2026; Sheng et al., 2024; Fu et al., 2024). Empirically, with a batch size of 96 (the maximum before OOM), DWS-guided scheduling consistently performs best (shown in Section 6.4).

Limitations. (i) For parsimony, our cost model omits small and stable stages (Section 3.1) and secondary effects such as the quadratic attention computation term. We quantify the resulting residuals in Appendix D and find that they are well below the *inevitable* prediction error. (ii) Our decoupled design supports transfer across deployments, but assumes a *fixed* model family (e.g., LLaDA2.0), block size, and decoding threshold, which typically serve as *static* deployment configurations (Bie et al., 2025). Changing them requires retraining, which needs only activity matrices A logged from routine serving.



(a) G1 (22-299 ms) (b) G2 (300-1136 ms) (c) G3 (1137-3899 ms) (d) G4 (3900-8605 ms)

Figure 6: **Cost-model fidelity across request cost ranges.** DWS-denoising isolates the denoising component from our full model, enabling a direct comparison with scalar baselines. Lower is better.

6 Experimental Evaluation

In this section, we evaluate (i) the *fidelity* of DWS-based cost model, (ii) the *accuracy* and CPU *efficiency* of predictor, and (iii) the end-to-end *serving performance* of DWS-guided SJF scheduler.

6.1 Experimental Settings

Testbed and Models. Our experiments are conducted on a server with NVIDIA PRO 6000 GPUs (96 GB each), AMD EPYC 9534 CPUs (64 physical cores each), and 256 GB of host memory. We use popular SGLang-0.5.10 as the serving framework and evaluate two advanced dLLMs, LLaDA2.0-mini (TP=1) and LLaDA2.0-flash (Bie et al., 2025) (TP=4).

Serving configurations. Following prior work (Chen et al., 2026), we set the maximum output length to 1024 tokens. The batch size is 96 (the maximum before OOM). Unless otherwise specified, we use the standard decoding configuration of LLaDA2.0 (e.g., $B = 32$, and decoding threshold is 0.95).

Datasets. For serving experiments, we use two *real-world serving* datasets, LMSYS-Chat-1M (Zheng et al., 2024) and ShareGPT (RyokoAI, 2023), together with a synthetic dataset Alpaca (Taori et al., 2023). We *deliberately* adopt the **same datasets** from prior AR-LLM serving studies (Zheng et al., 2026; Fu et al., 2024) to evaluate DWS-based dLLM serving performance. Evaluations of the cost model and predictor use LMSYS-Chat-1M, with the training and test prompts kept disjoint.

6.2 Cost model evaluation

Settings. We evaluate the fidelity of different cost models here. To isolate modeling error from prediction error, all models take ground-truth statistics (e.g., output blocks). We compare against two scalar proxies: output length and total denoising steps. Since neither accounts for prefill or KV-refresh costs, we additionally evaluate the denoising component of DWS model alone for direct comparison. Scalar predictions are mapped to costs via a linear fit on the same 1K profiling traces as DWS. We partition the 20K traces described in Section 3.1 into four *equal-sized* bins by measured cost and conduct *5-fold cross-validation* within each bin, with results averaged across folds.

Results. As shown in Figure 6, Total Steps consistently outperforms Output Length, but still incurs clear error because it treats all denoising steps uniformly. By accounting for where these steps occur, DWS-denoising reduces MAE by up to $1.80\times$ over Total Steps across all four cost ranges. Further incorporating prefill and KV-refresh costs yields DWS-full, reducing the overall MAE by $2.04\times$ on average over Total Steps. These results suggest that preserving the structured denoising process is essential, while incorporating prefill and KV-refresh costs further improves fidelity of the cost model.

6.3 Predictor evaluation

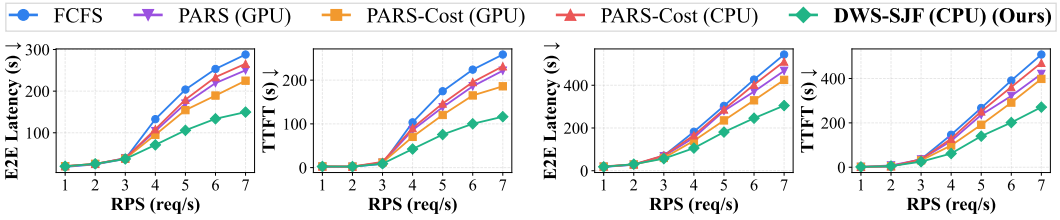
Settings. We evaluate the predictors in terms of cost-prediction accuracy, structure-prediction accuracy, and CPU efficiency, and further conduct ablation studies on the DWS predictor. We train all predictors on the same 100K traces (10K for validation) and evaluate them on 10K disjoint test traces.

Baselines. We compare DWS against three prompt-only predictors with different scalar prediction targets: *Output Length*, which predicts the generated output length; *Total Steps*, which predicts the total number of denoising steps; and *Direct Cost*, which directly regresses the measured per-request inference cost. All of them use the *same* MiniLM encoder. We further ablate Stage A and multi-horizon supervision (i.e., use only B-horizon), and evaluate an INT8-quantized DWS predictor.

Results. As shown in Table 1, DWS achieves the lowest cost MAE, 1.22–$2.50\times$ lower than the baselines. Despite a similar total-step MAE to the total-step predictor, DWS reduces cost MAE by $1.46\times$, confirming the benefit of preserving the denoising *structure* beyond merely counting

Table 1: **Prediction accuracy of prompt-only predictors and DWS *ablations*.** Cost MAE measures total-cost error; the others measure denoising-structure error. Throughput measurements use the same 256 prompts. "–" denotes a non-applicable metric. **Best** and second-best results are marked.

Predictor	Cost MAE (ms) ↓	Total-step MAE ↓	Per-block step MAE ↓	Block-count MAE ↓	Throughput on CPU (req/s) ↑
Output Length	887.97	–	–	**2.01**	37.67
Total Steps	517.04	62.50	–	–	37.54
Direct Cost (*untransferable*)	432.29	–	–	–	37.55
DWS	**355.24**	**56.06**	**4.41**	2.14	36.35
DWS (*w/o* Stage A)	604.43	87.91	6.02	3.68	36.34
DWS (*single*-horizon)	445.65	59.70	5.33	2.82	36.51
DWS (*INT8*)	357.17	56.42	4.62	2.19	**84.29**



(a) Performance under LMSYS-Chat-1M. (b) Performance under ShareGPT *(cross dataset)*.

Figure 7: **Online chatbot serving performance.** We report average E2E latency and TTFT (P90 in Appendix I). All predictors are trained on LMSYS-Chat-1M, so (b) tests cross-dataset *generalization*.

steps. Although Direct Cost performs better than other scalar baselines, it entangles prediction with deployment-specific costs and thus requires *retraining* across deployments.

For ablations, removing Stage A or multi-horizon supervision can increase the cost MAE by $1.70\times$ and $1.25\times$, respectively, showing that both designs help the DWS predictor learn the denoising structure better. INT8 quantization incurs only a *negligible* accuracy loss ($< 1\%$ in cost MAE), while improving throughput by $2.32\times$, enabling faster estimation of request inference cost to support downstream serving decisions and making the predictor more *practical* for deployment.

6.4 Real-world serving evaluation

Settings. We evaluate shortest-job-first (SJF) scheduling with different predictors, where requests with lower predicted costs are given higher priority. Following prior LLM serving systems (Zheng et al., 2026; Shahout et al., 2025), we incorporate a simple aging mechanism to *prevent starvation*: a waiting request's predicted cost is discounted by 10% every 30 s until it is selected.

Baselines. *(i) FCFS*, the *default* scheduling policy in serving frameworks such as SGLang and vLLM, which can suffer from head-of-line (HOL) blocking (Qiu et al., 2024). *(ii) PARS (GPU)* (Tao et al., 2026), a recent SJF scheduler for AR LLMs, which builds on learning-to-rank (Fu et al., 2024) and introduces noise filtering, with a BERT-base-uncased (Devlin et al., 2019) encoder. We adapt it to dLLM serving while preserving its *output-length*-based ranking and its original deployment, where the predictor shares the *GPU* with the serving model. *(iii) PARS-Cost (GPU)* strengthens PARS by replacing its prediction target and ranking objective with the direct *cost*. *(iv) PARS-Cost (CPU)*, whose predictor runs on a single CPU core, *matching* the resources of our predictor. *(v) DWS-SJF*, which schedules requests using the cost estimated by our DWS cost model and DWS (INT8) predictor.

Online Chatbot. Following prior studies (Zheng et al., 2026; Fu et al., 2024), we vary the request rate (RPS) to evaluate each scheduler under different system loads, and report the average end-to-end (E2E) latency and TTFT. Additional metrics (e.g., P90 latency) and results are provided in Appendix I.

Results. As shown in Figure 7, all schedulers perform similarly under light load, where requests rarely queue. As the load increases, DWS-SJF grows the slowest and consistently achieves the lowest E2E latency and TTFT on both datasets. On LMSYS-Chat-1M, it reduces E2E latency and TTFT by up to $1.92\times$ and $2.22\times$ over FCFS, and by up to $1.50\times$ and $1.65\times$ over PARS-Cost(GPU), the strongest baseline. These gains carry over to ShareGPT *without* retraining. The results further reveal the significance of efficient prediction: PARS-Cost (CPU) falls behind because slow prediction delays

Table 2: **Performance of schedulers on SDG tasks.** All predictors are trained on LMSYS-Chat-1M traces collected with LLaDA2.0-mini. We evaluate *(i)* in-domain, *(ii)* cross-dataset, *(iii)* cross-model, and *(iv)* cross-model-and-dataset settings to test the *generalization* of these schedulers.

Serving Model		Dataset	SDG Metrics	FCFS	PARS (GPU)	PARS-Cost (GPU)	PARS-Cost (CPU)	DWS-SJF (CPU) (Ours)
LLaDA 2.0-mini	**(i)**	LMSYS-Chat-1M	Time to 1K Req. (s) ↓	328	275	168	290	**120**
			Completed in 3 min ↑	517	661	1021	625	**1299**
	(ii)	Alpaca	Time to 1K Req. (s) ↓	197	159	108	165	**75**
			Completed in 3 min ↑	912	1127	1545	1069	**1836**
LLaDA 2.0-flash	**(iii)**	LMSYS-Chat-1M	Time to 1K Req. (s) ↓	1598	1194	1085	1382	**592**
			Completed in 3 min ↑	103	155	181	138	**310**
	(iv)	Alpaca	Time to 1K Req. (s) ↓	909	752	681	825	**407**
			Completed in 3 min ↑	198	263	292	245	**468**

cost estimates, thus the scheduler may blindly choose expensive requests, reintroducing the HOL blocking that SJF is meant to avoid. This is precisely why we keep the DWS predictor efficient.

Offline Synthetic Data Generation (SDG). SDG often aims to obtain a target number of samples from a large prompt pool as quickly as possible, making it a natural testbed for SJF scheduling. Following prior work (Zheng et al., 2026; Fu et al., 2024), we submit 3K requests at once and evaluate (i) how long to complete 1K requests and (ii) how many requests are completed within 3 minutes. For cross-model evaluation, all cost-based schedulers are re-profiled on the *same* 1K LLaDA2.0-flash traces.

Results. Table 2 shows that DWS-SJF performs best on both metrics across all serving models and datasets. On LLaDA2.0-mini with LMSYS-Chat-1M, it completes the first 1K requests $2.73\times$ faster than FCFS and finishes $2.51\times$ more requests within 3 minutes, outperforming PARS-Cost (GPU) (the strongest baseline) by $1.40\times$ and $1.27\times$, respectively. These gains also hold on Alpaca and LLaDA2.0-flash *without retraining*, suggesting its *generalization* beyond the training workload and serving model. Notably, the gains of PARS-Cost (GPU) shrink substantially when transferred to LLaDA2.0-flash, as its predictor is *coupled* to deployment-specific costs. In contrast, DWS-SJF retains most of its gains by *decoupling* its predictor from deployment-specific cost factors.

Additional studies. Due to space constraints, we defer the following studies to the appendix: (i) additional results, including a MiniLM-based PARS-Cost baseline, a comparison with Oracle-SJF, and tail latency (Appendix I), (ii) cross-deployment experiments (PRO 6000 96G TP=1 $\rightarrow$ A6000 48G TP=2) (Appendix K), (iii) effect of the encoder (Appendix J), and (iv) long-context robustness (Appendix D).

7 Related Work

Due to space constraints, we focus here on the most closely related dLLM serving studies and defer further discussion to Appendix M, including dLLM inference acceleration and AR LLM serving.

For dLLMs, a growing body of recent work focuses on improving the efficiency of dLLM serving. dLLM-Serve (Fan et al., 2026) improves memory and resource efficiency through phase-aware execution, while Optimus (Wei et al., 2026) and BlockServe (Zhu et al., 2026) improve serving efficiency under dynamic loads and heterogeneous convergence through adaptive decoding and batching. DiLaServe (Chang et al., 2026) further targets SLO-aware serving with deadline-aware scheduling and adaptive resource management. These systems primarily optimize dLLM execution at runtime, so they are *complementary* to DWS: DWS provides a fine-grained and deployment-adaptable cost signal *before* inference, which can further support scheduling and resource management.

8 Conclusion

We present DWS, a structured workload abstraction for dLLM serving. Unlike scalar proxies such as output length or total denoising steps, DWS preserves the heterogeneous computation structure across output blocks and denoising steps, enabling a more *faithful* estimation of request inference cost. We further design an accurate prompt-only DWS predictor for each request's *fine-grained* denoising structure, whose lightweight design enables *efficient* prediction even on a single CPU core. By *decoupling* request-dependent execution behavior from deployment-specific cost factors, DWS can adapt across deployments through lightweight profiling without retraining its prompt-only predictor. Experiments show that DWS consistently improves both cost-modeling fidelity and cost-prediction accuracy, yielding better serving performance with strong generalization.

AI USE STATEMENT

In this work, we used generative AI tools to improve the clarity and readability of the manuscript.

ETHICS STATEMENT

We are not aware of any specific ethical concerns associated with this work. Our experiments use publicly available datasets (LMSYS-Chat-1M, ShareGPT, Alpaca) purely as serving workloads.

REPRODUCIBILITY STATEMENT

Our code of model training and the DWS system is available at `https://anonymous.4open.science/r/DS1`.

Implementation details and hyperparameters are provided in Section 6.1 and Appendix H.

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. *Advances in neural information processing systems*, 34:17981–17993, 2021.

Tiwei Bie, Maosong Cao, Kun Chen, Lun Du, Mingliang Gong, Zhuochen Gong, Yanmei Gu, Jiaqi Hu, Zenan Huang, Zhenzhong Lan, et al. LLaDA2.0: Scaling up diffusion language models to 100B. *arXiv preprint arXiv:2512.15745*, 2025.

Tzu-Tao Chang, Benjamin Yuanyang Hong, Kiet Pham, and Shivaram Venkataraman. DiLaServe: High SLO attainment serving for diffusion language models. *arXiv preprint arXiv:2606.29094*, 2026.

Zhiben Chen, Youpeng Zhao, Yang Sui, Jun Wang, and Yuzhang Shang. TIDE: Efficient and lossless MoE diffusion LLM inference with I/O-aware expert offload. *arXiv preprint arXiv:2605.20179*, 2026.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pp. 4171–4186, 2019.

DiffusionGemma Team, Adrien Ali Taïga, James Assiene, Daniele Calandriello, Rahma Chaabouni, João Gante, Tamara von Glehn, Nate Keating, Chris Knutsen, Martin Kukla, Tianlin Liu, Ivan Lobov, Ofir Nabati, João Gabriel Oliveira, Nicolas Perez-Nieves, Nastasia Prutianova, Bobak Shahriari, Jean Tarbouriech, Pavel Tyletski, Çağlar Ünlü, Cindy Wu, Glenn Cameron, Jerome Connor, Sertan Girgin, Maarten Grootendorst, Alon Levkovitch, Eliya Nachmani, Omar Sanseviero, Piotr Stanczyk, Quentin Berthet, Andrew Campbell, Clément Crepy, Valentin De Bortoli, Arnaud Doucet, Romuald Elie, Alexandre Galashov, Klaus Greff, Alexis Jacq, David Ruhe, Yu-Han Wu, Sebastian Flennerhag, Brendan O'Donoghue, George Scrivener, and Shantanu Thakoor. DiffusionGemma technical report, 2026. URL `https://arxiv.org/abs/2608.00146`.

Jiakun Fan, Yanglin Zhang, Xiangchen Li, and Dimitrios S Nikolopoulos. dLLM-Serve: Bridging the memory gap in diffusion language model serving. In *Proceedings of the 40th ACM International Conference on Supercomputing*, pp. 515–526, 2026.

Ronald Aylmer Fisher. *Statistical methods for research workers*. Oliver and Boyd, 1930.

Yichao Fu, Siqi Zhu, Runlong Su, Aurick Qiao, Ion Stoica, and Hao Zhang. Efficient LLM scheduling by learning to rank. *Advances in Neural Information Processing Systems*, 37:59006–59029, 2024.

Andrew Gelman. Analysis of variance–why it is more important than ever. *The Annals of Statistics*, 33(1):1–31, 2005. ISSN 00905364.

Shansan Gong, Shivam Agarwal, Yizhe Zhang, Jiacheng Ye, Lin Zheng, Mukai Li, Chenxin An, Peilin Zhao, Wei Bi, Jiawei Han, et al. Scaling diffusion language models via adaptation from autoregressive models. In *International Conference on Learning Representations*, volume 2025, pp. 5046–5073, 2025.

Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. s^3: Increasing gpu utilization during generative inference for higher throughput. *Advances in Neural Information Processing Systems*, 36:18015–18027, 2023.

Jia-Nan Li, Jian Guan, Wei Wu, and Chongxuan Li. ReFusion: A diffusion large language model with parallel autoregressive decoding. In *International Conference on Learning Representations*, volume 2026, pp. 53846–53869, 2026.

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, 2025a.

Yuxin Ma, Lun Du, Lanning Wei, Kun Chen, Qian Xu, Kangyu Wang, Guofeng Feng, Guoshan Lu, Lin Liu, Xiaojing Qi, et al. dInfer: An efficient inference framework for diffusion language models. *arXiv preprint arXiv:2510.08666*, 2025b.

Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Large language diffusion models. *Advances in Neural Information Processing Systems*, 38:50608–50646, 2025.

Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew T. Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. Efficient interactive LLM serving with proxy model-based sequence length prediction. In *The 5th International Workshop on Cloud Intelligence / AIOps at ASPLOS 2024*, volume 5, pp. 1–7, San Diego, CA, USA, 2024. Association for Computing Machinery.

RyokoAI. ShareGPT52K. `https://huggingface.co/datasets/RyokoAI/ShareGPT52K`, 2023.

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.

SGLang. Power up diffusion LLMs: Day-0 support for LLaDA 2.0. `https://www.lmsys.org/blog/2025-12-19-diffusion-llm/`, December 2025. LMSYS Blog.

Rana Shahout, Eran Malach, Chunwei Liu, Weifan Jiang, Minlan Yu, and Michael Mitzenmacher. Don't stop me now: Embedding based scheduling for LLMs. In *The Thirteenth International Conference on Learning Representations*, 2025.

Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E Gonzalez, and Ion Stoica. Fairness in serving large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 965–988, 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.

Yiheng Tao, Yihe Zhang, Matthew Dearing, Xin Wang, Yuping Fan, Michael Papka, and Zhiling Lan. Ranking before serving: Low-latency LLM serving via pairwise learning-to-rank. In *ISC High Performance 2026 Research Paper Proceedings (41st International Conference)*, pp. 1–13. Prometeus GmbH, 2026.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford Alpaca: An instruction-following LLaMA model. `https://github.com/tatsu-lab/stanford_alpaca`, 2023.

vLLM. DiffusionGemma: The first diffusion LLM (dLLM) natively supported in vLLM. https://vllm-project.github.io/2026/06/10/diffusion-gemma.html, June 2026. vLLM Blog.

Jing Wang, Yu-Yang Qian, Ke Xue, Chao Qian, Peng Zhao, and Zhi-Hua Zhou. Robust length prediction: A perspective from heavy-tailed prompt-conditioned distributions. *arXiv preprint arXiv:2604.07931*, 2026a.

Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. *Advances in neural information processing systems*, 33:5776–5788, 2020.

Xu Wang, Chenkai Xu, Yijie Jin, Jiachun Jin, Hao Zhang, Kai Yu, and Zhijie Deng. Diffusion LLMs can do faster-than-ar inference via discrete diffusion forcing. In *International Conference on Learning Representations*, volume 2026, pp. 2282–2299, 2026b.

Chiyue Wei, Cong Guo, Bowen Duan, Junyao Zhang, Haoxuan Shan, Yifei Wang, Yangjie Zhou, Hai Li, Danyang Zhuo, Yiran Chen, et al. Optimus: Elastic decoding for efficient diffusion LLM serving. *arXiv preprint arXiv:2605.24832*, 2026.

Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. FastServe:iteration-level preemptive scheduling for large language model inference. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pp. 57–74, 2026a.

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, 2026b.

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

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

Haoyu Zheng, Yongqiang Zhang, Fangcheng Fu, Xiaokai Zhou, Hao Luo, Hongchao Zhu, Yuanyuan Zhu, Hao Wang, Xiao Yan, and Jiawei Jiang. Scheduling LLM inference with uncertainty-aware output length predictions. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=I5IMkvVKd7.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. LMSYS-Chat-1M: A large-scale real-world LLM conversation dataset. In *The Twelfth International Conference on Learning Representations*, 2024.

Zangwei Zheng, Xiaozhe Ren, Fuzhao Xue, Yang Luo, Xin Jiang, and Yang You. Response length perception and sequence scheduling: An LLM-empowered LLM inference pipeline. *Advances in Neural Information Processing Systems*, 36:65517–65530, 2023.

Yuanjie Zhu, Liangwei Yang, Ke Xu, Weizhi Zhang, Shanghao Li, Zihe Song, and Philip S Yu. BlockServe: Block-grained continuous batching for high-throughput diffusion LLM serving. *arXiv preprint arXiv:2607.08930*, 2026.

A Notation Conventions

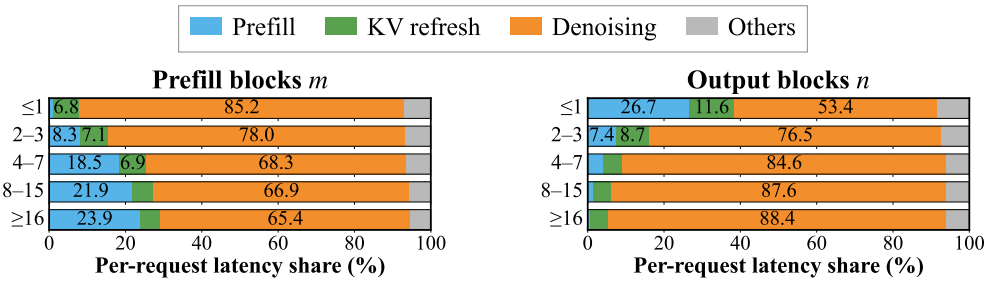
Table 3 summarizes the main notation used throughout the paper. We use (b, s) to index a denoising cell, where b denotes the output-block index and s the within-block denoising-step index. Quantities marked with * denote measured execution costs, whereas their unmarked counterparts denote the corresponding quantities in our cost model. For prediction, $D(x)$ denotes the prompt-conditioned DWS target and $\hat{D}(x)$ its predicted value.

Table 3: Summary of notation.

Symbol	Meaning
Request and denoising execution	
x	Input prompt of a request.
B	Configured block size, i.e., the maximum number of tokens and denoising steps within one output block.
$N_{\max}$	Maximum number of output blocks allowed by the serving configuration.
m	Number of prompt blocks.
n	Number of generated output blocks.
b	Output-block index, $b \in \{1, \dots, N_{\max}\}$.
s	Within-block denoising-step index, $s \in \{1, \dots, B\}$.
s_b	Number of denoising steps executed for output block b.
$A \in \{0,1\}^{N_{\max} \times B}$	Denoising activity matrix of a realized request execution.
$A_{b,s}$	Activity indicator of cell (b, s); $A_{b,s} = 1$ iff block b is generated and executes at least s denoising steps.
Cost modeling	
C	Modeled total inference cost of a request.
C_{prefill}, $C_{\text{KVrefresh}}$, $C_{\text{denoising}}$	Modeled costs of prefill, KV refresh, and denoising, respectively.
$c^{\star}_{b,s}$	Measured cost of denoising cell (b, s) in a realized execution.
$K(m) \in \mathbb{R}^{N_{\max} \times B}$	Profiled cell-cost matrix for a request with m prompt blocks.
$K_{b,s}(m)$	Modeled cost of denoising cell (b, s) for prompt length m.
$W \in \mathbb{R}^{N_{\max} \times B}$	Request-shared baseline cell-cost matrix.
$\bar{m}$	Reference prompt-block count used when calibrating the cell-cost model.
δ	Prompt-length correction coefficient in $K_{b,s}(m) = W_{b,s} + \delta(m - \bar{m})$.
α	Profiled per-block coefficient for the prefill cost, $C_{\text{prefill}}(m) = \alpha m$.
β	Profiled per-block coefficient for the KV-refresh cost, $C_{\text{KVrefresh}}(n) = \beta n$.
DWS and workload prediction	
$D(x) \in [0,1]^{N_{\max} \times B}$	Denoising Workload Surface (DWS) conditioned on prompt x.
$D_{b,s}(x)$	Activity probability of cell (b, s): $P(A_{b,s} = 1 \mid x)$.
$\hat{D}(x)$	DWS predicted from prompt x.
$p^{\text{block}}_k(x)$	Predicted probability $P(n = k \mid x)$ of generating exactly k output blocks.
$p^{\text{step}}_{b,t}(x)$	Predicted conditional probability $P(s_b = t \mid n \geq b, x)$.
$z(x)$	Pooled prompt representation produced by the predictor encoder.
h	Step horizon used in multi-horizon workload supervision.
$T^{(h)}$	Workload truncated at horizon h: $\sum_{b=1}^{n} \min(s_b, h)$.

B Modeling Prefill and KV-Refresh Costs

This section details the prefill and KV-refresh cost models summarized in Section 3.1. We first identify their cost drivers (Appendix B.1) and validate the linear models over the full supported context range (Appendix B.2), and then examine whether higher-order terms improve fidelity in long-context regimes (Appendix D).



(a) Varying prompt blocks.

(b) Varying output blocks.

Figure 8: Latency breakdown under varying block counts.

Table 4: Two-way ANOVA decomposition of component latency.

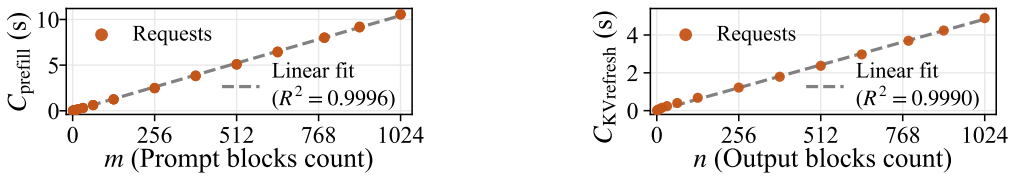
Component	Prompt blocks m	Output blocks n	Interaction
Prefill	**99.87%**	0.01%	0.12%
KV refresh	0.15%	**99.26%**	0.59%

B.1 Cost Drivers

Mechanically, prefill processes each prompt block once, whereas KV refresh is performed once after each output block is fully denoised. This suggests that the number of prompt blocks m and the number of output blocks n are their respective primary cost drivers. To verify this, we apply a two-way ANOVA decomposition (Fisher, 1930; Gelman, 2005) to the 20K traces described in Section 3.1, quantifying the relative contributions of m, n, and their interaction. As shown in Table 4, m accounts for 99.87% of the decomposed variation in prefill latency, while n accounts for 99.26% of that in KV-refresh latency, with interaction contributions below 1% in both cases. These clear separations justify modeling prefill and KV-refresh costs as functions of m and n, respectively.

B.2 Linear Cost Models

The ANOVA decomposition identifies the cost drivers but does not establish their functional relationship with cost. We therefore profile each component while sweeping its driver over the full context range supported by LLaDA2.0.



(a) C_{prefill} scales linearly with m.

(b) $C_{\text{KVrefresh}}$ scales linearly with n.

Figure 9: Linear fits for prefill and KV-refresh costs.

Prefill cost. We sweep the prompt length from 32 tokens ($m = 1$) to 32K tokens ($m = 1{,}024$, the context limit of LLaDA2.0). As shown in Figure 9a, the linear model $C_{\text{prefill}}(m) \triangleq \alpha m$ fits the measurements closely ($R^2 = 0.9996$). Although attention introduces an $O(m^2)$ term in principle, this term remains small relative to the linear cost of the QKV projections and feed-forward layers within this range, as further analyzed in Appendix D.

KV-refresh cost. Since each completed output block triggers exactly one KV refresh, the number of refreshes equals n. We sweep the output length from 32 tokens ($n = 1$) to 32K tokens ($n = 1{,}024$) by disabling early termination (`ignore_eos=True`). As shown in Figure 9b, the linear model

Table 5: Paired comparison between the first and last denoising steps within the same block on LLaDA2.0-mini.

Metric	First step	Last step	Δ
Forward latency (ms)	6.043	8.622	+2.579
Routed-expert layers, total (ms)	3.038	5.670	+2.632
Attention layers, total (ms)	1.114	1.116	+0.002
Activated experts per layer	32.7	66.3	+33.6
Padded rows per layer	611	1,102	+491

$C_{\text{KVrefresh}}(n) \triangleq \beta n$ also fits the measurements closely ($R^2 = 0.9990$), with higher-order effects analyzed in Appendix D.

Figure 8 provides a complementary view by breaking down per-request latency by block counts. Along the prompt dimension, the prefill share increases steadily with m, from 1.1% for requests with at most one prompt block to 23.9% for those with at least 16 blocks. Along the output dimension, prefill and KV refresh together account for 38.3% of latency for single-block outputs, while denoising increasingly dominates as n grows, rising from 53.4% to 88.4%. These trends are consistent with the ANOVA results and further show that prefill and KV refresh weigh most on short requests, which are exactly the requests that SJF scheduling must identify correctly.

C Analysis of the Per-Step Denoising Cost

In the main text, we observe that denoising steps at different execution positions can incur different costs. In this section, we further analyze the sources of such cost heterogeneity.

In short, we identify two major sources of per-step denoising-cost heterogeneity: (i) dynamic expert routing in mixture-of-experts (MoE) architectures and (ii) the growing context length. *Notably*, the former is particularly relevant to practical dLLM serving: among all dLLM variants natively supported by the latest available releases of SGLang and vLLM[3], 9/10 use MoE backbones.

C.1 Model Architecture

We first analyze the pronounced within-block cost variation in LLaDA2.0-mini through a module-level latency breakdown. Table 5 compares the first and last denoising steps of 64 blocks. The average forward latency increases from 6.043 ms to 8.622 ms. This increase is concentrated in routed-expert computation: the total latency of the routed-expert layers increases from 3.038 ms to 5.670 ms, while the attention latency remains nearly unchanged. Meanwhile, the average number of activated experts per layer increases from 32.7 to 66.3, and the average number of padded rows increases from 611 to 1,102.

We next examine the routing behavior underlying this change. Although each token always selects a fixed number of experts, its hidden state evolves during denoising, which changes the token-to-expert assignments. As these assignments become more dispersed, a block covers a larger set of experts. In the current MoE kernel, token rows are padded separately for each expert; therefore, changes in the routing distribution can increase padded computation and alter expert-weight access patterns.

To further isolate the effect of routing itself, we fix the hidden states and assign identical weights to all experts, while changing only the expert-ID assignments. Even with bitwise-identical model outputs, different routing patterns still incur different execution costs. This result shows that the routing distribution itself can directly affect the cost of a denoising step.

[3]SGLang-0.5.20 (2026-09-18) and vllm-0.29.0 (2026-09-09)

C.2 Context Length

Besides model architecture, the cost of a denoising step also depends on the preceding context length. Within the same output block, the context length remains unchanged across denoising step s; instead, it grows with the input prompt length and as generation proceeds to later output blocks.

To quantify this effect, we vary the input context length while controlling the denoising state and block position. As the input length increases from 256 tokens to approximately 32K tokens, the attention-kernel latency increases from 0.6091 ms to 1.4104 ms, while the overall forward latency increases from 6.5730 ms to 7.3447 ms.

Overall, the two factors affect complementary dimensions of denoising execution: dynamic routing induced by the model architecture mainly drives cost variation across s within the same block, while context growth further changes the execution cost across requests and output-block positions.

C.3 Implications for Serving Cost Modeling

These analyses have different implications across model architectures. (i) For dense dLLMs, within-block denoising costs can be relatively homogeneous, while context growth still introduces systematic cost variation across requests and output positions. (ii) For the *increasingly prevalent* MoE dLLMs, dynamic expert routing further introduces pronounced variation across within-block denoising steps, making the homogeneous per-step assumption used by total-step-based models increasingly inaccurate.

Importantly, DWS does not rely on MoE-specific mechanisms. Its two-dimensional execution representation remains applicable to dense dLLMs, while its fine-grained modeling becomes particularly valuable for MoE models where per-step cost heterogeneity is stronger. Specifically, the DWS-based cost model preserves the execution structure indexed by output block b and within-block denoising step s, while separating request-dependent execution activity from deployment-specific cell costs. The profiled cell-cost matrix captures position-dependent cost differences, including those arising from dynamic MoE execution, while prompt-side context dependence is modeled through the prompt-length correction and output-side context growth through the block-indexed cost profile.

Therefore, DWS provides a ***unified*** structured formulation across dense and MoE dLLMs, while offering greater modeling benefits when the underlying per-step costs are more heterogeneous, as in prevalent MoE-based serving models.

D Residual Analysis of the Cost Model

In this section, we analyze the residuals induced by the simplifications of our cost model. We first quantify the effect of omitting the small and stable stages under real serving workloads (Appendix D.1). We then study how long prompts and long generated contexts affect the modeling fidelity of prefill, KV refresh, and denoising (Appendices D.2–D.4).

D.1 Effect of Omitted Stages

Settings. Section 6.2 has shown the overall fidelity of our cost model in terms of MAE. Here, we further examine whether the remaining modeling residuals affect the request ordering used by SJF. We feed the ground-truth m and A from 3K LMSYS-Chat-1M traces into the cost model, thereby excluding prediction error. The measured end-to-end cost includes the *Others* stages omitted from our model, which account for only $6.7 \pm 1.4\%$ of latency (Figure 3). Therefore, the resulting residual reflects both these omitted stages and other modeling simplifications, such as the quadratic attention term omitted from our linear cost models. We report both fitting accuracy and ranking consistency.

Results. The modeled cost achieves a MAPE of 4.30% and $R^2 = 0.9967$. Importantly, it largely preserves request ordering, with a Spearman's ρ of 0.9991, and 99.21% pairwise ordering accuracy. Thus, the remaining modeling residuals have little impact on cost-aware scheduling.

Table 6: **Prefill-cost modeling under long prompts.** MAPE of the linear model used in the main paper and a quadratic alternative across different prompt-length ranges.

Model	≤1K	1–2K	2–4K	4–8K	8–16K	16–32K
αm	8.99%	5.01%	4.18%	3.03%	2.00%	1.61%
$\alpha_1 m + \alpha_2 m^2$	7.42%	3.73%	2.50%	2.34%	1.47%	1.52%

Table 7: **KV-refresh-cost modeling under long outputs.** MAPE of the linear model used in the main paper and a quadratic alternative across different output-length ranges.

Model	≤1K	1–2K	2–4K	4–8K	8–16K	16–32K
βn	5.71%	6.05%	5.53%	6.60%	8.08%	10.17%
$\beta_1 n + \beta_2 n^2$	4.15%	4.55%	4.82%	5.70%	7.68%	9.40%

D.2 Prefill Cost under Long Prompts

Settings. As the prompt context grows, higher-order cost effects may become non-negligible; for example, attention computation can introduce a quadratic term in the prompt length. Meanwhile, other major operations in prefill, such as projections and feed-forward layers, scale approximately linearly with sequence length. Therefore, although the main paper adopts the parsimonious linear model $C_{\text{prefill}}(m) = \alpha m$, we further examine whether incorporating such higher-order effects improves modeling fidelity under long prompts. We compare it against a quadratic alternative, $C_{\text{prefill}}(m) = \alpha_1 m + \alpha_2 m^2$, over prompt lengths up to 32K tokens, the maximum context length supported by LLaDA2.0. Since real serving datasets contain relatively few prompts in these extreme length ranges, we construct prompts using randomly sampled tokens for this controlled experiment. We report MAPE separately over six prompt-length ranges.

Results. As shown in Table 6, the prefill cost remains well approximated by the linear model throughout the supported context range. Although the quadratic model is more flexible and consistently reduces fitting error, its additional gain becomes small in the long-context regime. For 8–16K prompts, it reduces MAPE from 2.00% to 1.47%, while for 16–32K prompts the improvement further shrinks to $1.61\% \rightarrow 1.52\%$. Thus, although self-attention theoretically introduces quadratic computation with sequence length, this higher-order effect provides little additional modeling fidelity within the context range supported by LLaDA2.0. Moreover, under the real-world workloads characterized in Figure 3, prefill accounts for only 5.8% of the total inference latency on average, so such component-level improvements have an even smaller effect on the overall cost. We therefore retain the linear form $C_{\text{prefill}}(m) = \alpha m$ for parsimony.

D.3 KV-Refresh Cost under Long Outputs

Settings. Each completed output block triggers one KV-refresh operation. While the refreshed block has a fixed size, later refreshes operate with an increasingly long generated context, so the cost of an individual refresh may grow with the output-block index. For example, if the refresh cost at block b grows approximately as $\gamma_0 + \gamma_1 b$, accumulating it over n output blocks gives

$$\sum_{b=1}^{n} (\gamma_0 + \gamma_1 b) = \gamma_0 n + \frac{\gamma_1}{2} n(n+1), \tag{10}$$

which motivates examining a quadratic component in the total KV-refresh cost. We therefore compare the linear model $C_{\text{KVrefresh}}(n) = \beta n$ used in the main paper against a quadratic alternative, $C_{\text{KVrefresh}}(n) = \beta_1 n + \beta_2 n^2$. For this experiment, we use real request traces to preserve the actual generation behavior of the dLLM, and raise the maximum generation length to 32K tokens, corresponding to the maximum context length supported by LLaDA2.0. We report MAPE across six output-length ranges.

Results. As shown in Table 7, the residual error of the linear KV-refresh model gradually increases for very long outputs, reaching 8.08% for 8–16K tokens and 10.17% for 16–32K tokens. However,

Table 8: Denoising-cost modeling under long prompts. MAPE of the linear prompt-length correction and a quadratic alternative across different prompt-length ranges.

Model	≤1K	1–2K	2–4K	4–8K	8–16K	16–32K
$W_{b,s} + \delta(m - \bar{m})$	**3.29%**	**2.32%**	1.83%	2.03%	2.51%	3.57%
$W_{b,s} + \delta_1(m - \bar{m}) + \delta_2(m - \bar{m})^2$	3.57%	2.95%	**1.53%**	**1.49%**	**1.87%**	**2.08%**

introducing the mechanism-motivated quadratic term provides only limited additional fidelity, reducing the corresponding errors to 7.68% and 9.40%. In particular, even in the longest 16–32K range, the quadratic model improves MAPE by only 0.77 percentage points. This indicates that the residual growth under very long outputs cannot be substantially explained by simply increasing the polynomial order of the model. Moreover, under the real-world workloads characterized in Figure 3, KV refresh accounts for only 6.7% of the total inference latency on average, further limiting the impact of this modest component-level improvement on the overall cost. Given the limited gain from the additional parameterization, we retain the linear form $C_{\mathrm{KVrefresh}}(n) = \beta n$ in the main cost model.

D.4 Denoising Cost under Long Context

We separately study how long prompts and long generated outputs affect the modeling fidelity of our denoising-cost model. **Long prompts.** As discussed in Section 3.2, the cost of a denoising step depends on the preceding context. Our cost model captures the output-side context growth through the output-block index b in the baseline cell-cost matrix W, while modeling the request-dependent prompt-length effect using a shared linear correction:

$$K_{b,s}(m) = W_{b,s} + \delta(m - \bar{m}). \tag{11}$$

As the prompt becomes longer, however, higher-order effects from attention may become increasingly visible. We therefore examine whether introducing an additional quadratic prompt-length term improves modeling fidelity:

$$K^{\mathrm{quad}}_{b,s}(m) = W_{b,s} + \delta_1(m - \bar{m}) + \delta_2(m - \bar{m})^2. \tag{12}$$

The corresponding request-level denoising cost is

$$C^{\mathrm{quad}}_{\mathrm{denoising}} = \langle W, A \rangle + \delta_1(m - \bar{m})T + \delta_2(m - \bar{m})^2 T, \tag{13}$$

where $T = \sum_{b,s} A_{b,s}$ denotes the total number of executed denoising cells. Compared with the linear formulation, this alternative introduces only one additional scalar coefficient while retaining the same structured baseline matrix W. Following the controlled setup in Appendix D.2, we construct prompts of different lengths using randomly sampled tokens, since real serving datasets contain relatively few prompts in the extreme length ranges. The random-token construction is used to control the prompt length, while the denoising activity matrix A and denoising latency are obtained from the actual generation process. We sweep the prompt length up to 32K tokens, the maximum context length supported by LLaDA2.0, and report MAPE separately over six prompt-length ranges.

As shown in Table 8, the benefit of the quadratic correction becomes increasingly apparent toward the long-context regime. While the quadratic model slightly increases MAPE for prompts shorter than 2K tokens, it consistently improves over the linear model beyond 2K. The reduction grows from 0.30 percentage points for 2–4K prompts to 0.54 and 0.64 points for 4–8K and 8–16K prompts, respectively, and reaches 1.49 points in the longest 16–32K range, reducing MAPE from 3.57% to 2.08%. These results indicate that higher-order prompt-length effects become more pronounced under very long contexts. At the same time, the linear correction remains accurate throughout the supported context range, with MAPE below 3.6% in every range. Given its low residual error and simpler parameterization, we retain the linear form $K_{b,s}(m) = W_{b,s} + \delta(m - \bar{m})$ for parsimony.

Long outputs. We next examine the effect of long generated contexts on denoising cost. Unlike the prompt-side dependence, output-side context growth is explicitly indexed by the output-block position b and is thus largely captured by the corresponding entries of the baseline matrix W. We therefore directly evaluate whether the existing cell-cost model remains accurate as generation proceeds to

Table 9: Denoising-cost modeling under long outputs. MAPE of the profiled cell-cost model across different output-length ranges.

Output length	$\leq$1K	1–2K	2–4K	4–8K	8–16K	16–32K
MAPE	3.56%	3.14%	3.40%	4.86%	5.29%	5.73%

increasingly late output blocks, without introducing an additional output-length correction. For this experiment, we use real request traces to preserve the actual denoising behavior of the dLLM and raise the maximum generation length to 32K tokens, corresponding to the maximum context length supported by LLaDA2.0. We report MAPE separately over six output-length ranges.

As shown in Table 9, the modeling error remains relatively stable for outputs up to 4K tokens, with MAPE between 3.14% and 3.56%. As the generated context grows further, the residual error becomes increasingly visible: MAPE rises to 4.86% for 4–8K outputs, 5.29% for 8–16K, and 5.73% in the longest 16–32K range. This trend indicates that secondary context-dependent effects beyond the block-indexed baseline become more pronounced under very long generated contexts. At the same time, the b-indexed baseline matrix W remains accurate throughout the supported output range, with MAPE below 5.8% in every range. These results suggest that explicitly indexing the output-side context growth by b captures its dominant effect without requiring an additional output-length correction. Given the low residual error and the absence of an additional parameterization, we retain the original cell-cost formulation $K_{b,s}(m) = W_{b,s} + \delta(m - \bar{m})$.

E Automatic Profiling and Refresh of cost factors Online

As discussed in Section 4, DWS decouples the request-dependent execution structure from the deployment-specific cost factors α, β, and $K(\cdot)$. This separation allows the cost factors to be automatically calibrated from execution traces and periodically refreshed without retraining the DWS predictor. We describe the profiling and fitting procedure below.

Trace collection. For each profiled request i, the serving engine records its prompt-block count m_i, output-block count n_i, and the denoising depth $s_{i,b}$ of each generated block. These observations directly determine the activity matrix A_i, where $A_{i,b,s} = \mathbb{1}\{b \leq n_i \wedge s \leq s_{i,b}\}$, and the total number of denoising steps $T_i = \sum_{b,s} A_{i,b,s}$. Meanwhile, CUDA-event instrumentation measures the corresponding prefill, KV-refresh, and denoising costs, denoted by y_i^{pre}, y_i^{kv}, and y_i^{den}, respectively. To isolate per-request cost from batching effects, profiling requests are measured individually under the target deployment configuration.

Profiling prefill and KV-refresh costs. Recall that we model the prefill and KV-refresh costs as αm and βn, respectively. Their coefficients can therefore be estimated directly by aggregating the corresponding measured costs:

$$\alpha = \frac{\sum_i y_i^{\mathrm{pre}}}{\sum_i m_i}, \qquad \beta = \frac{\sum_i y_i^{\mathrm{kv}}}{\sum_i n_i}. \tag{14}$$

Both parameters can thus be refreshed using only simple running statistics.

Profiling the cell-cost function. The denoising component is more challenging because it contains the fine-grained cell-cost function $K_{b,s}(m) = W_{b,s} + \delta(m - \bar{m})$. For a profiling set of N requests, we use the step-weighted prompt center $\bar{m} = \frac{\sum_i m_i T_i}{\sum_i T_i}$, such that the modeled denoising cost of request i is $\hat{y}_i^{\mathrm{den}} = \langle W, A_i \rangle + \delta(m_i - \bar{m})T_i$.

We jointly fit the baseline cell-cost matrix W and the prompt-length correction δ using weighted least squares:

$$\begin{aligned} \min_{W \geq 0, \delta} \quad & \frac{1}{N} \sum_i \omega_i \left(\langle W, A_i \rangle + \delta(m_i - \bar{m})T_i - y_i^{\mathrm{den}} \right)^2 \\ & + \lambda_s \|\nabla_s W\|_F^2 + \lambda_b \|\nabla_b W\|_F^2, \end{aligned} \tag{15}$$

where $\omega_i = (T_i + 10)^{-2}$ reduces the dominance of long-running requests in the fitting objective. The operators ∇_s and ∇_b denote first-order differences between neighboring cells along the denoising-

Table 10: **Effect of predictor precision.** All variants use the same trained DWS predictor and are evaluated on a single CPU core. Lower is better for MAE, while higher is better for throughput.

Precision	Total-step MAE ↓	Per-block step MAE ↓	Block-count MAE ↓	Throughput (req/s) ↑
FP32 (*Original*)	56.06	4.41	2.14	36.35
FP16	56.19	4.47	2.16	17.08
INT8	56.42	4.62	2.19	**84.29**
INT4	56.92	4.82	2.26	58.71

step and output-block dimensions, respectively. The corresponding smoothness terms stabilize the estimation under uneven cell coverage, while the non-negativity constraint on W prevents invalid negative baseline cell costs. We select λ_s and λ_b on held-out profiling traces and refit the parameters using the complete profiling set.

Incremental statistics and refresh. Importantly, refreshing the profile does not require retaining and repeatedly processing all historical traces. Let $a_i = \mathrm{vec}(A_i)$ and $z_i^0 = m_i T_i$. The data term in Equation 15 can be represented by additive sufficient statistics, including

$$G = \sum_i \omega_i a_i a_i^\top, \qquad r = \sum_i \omega_i a_i y_i^{\mathrm{den}}, \qquad q = \sum_i \omega_i a_i z_i^0, \tag{16}$$

together with $\sum_i \omega_i (z_i^0)^2$, $\sum_i \omega_i z_i^0 y_i^{\mathrm{den}}$, $\sum_i T_i$, and $\sum_i m_i T_i$. Therefore, for a newly collected batch of traces, its statistics $\Delta\mathcal{S}$ can be merged into the existing profiling state $\mathcal{S}$ simply as

$$\mathcal{S} \leftarrow \mathcal{S} + \Delta\mathcal{S}. \tag{17}$$

Once sufficient new profiling traces are accumulated, we recompute α and β using Equation 14 and re-solve Equation 15 to refresh W and δ. The resulting parameters constitute a new cost profile, while the DWS predictor itself remains unchanged.

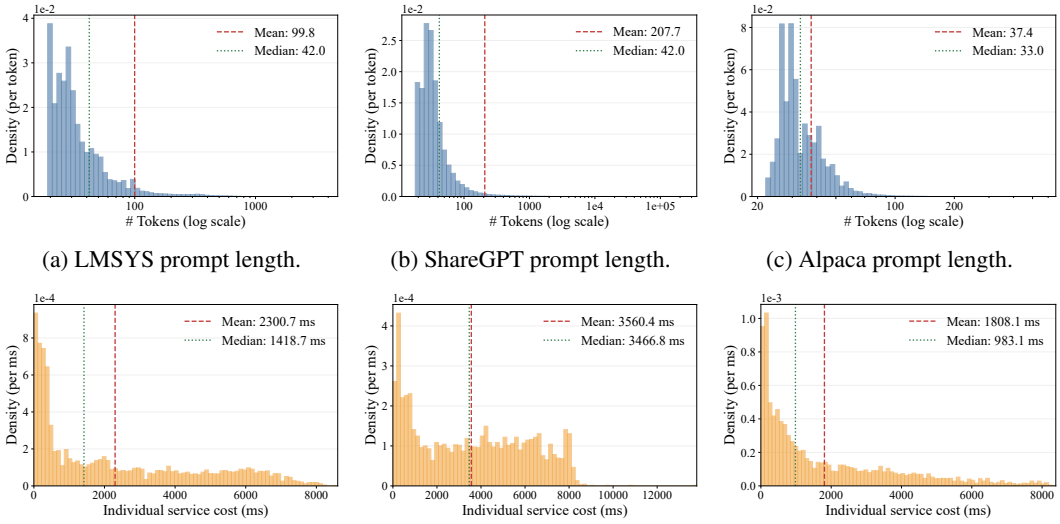
Deployment-specific profiles. Since the cost factors characterize the execution environment, each deployment configuration maintains its own cost profile. Any change to the serving model, hardware, tensor-parallel configuration, or decoding configuration therefore starts a new profile with fresh statistics.

Scope of predictor reuse. Although every such change requires a new cost profile, whether the DWS predictor must also be retrained depends on whether and how much the change alters the execution distribution $A \mid x$. (i) Hardware deployments (e.g., changes in GPU type or tensor parallelism) leave $A \mid x$ unchanged, so re-profiling alone suffices, without collecting a new predictor-training dataset or retraining the predictor. (ii) Changing the serving model within the same model family may alter $A \mid x$. Nevertheless, the experimental results in Section 6.4 and Appendix I show that the predictor remains effective with re-profiling alone (see Appendix L.2 for the analysis). (iii) In contrast, changing the core decoding configurations (e.g., block size) fundamentally reshapes the denoising structure captured by DWS, so the predictor needs to be retrained, as discussed in the limitations of Section 5. In general, transfer across hardware deployments is the most common case, which our decoupled design handles without retraining the predictor.

F Predictor Quantization and CPU Efficiency

Motivation and setup. As discussed in Section 6.4, predictor efficiency can have a substantial impact on serving performance, since delayed cost estimates can weaken cost-aware scheduling decisions. While GPU acceleration can reduce this overhead, deploying the predictor on GPUs would compete with the serving dLLM for compute and memory resources. We therefore keep the DWS predictor on CPU and further explore reduced-precision inference to improve its efficiency without substantially degrading prediction accuracy. Starting from the same trained DWS predictor, we evaluate four numerical formats, FP32, FP16, INT8, and INT4, under the same single CPU core setting. We measure both prediction accuracy, including total-step MAE, per-block step MAE, and block-count MAE, and predictor throughput.

Quantization results. As shown in Table 10, reduced-precision inference has only a minor impact on prediction accuracy. This robustness is likely because our predictor primarily relies on *coarse* prompt-level representations: token embeddings are first aggregated through pooling and then mapped to



(a) LMSYS prompt length. (b) ShareGPT prompt length. (c) Alpaca prompt length.

(d) LMSYS inference cost. (e) ShareGPT inference cost. (f) Alpaca inference cost.

Figure 10: **Characteristics of LLM serving workloads.** Top: prompt-length distributions of LMSYS-Chat-1M, ShareGPT, and Alpaca. Bottom: individual service-cost distributions of 4K requests from each dataset served by LLaDA2.0-mini. Dashed and dotted lines denote the mean and median, respectively.

workload distributions by lightweight prediction heads, while the subsequent probability aggregation further reduces sensitivity to small quantization perturbations. In particular, INT8 increases the total-step MAE from 56.06 to 56.42 (0.64%) and the final cost MAE by only 0.54% compared with FP32, while improving single-core throughput from 36.35 to 84.29 requests/s ($2.32\times$). Given this *negligible* accuracy degradation, we adopt INT8, which achieves the highest measured throughput, for all downstream serving experiments.

Interestingly, predictor throughput does not improve monotonically with lower numerical precision. This is consistent with the hardware- and kernel-dependent nature of reduced-precision CPU inference: our Zen 4 CPU provides optimized integer execution (e.g., AVX-512 VNNI), while FP16 and INT4 may rely on less efficient kernels or incur additional conversion/packing overhead. These architectural characteristics are consistent with INT8 achieving the highest measured throughput on our deployment.

G Characteristics of LLM Serving Workloads

We further characterize the workloads used in our serving evaluation from two complementary perspectives: the input prompt length and the actual per-request inference cost. For prompt length, we report the distributions over the complete LMSYS-Chat-1M, ShareGPT, and Alpaca datasets. For inference cost, we additionally profile 4K individually served requests from each dataset with LLaDA2.0-mini, using the same serving configuration as in the main experiments. Figure 10 summarizes the resulting distributions.

Prompt-length heterogeneity. The prompt distributions vary substantially across datasets. LMSYS-Chat-1M has a mean/median prompt length of $99.8/42$ tokens, while ShareGPT has the same median of 42 tokens but a substantially larger mean of 207.7, reflecting its heavier long-context tail. In contrast, Alpaca is considerably more concentrated around short prompts, with a mean/median length of $37.4/33$ tokens. These differences expose substantial distribution shifts in the input characteristics encountered by the serving system.

Inference-cost heterogeneity. The resulting dLLM inference costs are also highly heterogeneous, both within and across workloads. For LMSYS-Chat-1M and Alpaca, the mean service costs (2300.7 and 1808.1 ms) are substantially larger than their medians (1418.7 and 983.1 ms), indicating

pronounced right-skewed cost distributions. ShareGPT exhibits a generally heavier workload, with a mean/median cost of $3560.4/3466.8$ ms and requests broadly distributed over multi-second execution times. Notably, LMSYS-Chat-1M and ShareGPT have the same median prompt length of 42 tokens, yet their median inference costs differ by more than $2\times$, illustrating that similar input-length characteristics do not imply similar dLLM execution workloads.

Overall, these distributions reveal both substantial *within-workload cost heterogeneity* and clear *cross-workload distribution shifts*. The former creates opportunities for cost-aware request scheduling, while the latter motivates evaluating serving policies across diverse workloads rather than tuning them to a single request distribution.

H Predictor Training Details

We provide additional training details for the coarse-to-fine DWS predictor introduced in Section 4. Importantly, the predictor is trained entirely from request execution structure, without using measured latency or any deployment-specific cost factors α, β, or $K(\cdot)$. This keeps the learned predictor independent of the serving hardware and allows it to be reused with different deployment-specific cost profiles.

Predictor input. Due to the encoder's maximum input length of 512 tokens, we follow prior work (Qiu et al., 2024) and left-truncate longer prompts, i.e., only their last 512 tokens are fed to the encoder. The predictor takes the prompt text and a prompt-length feature as input.

Stage A: workload-oriented encoder adaptation. Stage A adapts the pretrained MiniLM encoder from generic semantic representation to dLLM denoising workload. Recall that, for a horizon h, the supervision target is $T^{(h)} = \sum_{b=1}^{n} \min(s_b, h) = \sum_{b=1}^{N_{\max}} \sum_{s=1}^{h} A_{b,s}$. We use the horizon set $\mathcal{H} = \{1, 4, 8, 12, \ldots, B\}$ and let $\hat{T}^{(h)}$ denote the corresponding prediction from the Stage-A workload head. Because the numerical ranges of $T^{(h)}$ increase with h, we normalize the prediction and target separately for each horizon before computing the loss. Denoting the normalization scale for horizon h by ν_h, we optimize

$$\mathcal{L}_{\mathrm{A}} = \frac{1}{|\mathcal{H}|} \sum_{h \in \mathcal{H}} \mathrm{SmoothL1}\left(\frac{\hat{T}^{(h)} - T^{(h)}}{\nu_h} \right), \tag{18}$$

where $\mathrm{SmoothL1}(u) = \frac{1}{2}u^2$ for $|u| < 1$ and $\mathrm{SmoothL1}(u) = |u| - \frac{1}{2}$ otherwise. This horizon-wise normalization prevents large-h targets from dominating optimization merely because of their larger scale. We do not introduce additional ranking or count losses in this stage.

Training proceeds progressively to avoid abruptly perturbing the pretrained representation. We first freeze the encoder for one epoch and train only the workload head, and then jointly optimize the head and all 12 encoder layers for another 20 epochs. The head uses a learning rate of 3×10^{-4}, whereas the encoder uses a smaller learning rate of 4×10^{-5}. For both, the learning rate is held constant for the first half of each training phase and then linearly decayed to zero without warm-up. We use a batch size of 128 and select the Stage-A checkpoint according to validation workload MAE.

For consecutive horizons $h' < h$ in $\mathcal{H}$, the difference $T^{(h)} - T^{(h')} = \sum_{s=h'+1}^{h} \sum_b A_{b,s}$ aggregates the number of blocks reaching each step $s \in (h', h]$. The multi-horizon targets therefore provide binned column sums of DWS in expectation, capturing at a coarse granularity the within-block cost heterogeneity that dominates dLLM inference cost (Appendix C), while leaving the block dimension and step-level resolution to Stage B. This is consistent with the ablation in Table 1, where replacing them with the single total-step target increases cost MAE by $1.25\times$.

Stage B: DWS refinement. Stage B replaces the coarse multi-horizon supervision with direct supervision on the final DWS. For each request, its realized activity matrix $A \in \{0,1\}^{N_{\max} \times B}$ provides a binary observation for every denoising cell, while the factorized predictor in Equation 8 produces the corresponding activity probabilities $\hat{D} \in [0,1]^{N_{\max} \times B}$. We directly optimize the Brier loss over the complete surface:

$$\mathcal{L}_{\mathrm{B}} = \frac{1}{N_{\max} B} \sum_{b=1}^{N_{\max}} \sum_{s=1}^{B} \left(\hat{D}_{b,s} - A_{b,s} \right)^2. \tag{19}$$

Table 11: **Additional online serving metrics at RPS=7** (the maximum RPS evaluated in the main-text online experiments). MiniLM-based PARS-Cost (INT8) uses the *same* encoder and CPU resources as our predictor. We report average and P90 E2E latency, TTFT, and per-token latency. Serving model is LLaDA2.0-mini. Lower is better.

Dataset	Scheduler	Statistic	E2E Latency (s)	TTFT (s)	Per-token Latency (s/token)
LMSYS-Chat-1M	FCFS	Avg	287.93	258.57	5.52
		P90	674.90	646.80	15.01
	MiniLM-based PARS-Cost (INT8)	Avg	237.12	205.28	2.73
		P90	608.73	568.39	3.54
	DWS (INT8)-SJF (Ours)	Avg	149.71	116.44	0.58
		P90	605.08	556.41	1.07
ShareGPT	FCFS	Avg	543.78	507.98	6.02
		P90	974.57	936.09	11.87
	MiniLM-based PARS-Cost (INT8)	Avg	449.27	412.05	3.50
		P90	843.76	780.19	5.00
	DWS (INT8)-SJF (Ours)	Avg	304.90	270.99	0.92
		P90	839.08	765.32	1.49
Alpaca	FCFS	Avg	124.76	107.77	2.91
		P90	217.80	201.10	9.11
	MiniLM-based PARS-Cost (INT8)	Avg	97.30	82.55	1.43
		P90	172.79	147.84	2.40
	DWS (INT8)-SJF (Ours)	Avg	61.00	44.17	0.29
		P90	116.80	72.47	0.66

This objective directly aligns training with the probabilistic interpretation of DWS, while the factorized parameterization itself guarantees the survival structure discussed in Section 4. Notably, Stage B still contains no latency-based objective; deployment-specific cell costs are only applied after prediction through the profiled cost model.

We again use progressive unfreezing. The encoder is frozen for the first four epochs, allowing the newly initialized DWS heads to adapt to the representation learned in Stage A. We then unfreeze only the top four encoder layers and jointly optimize them with the prediction heads for another four epochs. The encoder and prediction heads use learning rates of 5×10^{-6} and 2×10^{-4}, respectively. A single learning-rate schedule is maintained continuously across the eight epochs, with a 3% warm-up followed by cosine decay to 5% of the initial learning rate. The batch size is 256.

I Additional Serving Results

Additional metrics. We further report normalized latency (i.e., per-token latency), P90 (i.e., 90th percentile) latency and P90 TTFT at RPS=7, the heaviest load in Section 6.4, and additionally include the Alpaca dataset.

As shown in Table 11, DWS (INT8)-SJF consistently reduces all metrics over FCFS across three datasets. Compared with the improvement of average latency and TTFT, the gains in P90 E2E latency and TTFT are more modest, since the latency tail under SJF mainly consists of expensive requests that are deprioritized by design. Nevertheless, these tail metrics are still reduced, suggesting that, together with the aging mechanism, SJF does not improve average latency at the cost of starving expensive requests.

MiniLM-based PARS-Cost. We additionally construct a MiniLM-based PARS-Cost baseline that replaces its original BERT-base encoder with the same MiniLM encoder as DWS, applies INT8 quantization, and runs on the same single CPU core. Compared with the original BERT-based PARS-Cost, replacing its BERT encoder (110M backbone) with the smaller MiniLM (33M backbone) generally weakens its scheduling performance, likely because the smaller encoder is *less capable* of faithfully representing prompt semantics, making accurate cost prediction more difficult. Moreover, DWS (INT8)-SJF still outperforms MiniLM-based PARS-Cost across all metrics across three datasets.

Table 12: **Comparison with Oracle-SJF on SDG tasks.** Oracle-SJF schedules requests by their actual inference costs measured before, serving as an approximate *upper bound* for SJF scheduling. All predictors are trained on LMSYS-Chat-1M traces collected with LLaDA2.0-mini.

Serving Model	Dataset	Time to 1K Req. (s) ↓			Completed in 3 min ↑		
		FCFS	DWS-SJF (CPU) (Ours)	Oracle -SJF	FCFS	DWS-SJF (CPU) (Ours)	Oracle -SJF
LLaDA2.0-mini	LMSYS-Chat-1M	328	120	107	517	1299	1373
	Alpaca	197	75	62	912	1836	1959
LLaDA2.0-flash	LMSYS-Chat-1M	1598	592	531	103	310	350
	Alpaca	909	407	365	198	468	509

Table 13: **Effect of predictor encoder on DWS prediction.** All variants use the *same* predictor architecture and training setup, differing only in the encoder, and are evaluated without quantization on a single CPU core. Values in parentheses are relative to MiniLM-L12 (*Ours*); green and red denote favorable and unfavorable changes, respectively.

Encoder	Backbone Params. (M) ↓	Cost MAE (ms) ↓	Throughput on CPU (req/s) ↑
MiniLM-L6	23 (0.70×)	452.60 (1.27×)	70.72 (1.95×)
MiniLM-L12 (*Ours*)	33 (1.00×)	355.24 (1.00×)	36.35 (1.00×)
BERT-base-uncased	110 (3.33×)	338.09 (0.95×)	10.85 (0.30×)
Qwen3-0.6B (Yang et al., 2025)	600 (18.18×)	324.71 (0.91×)	2.20 (0.06×)

These results show that the gains of DWS do not merely come from a lightweight predictor, but from explicitly modeling and predicting the structured denoising cost.

Comparison with Oracle-SJF. We further compare DWS-SJF with an Oracle-SJF scheduler on the offline SDG tasks. Oracle-SJF follows the same SJF scheduling policy as DWS-SJF, but ranks requests by their *actual* inference costs measured before instead of predicted costs. Since such costs are only available after execution, Oracle-SJF is not deployable in practice and serves as an approximate *upper bound* for prediction-based SJF scheduling.

As shown in Table 12, DWS-SJF closely approaches Oracle-SJF across all settings, recovering 83.8%–94.3% of the oracle improvement over FCFS. This holds even on the unseen Alpaca dataset and LLaDA2.0-flash model, although the predictor is trained only on LMSYS-Chat-1M traces collected with LLaDA2.0-mini. In particular, the recovered fraction on LLaDA2.0-flash (83.8%–94.3%) is comparable to that on LLaDA2.0-mini (88.3%–94.1%), indicating that the DWS shift within the model family has limited impact on scheduling.

The remaining gap between DWS-SJF and Oracle-SJF stems from both prediction error and prediction latency, as Oracle-SJF knows the exact cost of every request upon arrival. Hence, the DWS predictor already captures *most* of the scheduling benefit attainable with exact cost knowledge, indicating that its prompt-only cost estimates are sufficiently accurate to guide SJF scheduling.

J Effect of Predictor Encoder

We further study how the encoder capacity affects DWS prediction accuracy and CPU efficiency. Our default predictor uses MiniLM-L12 (33M backbone), aiming to retain sufficient prompt representation capacity while remaining lightweight for CPU deployment. We replace it with MiniLM-L6, BERT-base-uncased, and Qwen3-0.6B, while keeping the remaining predictor architecture and training setup unchanged.

Results. As shown in Table 13, increasing the encoder size generally improves DWS prediction accuracy, suggesting that a higher-capacity encoder can better represent the prompt semantics relevant to the subsequent denoising behavior. Accordingly, replacing MiniLM-L12 with the smaller MiniLM-L6 increases cost MAE by 27%, although its CPU throughput improves by 1.95×. Conversely, increasing the backbone from MiniLM-L12 (33M) to BERT-base (110M) and Qwen3-0.6B (600M) further reduces cost MAE by 5% and 9%, respectively, consistent with their stronger representation capacity. However, these accuracy gains come at a substantial efficiency cost: their CPU throughput

Table 14: **Source and target deployment configurations.** The two deployments differ substantially in both hardware and system stack. The same trained DWS predictor is reused *without any retraining or fine-tuning*; only the deployment-specific cost factors are re-profiled on the target deployment.

Configuration	Source deployment	Target deployment
GPU	RTX PRO 6000	RTX A6000
GPU architecture	Blackwell	Ampere
GPU memory	96 GiB	48 GiB
GPU power limit	600 W	300 W
Tensor parallelism	TP=1	TP=2
PCIe	PCIe 5.0 $\times$16	PCIe 4.0 $\times$16
GPU interconnect	PCIe	Paired NVLink + PCIe
CPU	2$\times$ AMD EPYC 9534	2$\times$ Intel Xeon Platinum 8375C
Physical CPU cores	128	64
CPU L3 cache	512 MiB	108 MiB
AVX-512 BF16	Supported	Not reported
Host memory	256 GB	512 GB
OS	Ubuntu 24.04 LTS	Ubuntu 22.04.5 LTS
NVIDIA driver	580.126.20	550.120
Linux kernel	7.0.0-31	6.5.0-25

decreases to only $0.30\times$ and $0.06\times$ that of MiniLM-L12. These results indicate diminishing returns from scaling the encoder: larger backbones provide increasingly stronger prompt representations and modestly improve prediction accuracy, but incur rapidly growing CPU overhead. We therefore adopt MiniLM-L12 as the default encoder, which retains most of the accuracy benefit of larger encoders while remaining efficient enough for CPU-side prediction.

K Cross-Deployment Experiment

DWS explicitly decouples the request-dependent execution structure predicted from the prompt from deployment-specific execution costs. We therefore examine whether the same DWS predictor can be directly reused under a substantially different deployment by re-profiling only the deployment-specific cost factors.

Settings. We use LLaDA2.0-mini in both deployments to isolate deployment transfer from model transfer. As summarized in Table 14, the source and target deployments differ substantially in GPU architecture, tensor-parallel configuration, CPU platform, memory hierarchy, interconnect, and software stack. In particular, the source deployment uses an RTX PRO 6000 Blackwell GPU with TP=1, whereas the target deployment uses RTX A6000 Ampere GPUs with TP=2.

Importantly, we reuse the identical trained DWS predictor from the source deployment, with *no retraining or fine-tuning* on the target deployment. We only re-profile α, β, and $K(\cdot)$ on the target deployment following Appendix E.

We evaluate both online chatbot serving and offline synthetic data generation on the target deployment. For online serving, we vary RPS from 2 to 8 on LMSYS-Chat-1M and ShareGPT, and report the heaviest load (RPS=8) in Table 15. For offline SDG, we follow the main-text setting by submitting 3K requests on LMSYS-Chat-1M and Alpaca, and report the time to complete the first 1K requests and the number completed within 3 minutes.

Results. As shown in Table 15, DWS-SJF consistently improves both online and offline serving performance after transferring to the substantially different target deployment. These gains persist across different workloads and system loads, despite substantial changes in GPU architecture, tensor parallelism, and CPU platform. Importantly, the DWS predictor is reused *unchanged* throughout the transfer, and only the deployment-specific cost profile is re-profiled.

These results validate the decoupled design of DWS: the predictor captures request-dependent execution structure that remains reusable across deployments, while hardware-specific execution

Table 15: **Serving performance after cross-deployment transfer.** The DWS predictor is transferred unchanged from the source deployment, while only the deployment-specific cost profile is refreshed. Online results are reported at RPS=8, the heaviest load evaluated on the target deployment. Offline experiments submit 3K requests following the main-text protocol.

Workload	Metric	FCFS	DWS (INT8)-SJF	Improvement
Online / LMSYS-Chat-1M	Avg. E2E latency (s) ↓	1350.14	827.82	1.63×
	Avg. TTFT (s) ↓	1288.99	767.36	1.68×
Online / ShareGPT	Avg. E2E latency (s) ↓	1679.76	1105.02	1.52×
	Avg. TTFT (s) ↓	1606.41	1033.89	1.55×
Offline / LMSYS-Chat-1M	Time to 1K Req. (s) ↓	678.26	250.73	2.71×
	Completed in 3 min ↑	229	747	3.26×
Offline / Alpaca	Time to 1K Req. (s) ↓	422.22	147.95	2.85×
	Completed in 3 min ↑	443	1147	2.59×

costs are handled separately through lightweight profiling. Consequently, adapting DWS to a new deployment does not require collecting new training data or retraining the predictor.

L Theoretical Analysis of DWS

This section provides the formal analysis behind the design of DWS. Appendix L.1 quantifies the bias of the two scalar workload proxies and illustrates that of the direct-cost target with an example. Appendix L.2 proves Proposition 1, which bounds the cost-estimation error of any DWS predictor under every deployment. Appendix L.3 then shows that the factorized parameterization in Equation 8 is expressively complete.

Setup. To formalize transfer across hardware deployments, we fix the serving model, block size, and decoding configuration, so that the execution distribution $A \mid x$ is shared by all deployments considered, and deployments differ only in their profiled cost factors $\theta = (\alpha, \beta, K(\cdot))$. The limitations in Section 5 further allow changing the serving model within the same model family, as evaluated in Section 6.4; we discuss this case in the remarks of Appendix L.2.

We write $\langle \cdot, \cdot \rangle$ for the Frobenius inner product, $T = \sum_{b,s} A_{b,s}$ for the total number of denoising steps, and U for the matrix that equals one in its first column and zero elsewhere, so that $n = \langle U, A \rangle$.

Since each generated block triggers exactly one KV refresh, β can be absorbed into the first column of the cell-cost matrix. With $\widetilde{K}_\theta(m) \triangleq K(m) + \beta U$, Equation 5 becomes $C_\theta = \alpha m + \langle \widetilde{K}_\theta(m), A \rangle$, which is linear in A. Since $m(x)$ is determined by the prompt and $\mathbb{E}[A \mid x] = D(x)$, the linearity of conditional expectation gives

$$\mathbb{E}[C_\theta \mid x] = \alpha m(x) + \langle \widetilde{K}_\theta(m(x)), D(x) \rangle, \tag{20}$$

which is Equation 6 in matrix form and underlies all analyses below.

L.1 Bias of Scalar Prediction Targets

Section 1 shows empirically that output length and total denoising steps are inaccurate proxies for dLLM workload, and Section 5 attributes this to their lossy compression of DWS. Here we quantify their bias under a given deployment, and show that no fixed calibration constant makes either proxy unbiased. We then illustrate the bias of the direct-cost target under deployment changes with an example. For the two proxies, taking the conditional expectation of Equation 4 gives $\mathbb{E}[C_{\text{denoising}} \mid x] = \sum_{b,s} K_{b,s}(m(x))\, D_{b,s}(x)$.

Total denoising steps. A total-step proxy assigns the same cost $\bar{c}$ to every denoising step, regardless of its position or the prompt length.

Proposition 2 (Bias of the total-step proxy)**.** *For any constant $\bar{c}$ and any prompt x,*

$$\mathbb{E}[C_{\text{denoising}} \mid x] - \bar{c}\,\mathbb{E}[T \mid x] = \sum_{b,s} \big(K_{b,s}(m(x)) - \bar{c}\big) D_{b,s}(x). \tag{21}$$

Proof. The claim follows from $\mathbb{E}[T \mid x] = \sum_{b,s} D_{b,s}(x)$. □

Thus, the proxy is unbiased for x if and only if $\bar{c}$ equals the effective mean step cost $\mu_T(x) \triangleq \sum_{b,s} K_{b,s}(m(x)) D_{b,s}(x) / \sum_{b,s} D_{b,s}(x)$. A single constant $\bar{c}$ is therefore unbiased for all prompts only if $\mu_T(x)$ is invariant across prompts, which fails in general because both $D(x)$ and $K(m(x))$ vary across requests.

Output length. The output-length proxy further assumes that every generated block requires the same number of denoising steps $\bar{s}$, and estimates the denoising cost as $\bar{c}\bar{s}\,\widetilde{L}/B$, where $\widetilde{L} \triangleq Bn$ is the block-aligned output length.

Proposition 3 (Bias of the output-length proxy)**.** *For any constants $\bar{c}$ and $\bar{s}$ and any prompt x,*

$$\mathbb{E}[C_{\text{denoising}} \mid x] - \frac{\bar{c}\bar{s}}{B}\,\mathbb{E}[\widetilde{L} \mid x] = \underbrace{\sum_{b,s} \big(K_{b,s}(m(x)) - \bar{c}\big) D_{b,s}(x)}_{\textit{cell-cost heterogeneity}} + \underbrace{\bar{c}\,\big(\mathbb{E}[T \mid x] - \bar{s}\,\mathbb{E}[n \mid x]\big)}_{\textit{step-count heterogeneity}}. \quad (22)$$

Proof. The claim follows from $\mathbb{E}[\widetilde{L} \mid x]/B = \mathbb{E}[n \mid x]$ by adding and subtracting $\bar{c}\,\mathbb{E}[T \mid x]$. □

The first term is the bias of the total-step proxy, while the second term is specific to output length and vanishes if and only if $\bar{s} = \mathbb{E}[T \mid x]/\mathbb{E}[n \mid x]$, i.e., the expected number of denoising steps per generated block. For the actual output length $L = \widetilde{L} - R$ with $0 \le R < B$, the bias gains an additional term $\bar{c}\bar{s}\,\mathbb{E}[R \mid x]/B$, which is at most the proxy cost of a single block.

A worked example. Let $B = 4$ and let the cell costs be $K_{b,\cdot} = (5, 6, 8, 10)$ ms for every block b. Trace τ_1 generates four blocks with one denoising step each, whereas trace τ_2 generates one block with four steps, so both execute $T = 4$ steps. Their denoising costs are 20 ms and 29 ms, respectively, yet the total-step proxy assigns them the same cost. The output-length proxy is worse: since τ_1 generates four times as many blocks, it predicts τ_1 to be four times as costly, reversing the true order and thus misguiding SJF scheduling.

Direct cost. A direct-cost predictor regresses the measured cost under its training deployment $\theta_0 = (\alpha_0, \beta_0, K_0(\cdot))$, so its ideal prediction is $\mathbb{E}[C_{\theta_0} \mid x]$. Reusing this predictor on a new deployment θ without retraining requires some form of calibration. As an example, we consider the simplest calibration, which rescales the prediction by a constant $\lambda > 0$.

Proposition 4 (Bias of the direct-cost target under constant rescaling)**.** *For any deployment θ, any constant $\lambda > 0$, and any prompt x,*

$$\mathbb{E}[C_\theta \mid x] - \lambda\,\mathbb{E}[C_{\theta_0} \mid x] = (\alpha - \lambda\alpha_0)\, m(x) + \big\langle \widetilde{K}_\theta(m(x)) - \lambda \widetilde{K}_{\theta_0}(m(x)),\, D(x) \big\rangle, \quad (23)$$

where $\widetilde{K}_{\theta_0}(m) \triangleq K_0(m) + \beta_0 U$.

Proof. The claim follows by applying Equation 20 to both θ and θ_0. □

In this example, the bias vanishes if θ uniformly rescales θ_0, i.e., $\alpha = \lambda\alpha_0$ and $\widetilde{K}_\theta = \lambda \widetilde{K}_{\theta_0}$. Otherwise, the per-cell cost differences are weighted by $D(x)$, which varies across prompts, so no single λ removes the bias in general. In contrast, DWS only requires re-profiling θ (Appendix E), after which Equation 20 applies directly without retraining the predictor.

L.2 Proof of the error-control bound

We first formalize the quantities in Proposition 1. A predictor $\widehat{D}$ is any measurable function of the prompt taking values in $[0, 1]^{N_{\max} \times B}$. Following Equation 20, its predicted cost under deployment θ is $\widehat{C}_\theta(x) = \alpha m(x) + \langle \widetilde{K}_\theta(m(x)), \widehat{D}(x) \rangle$, the constant in Proposition 1 is $\kappa_\theta \triangleq \sup_m \|\widetilde{K}_\theta(m)\|_F$, and its Brier risk is

$$\mathcal{R}(\widehat{D}) \triangleq \frac{1}{N_{\max} B}\,\mathbb{E}_{x,A}\big\|\widehat{D}(x) - A\big\|_F^2, \quad (24)$$

whose empirical counterpart is the Stage-B loss in Appendix H. Next, we show that the excess Brier risk equals the mean squared distance between the predicted and true DWS.

Lemma 1 (Brier decomposition)**.** *For any predictor $\widehat{D}$,*

$$\mathcal{R}(\widehat{D}) - \mathcal{R}(D) = \frac{1}{N_{\max} B}\, \mathbb{E}_x \big\| \widehat{D}(x) - D(x) \big\|_F^2. \tag{25}$$

Hence, D is the unique minimizer of $\mathcal{R}$ up to almost-sure equality.

Proof. Given x, expanding $\|\widehat{D}(x) - A\|_F^2$ around $D(x)$ yields the cross term $2\langle \widehat{D}(x) - D(x), D(x) - A\rangle$, which vanishes under $\mathbb{E}_{A|x}$ since $\mathbb{E}[A \mid x] = D(x)$. Thus, $\mathbb{E}_{A|x}\|\widehat{D}(x) - A\|_F^2 = \|\widehat{D}(x) - D(x)\|_F^2 + \mathbb{E}_{A|x}\|D(x) - A\|_F^2$, and taking the expectation over x proves the identity. Its right-hand side is nonnegative and vanishes if and only if $\widehat{D} = D$ almost surely. □

Proof of Proposition 1. By Equation 20 and the Cauchy–Schwarz inequality,

$$\big|\widehat{C}_\theta(x) - \mathbb{E}[C_\theta \mid x]\big| = \big|\langle \widetilde{K}_\theta(m(x)), \widehat{D}(x) - D(x)\rangle\big| \le \kappa_\theta \big\|\widehat{D}(x) - D(x)\big\|_F. \tag{26}$$

Taking the expectation over x, applying Jensen's inequality $\mathbb{E}_x\|\widehat{D}(x) - D(x)\|_F \le \big(\mathbb{E}_x\|\widehat{D}(x) - D(x)\|_F^2\big)^{1/2}$, and invoking Lemma 1 yields Equation 9. □

Remarks. Proposition 1 assumes a fixed serving model and decoding configuration, as specified in the Setup. If the execution distribution changes (e.g., under a different serving model) while B and $N_{\max}$ are kept fixed, the target DWS $D'(x)$ may differ from the source DWS $D(x)$ that $\widehat{D}$ is trained to approximate. By the triangle inequality, Equation 9 then gains an additional *shift* term $\kappa_\theta\, \mathbb{E}_x\|D'(x) - D(x)\|_F$, where R remains the Brier risk on the source model. The experimental results in Section 6.4 and Appendix I show that this shift has limited practical impact within the same model family. When the predictor trained on LLaDA2.0-mini is reused on LLaDA2.0-flash without retraining, DWS-SJF still recovers most of the Oracle-SJF improvement over FCFS, comparable to that on LLaDA2.0-mini.

L.3 Expressive Completeness of the Factorized DWS

Section 4.1 shows that the factorized parameterization always satisfies the survival structure. Here we further show that it loses no expressiveness, i.e., it can represent every DWS that an execution can produce. Let

$$\mathcal{P} \triangleq \Big\{ D \in [0,1]^{N_{\max} \times B} :\ D_{1,1} = 1,\ D_{b,s+1} \le D_{b,s},\ D_{b+1,1} \le D_{b,1} \Big\} \tag{27}$$

denote the set of matrices satisfying this structure, which contains every DWS since $A_{1,1} = 1$ and both monotonicity constraints hold pointwise for A.

Proposition 5 (Expressive completeness)**.** *A matrix $D \in [0,1]^{N_{\max} \times B}$ admits the factorization in Equation 8, i.e.,*

$$D_{b,s} = \Big(\sum_{k=b}^{N_{\max}} p_k^{\text{block}} \Big) \Big(\sum_{t=s}^{B} p_{b,t}^{\text{step}} \Big) \tag{28}$$

for some distribution p^{block} over $\{1, \dots, N_{\max}\}$ and distributions p_b^{step} over $\{1, \dots, B\}$, if and only if $D \in \mathcal{P}$. Moreover, $\mathcal{P}$ is exactly the set of DWS matrices induced by some execution distribution.

Proof. Since survival probabilities are nonincreasing and start at one, every matrix of the form in Equation 28 lies in $\mathcal{P}$. Conversely, given $D \in \mathcal{P}$, let $q_b \triangleq D_{b,1}$ with $q_{N_{\max}+1} \triangleq 0$, and define $p_b^{\text{block}} \triangleq q_b - q_{b+1}$, which is a valid distribution with survival probabilities q_b. For every block with $q_b > 0$, let $D_{b,B+1} \triangleq 0$ and define $p_{b,t}^{\text{step}} \triangleq (D_{b,t} - D_{b,t+1})/q_b$, which is a valid distribution with survival probabilities $D_{b,s}/q_b$; if $q_b = 0$, then $D_{b,\cdot} = 0$ and p_b^{step} is arbitrary. Multiplying the two survival probabilities recovers $D_{b,s}$. Finally, drawing $n \sim p^{\text{block}}$ and, independently, $s_b \sim p_b^{\text{step}}$ for each block yields an execution with $\mathbb{P}(A_{b,s} = 1) = D_{b,s}$, so every $D \in \mathcal{P}$ is realizable. □

Since the expected cost depends on A only through its cell-wise activity probabilities (Equation 20), Proposition 5 implies that the factorization entails no loss for cost estimation, even though it does not model the joint distribution of A.

M Additional Related Work

Diffusion LLMs and inference frameworks. D3PM (Austin et al., 2021) formulates diffusion models over discrete state spaces, and MDLM (Sahoo et al., 2024) simplifies its absorbing-state variant into masked diffusion language models, which generate text by iteratively denoising masked tokens. Building on this paradigm, dLLMs have recently been scaled into competitive language models, such as LLaDA (Nie et al., 2025), Dream (Ye et al., 2025), and LLaDA2.0 (Bie et al., 2025). To accelerate their inference, several studies introduce dLLM-specific KV-cache mechanisms, such as Fast-dLLM (Wu et al., 2026b) and dKV-Cache (Ma et al., 2025a), which are *orthogonal* to DWS. Meanwhile, SGLang and vLLM have begun to natively support dLLMs, and dInfer (Ma et al., 2025b) provides a dedicated inference framework for them. These advances make accurate request-level cost estimation increasingly important for practical dLLM serving.

Cost-aware LLM serving. For AR LLMs, *output length* provides a natural proxy for inference cost. Accordingly, a line of work predicts either output length or its relative ordering to approximate shortest-job-first scheduling and mitigate head-of-line blocking (Qiu et al., 2024; Fu et al., 2024; Tao et al., 2026; Wang et al., 2026a). Predicted output lengths also guide batching and memory management: Zheng et al. (2023) group requests with similar predicted response lengths into the same micro-batch to reduce padding waste, while S^3 (Jin et al., 2023) allocates KV-cache memory according to predicted output lengths to enlarge the batch size and improve throughput.

Beyond non-preemptive scheduling, FastServe (Wu et al., 2026a) and TRAIL (Shahout et al., 2025) preempt running AR requests at token granularity to prioritize shorter ones. Block-autoregressive dLLMs naturally admit similar preemption at block boundaries, where no partially denoised state needs to be preserved. With its block-level fine-grained predictions (Section 5), DWS can readily estimate the remaining workload of in-flight requests to support such preemption. This makes DWS-guided preemptive scheduling a promising future work for further improving dLLM serving.