EVERY SECOND COUNTS: QUALITY-COST-AWARE LLM ROUTING FOR LOW-LATENCY CONCURRENT MULTI-LLM SERVING

Anonymous authors
Paper under double-blind review

ABSTRACT

As the Large Language Model (LLM) ecosystem grows, LLM routing is emerging as an important approach to balancing response quality and inference cost. Since real-world LLM serving involves continuously arriving requests, backlog-induced latency and quality degradation can no longer be overlooked. However, existing methods optimize only quality and cost, largely because they omit backlog-induced latency. In addition, they directly select a model without an admission mechanism, leading to quality degradation. To address these limitations, we propose *RTRouter*, a quality-cost-aware LLM routing framework that couples workload-awareness with quality admission. Specifically, we represent LLMs' unfinished work backlog as workload and update it online to avoid missing backlog-induced latency. Furthermore, we adaptively adjust the degree of pruning based on the quality vector and prune candidate models to obtain the feasible set, avoiding quality degradation. Extensive experiments on comprehensive routing benchmarks demonstrate that RTRouter improves the quality-cost-latency trade-off and effectively eliminates quality degradation.

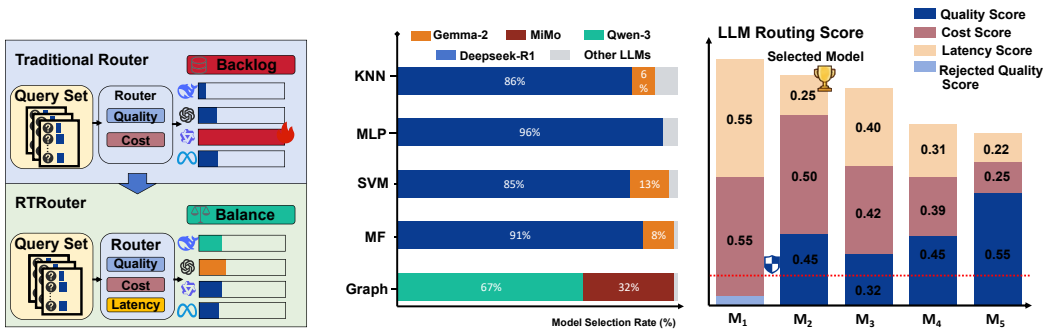
1 INTRODUCTION

The rapid expansion of large language models (LLMs) has produced a rich ecosystem that differs in scale, specialization, price, and inference time (Hu et al., 2024; Feng et al., 2025). As LLM deployments have evolved from general-purpose chat models to heterogeneous pools of specialized models, users need to select among multiple candidate backend models rather than rely on a single optimal model. LLM routing has therefore emerged as a key mechanism for assigning each query to a backend that balances response quality and inference cost (Ding et al., 2024; Ong et al., 2024). However, since real-world LLM serving involves continuously arriving requests, modern routers must account for two core issues: i) backlog-induced latency, and ii) quality degradation.

These issues stem from the widespread reliance on query accumulation and unbounded quality degradation. As illustrated in Figure 1a, traditional routers optimize for the Quality-Cost trade-off, resulting in backend LLM workload backlog, while RTRouter incorporates Latency to achieve workload balance. As shown in Figure 1b, traditional quality-cost-aware routers repeatedly select the same LLMs, concentrating traffic on a small subset of models while leaving much of the model pool underutilized. Because these decisions are made independently for each request, the resulting concentration creates localized queues and inflates end-to-end latency under continuous concurrent arrivals. RouteBalance incorporates query-queue load information and spreads traffic more effectively, but Figure 1c shows that routing exhibits a quality collapse phenomenon under high concurrency. This is because it maintains the query-queue load state globally without quality constraints, causing the latency term to grow infinitely and quality degradation under high-concurrency scenarios. *Therefore, how to optimize routers to jointly handle the backlog-induced latency and prevent quality degradation under high-concurrency scenarios remains an important challenge.*

To address this challenge, we propose RTRouter, a quality-cost-aware LLM routing framework for concurrent multi-LLM serving that combines unfinished work with quality admission. We represent each LLM's unfinished work as workload and update it online, making the backlog visible at each routing decision. Furthermore, we employ the predicted quality distribution to estimate the degree

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) Different Routing Strategies (b) Routing Distribution (c) Quality Admission

Figure 1: Comparison between traditional routers and RTRouter. (a) In high-concurrency scenarios, conventional routers optimize solely for the Quality-Cost trade-off, resulting in a backlog of backend LLM workloads. RTRouter incorporates Latency into the trade-off, achieving balanced workloads. (b) Traditional routers tend to concentrate their routing decisions on only one or two LLMs, leaving the remaining models idle. (c) RTRouter employs a quality-based admission mechanism to select only LLMs that meet the quality standard, avoiding quality degradation.

of quality crowding and adapt the degree of pruning. The degree of pruning decreases under high crowding to balance unfinished work among trusted candidates, but increases under low crowding to keep quality-degraded models out of the feasible set. This paper makes the following contributions:

- We propose RTRouter, which represents each LLM's unfinished work as workload and achieves workload-aware dispatch under concurrent multi-LLM serving.
- We adaptively adjust the degree of pruning based on the quality vector and prune candidate models to obtain the feasible set.
- We conducted extensive experiments on comprehensive routing benchmarks to show that RTRouter improves the joint quality, cost, and latency tradeoff.

2 Related Work

Workload-Unaware Routing. Large language models have developed rapidly and formed a rich catalog of heterogeneous backend models, making LLM routing a natural way to assign each query under quality and cost constraints (Hu et al., 2024). Early methods (Ding et al., 2024; Ong et al., 2024) largely followed a binary strong-versus-weak paradigm, such as Hybrid LLM and RouteLLM, which decide whether to send a query to a cheap model or an expensive one. The cascade and mixture routers then expanded the model set. For example, FrugalGPT (Chen et al., 2024) and AutoMix (Aggarwal et al., 2024) route or combine multiple models in sequence to trade quality against spend. Subsequent work (Hu et al., 2024; Li et al., 2026) treats routing as multi-class selection over a model zoo, where KNN and MLP routers choose a single backend from query features. Matrix-factorization routers (Ong et al., 2024; Hu et al., 2024) further model the interaction between query and model embeddings in a latent space, using their similarity to select one backend per request. Across this progression from binary gates to cascades, multi-class classifiers, and matrix-factorization routers, these routers remain only to balance quality and cost, without tracking backlog-induced latency under concurrent arrivals.

Latency-Dominated Routing. Since real-world LLM serving involves continuously arriving requests, backlog-induced latency can no longer be overlooked. RouteBalance (Da & Kalyvianaki, 2026) incorporates a latency term into the routing scoring function. But unlike quality and cost, backlog-induced latency will increase infinitely in high-concurrency scenarios. It will dominate the routing scoring function and outweigh quality and cost considerations. Latency causes the router to shift queries from high-quality to low-quality models, severely degrading service quality. In contrast, RTRouter constrains the quality-cost-latency trade-off strictly within a quality-feasible set, effectively preventing quality degradation.

Quality Admission. Quality admission effectively constructs the feasible set above. Although the Confidence-Driven LLM Router (Zhang et al., 2025) uses semantic entropy as a confidence metric for quality control, it is limited to binary edge-cloud switching. It serves only as a per-query verification mechanism. However, under concurrent multi-LLM serving with continuously arriving queries, quality admission must operate as a preceding gating mechanism. To achieve this, we analyze the near-best quality interval and adaptively adjust the degree of pruning for quality admission. This constructs the feasible set under complex environments and prevents quality degradation.

Although quality-cost routing, workload-aware serving, and quality admission have been explored, existing routers fail to jointly address backlog-induced latency and quality degradation under concurrent multi-LLM serving. To our knowledge, RTRouter is the first to explicitly formulate quality admission as a prerequisite to workload-aware routing in concurrent multi-LLM serving.

3 Problem Formulation

We define $\mathcal{Q}$ as the query space and $\mathcal{M} = \{m_1, \ldots, m_N\}$ as the pool of candidate LLMs. We further define $\{a_i\}$ as the query arrival times, which follow a Poisson process with rate λ. For each query $q \in \mathcal{Q}$ arriving at time a, the router must select $m^\star \in \mathcal{M}$ from the candidate LLMs. We define $Q_m(q)$, $C_m(q)$, and $T_m(q;a)$ as the quality, cost, and latency of the model m. In the admission set $\mathcal{F}_Q(q) \subseteq \mathcal{M}$, we construct a scoring function $S_m(q;a)$ based on $Q_m(q)$, $C_m(q)$, and $T_m(q;a)$. For each arriving query, the router first computes a score for every model, then compares the resulting scores across models, and finally selects the model $m^\star$ with the highest score as the routing result:

$$S_m(q;a) = w_q\, Q_m(q) + w_c\Big(1 - \frac{C_m(q)}{C_{\max}}\Big) + w_l\Big(1 - \frac{T_m(q;a)}{T_{\max}}\Big), \tag{1}$$

$$m^\star = \arg\max_{m \in \mathcal{F}_Q(q)} S_m(q;a). \tag{2}$$

In Eq. 1, $C_{\max}$ and $T_{\max}$ normalize cost and latency and (w_q, w_c, w_l) are preference weights. During the implementation of this function, two problems need to be addressed: i) how to construct $\mathcal{F}_Q(q)$ from quality predictions and ii) how to define and maintain the state of the backend workload, and how to obtain the quality, cost, and latency terms needed for Eq. 1. Section 4 develops RTRouter for these two problems.

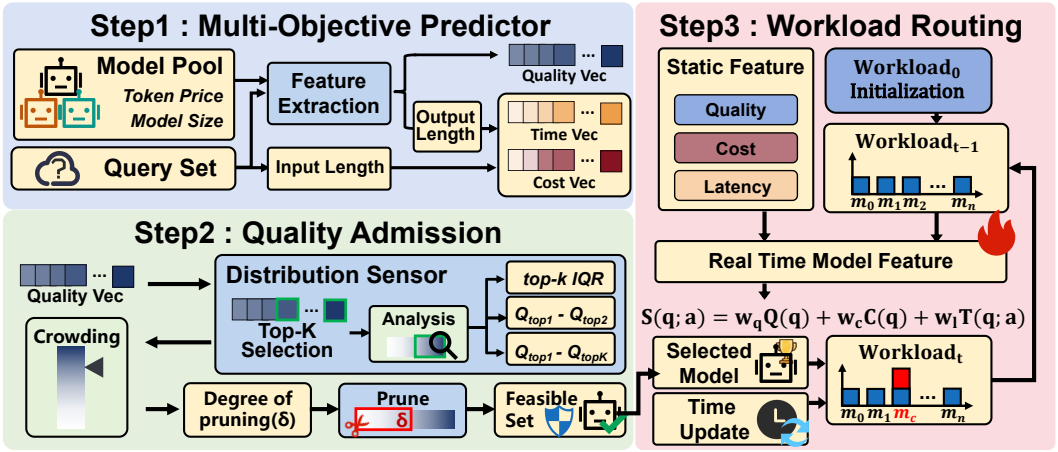


Figure 2: Overview of RTRouter.

4 Method

4.1 Overview of RTRouter

Figure 2 illustrates an overview of our RTRouter framework, which is systematically decomposed into three components: multi-objective predictor, quality admission, and workload routing. Specifically, the multi-objective predictor receives the model pool information and the query set. It applies

a pre-LN Transformer for feature extraction. This step outputs a quality vector and a length vector. The length vector serves as an intermediate state and is then combined with the input length, which is a feature of the query set, to compute the time and cost vectors. Quality admission depends on the quality vector obtained in Step 1. This quality vector is sent to the distribution sensor. The distribution sensor performs top-k selection and analysis. This sensor outputs the parameter crowding. The crowding then determines the pruning degree. The pruning mechanism then passes the original model set to obtain the feasible set. In workload routing, each LLM initializes a workload and updates it continuously as it routes. The current workload and the static feature obtained in Step 1 are combined to obtain the model's real-time feature. This feature is then added to the routing function. Within the feasible set, we obtain the selected model. Finally, the routing decision and the corresponding workload update refresh the unfinished work for the next query.

4.2 Multi-Objective Estimation Network

Feasible-set admission and unfinished work scoring cannot run on raw query text alone; they need backend estimates of quality, output length, and service time. RTRouter therefore trains the multi-objective predictor offline on routing logs and freezes its weights before deployment. Online routing then performs only a frozen forward pass, so it never retrains or updates the learned query-backend mapping under live arrivals. This keeps the online path lightweight and stable. Since the same predictions Q, L_{out}, and t are reused for every query-model pair, the feasible-set construction $\mathcal{F}_Q$ and the workload-aware routing rule can be changed independently without retraining the predictor. This decoupling lets us explore different admission strategies or routing rules at decision time while keeping the estimation module unchanged.

Feature Extraction. The Transformer takes one query and one candidate model as a pair (q, m). On the query side, a frozen MiniLM encodes the raw text into a dense vector, and a linear projection lifts that vector to width d to form the query token. On the model side, a learnable embedding table maps the model ID to a vector, and an MLP encodes model pool features such as size, price, and family or reasoner flags. The two vectors are fused into a single model token of width d. These two tokens are then stacked and fed into a Transformer encoder that applies layer normalization before each layer, so query and model information attend to each other. After encoding, the token states are flattened into one vector for the quality and length heads.

Quality and Length Heads. Two shallow MLPs read the same representation and specialize. The quality head outputs a logit ℓ_m, which we squash to a probability and blend with the offline task prior $\bar{Q}_{m,\text{task}}$. Specifically, the quality estimate Q_m is formed as a weighted combination: it takes the squashed logit $\sigma(\ell_m)$ with weight $(1 - q_{\text{prior}})$ and the task prior $\bar{Q}_{m,\text{task}}$ with weight q_{prior}. The length head does not regress the absolute length from scratch. Instead, it predicts a logarithmic residual r_m relative to the median for each model and task, $\bar{L}_{m,\text{task}}$, producing the length estimate before applying offline safety caps. In particular, the output length $L_{\text{out},m}$ is computed by exponentiating a weighted sum: the term $(1 - r_{\text{prior}})$ multiplies the sum of the log median and the residual, $(\log \bar{L}_{m,\text{task}} + r_m)$, while the term r_{prior} multiplies the log median, $\log \bar{L}_{m,\text{task}}$; the exponential of this combined expression gives $L_{\text{out},m}$. The offline pool mean $\bar{Q}_m$ is used only for the prior cut, while the query-conditioned quality estimate $Q_m(q)$ determines the query-adaptive feasible set.

Service Time. We compute service time estimates by combining model-size coefficients with the generated input and output sequence lengths. Each model m is characterized by a set of coefficients $(\kappa_m, \eta_m, \zeta_m)$ that are determined by model size, where $\eta_m \propto \sqrt{\text{size}_m}$, κ_m is a prefill fraction of η_m, and ζ_m is a model-specific fixed computational overhead. The parameter t_m is measured in the same virtual-second unit as the unfinished work W_m. For each incoming query, this stage produces a frozen bundle $\{Q_m, L_{\text{out},m}, t_m\}_{m \in \mathcal{M}}$, which encapsulates the preliminary quality estimate, output length, and service time. Crucially, what this stage deliberately does not emit is C_m or T_m. These quantities still require W_m and pricing information, which only become available after the workload state is updated in Section 4.4. In this formulation, $L_{\text{in}}(q)$ denotes the input query length, and $L_{\text{out},m}(q)$ represents the predicted output length for model m. The terms κ_m and η_m capture the prefill and decoding throughput characteristics, respectively, while ζ_m reflects the fixed computational overhead. Consequently, the service time is defined as:

$$t_m(q) = \kappa_m L_{\text{in}}(q) + \eta_m L_{\text{out},m}(q) + \zeta_m. \tag{3}$$

4.3 QUALITY ADMISSION WITH CROWDING-ADAPTIVE BANDWIDTH

Using the quality estimate Q from the multi-objective predictor, RTRouter next addresses the first interface in Section 3 by determining which LLMs are quality-eligible before computing any workload-aware score. This step is crucial because it establishes a quality-qualified candidate pool before introducing any workload or latency signals. Without that gate, a latency-heavy utility evaluated across the entire model pool can favor idle weak LLMs simply because their queues are empty, ignoring their inherent quality deficiencies, and the later unfinished-work step cannot fix the mistake. Consequently, RTRouter builds the feasible set $\mathcal{F}_Q(q)$ in this subsection and deliberately leaves the scoring metric S_m to Section 4.4. This cleanly separates quality admission from workload routing, ensuring that quality-qualified models enter the subsequent scoring and selection process.

Prior Cut. The first cut uses an offline pool mean $\bar{Q}_m$ for each backend, distinct from the task-conditioned head prior $\bar{Q}_{m,\text{task}}$. This pool-level statistic is query-independent and serves as a computationally cheap prefilter for the quality admission stage. Specifically, it keeps only models at or above the α-quantile of $\{\bar{Q}_m\}$ over the pool, with α=0.10. Formally, the prior candidate set is defined as $\mathcal{M}_{\text{prior}} = \{m \in \mathcal{M} : \bar{Q}_m \geq Q_{(\alpha)}\}$, which also acts as a default fallback if the subsequent feasible set is empty, as outlined in Algorithm 1.

Feature Analysis of the Near-top Head. Within $\mathcal{M}_{\text{prior}}$, RTRouter examines the shape of the predicted quality scores instead of treating each score list as uniformly sharp. It sorts these scores from high to low and restricts attention to a small top set of size $k = 5$. Four signals then describe proximity near the best score. They are the gap between the top two scores, the gap from the best score to the last score in that top set, the interquartile range among the top-k scores, and the count of models close to the best score. We convert these signals to unit ambiguity scores and combine them into a single crowding index $u(q) \in [0, 1]$. A large u indicates that several models are closely matched at the top, while a small u indicates that a clear leader stands above the rest. Formally, this crowding index is defined as:

$$u(q) = \Phi(\Delta_{12}, \Delta_{1k}, \text{IQR}_k, N_{\text{close}}) \in [0, 1], \tag{4}$$

where Δ_{12} is the gap between the top two scores, Δ_{1k} is the gap from the best score to the k-th score, IQR_k measures the dispersion within the top-k set, and N_{close} denotes the number of models close to the best score. Φ is a fixed convex combination $\Phi = 0.35\, u_{12} + 0.30\, u_{1k} + 0.20\, u_{\text{IQR}} + 0.15\, u_N$ of the four unit ambiguity scores.

Crowding to Bandwidth. The crowding index $u(q)$ computed in the previous near-top head analysis serves a specific, limited role. It sets only the bandwidth of the near-best set, dynamically adjusting how many top-ranked models it retains for subsequent steps. Importantly, it never replaces admission, ensuring that the quality gate established in Section 4.3 remains strictly intact. Using held-out anchors for the crowding index u, RTRouter maps u to a spread coefficient $s(q) \in [0, 1]$, and linearly interpolates the bandwidth endpoints β, $\delta_{\min}$, and $\delta_{\max}$ between a locked narrow setting and a locked wide setting. Specifically, we precompute these locked settings offline and keep them fixed across all queries, serving as the interpolation extremes. The logic is direct: a large u shrinks the band to protect ranking, while a small u widens the band so that Section 4.4 can balance unfinished work among trusted models in the near-best set. Formally, the interpolation of these bandwidth parameters is defined as:

$$\phi(q) = \phi_{\text{narrow}} + s(q)\big(\phi_{\text{wide}} - \phi_{\text{narrow}}\big), \quad \phi \in \{\beta, \delta_{\min}, \delta_{\max}\}, \tag{5}$$

where ϕ_{narrow} and ϕ_{wide} denote the pre-computed offline endpoints for each respective bandwidth parameter ϕ, and are locked as fixed hyperparameters before online routing begins. The spread coefficient $s(q)$ is obtained via a monotonic mapping $s(q) = \mathcal{S}(u(q))$. The resulting parameters from Eq. 5 are then directly consumed to build the feasible set $\mathcal{F}_Q(q)$, which completes the quality admission stage before the scoring metric S_m is evaluated in Section 4.4.

Near-Best Feasible Set. Given the interpolated parameters from Section 4.3, RTRouter now determines the query-adaptive bandwidth. This bandwidth is derived from the interquartile range of the predicted quality scores $\{Q_m(q)\}$ within $\mathcal{M}_{\text{prior}}$, and is subsequently clipped into the range $[\delta_{\min}(q), \delta_{\max}(q)]$. The calculation is formally defined in Eq. 6:

$$\delta(q) = \text{clip}\big(\beta(q) \cdot \text{IQR}(\{Q_{m'}(q)\}_{m' \in \mathcal{M}_{\text{prior}}}),\, \delta_{\min}(q), \delta_{\max}(q)\big), \tag{6}$$

where $\delta(q)$ represents the final, query-adaptive tolerance margin around the best score. This feasible set serves as the final output of the quality admission stage. Every LLM with an admitted name

then re-enters $\mathcal{F}_Q$, while rejected names are barred from S_m. Under this design, the prior blocks historically weak models, the top-k and margin stack diagnoses head crowding, and the IQR band follows the local geometry of the current prediction. Consequently, the utility scoring in Eq. 2 is still computed only inside $\mathcal{F}_Q(q)$. Models outside the set never enter utility scoring even if they are idle, ensuring that the quality gate remains strictly enforced before any workload-aware routing occurs. Therefore, the quality-eligible set is formally constructed as:

$$\mathcal{F}_Q(q) = \big\{ m \in \mathcal{M}_{\text{prior}} : Q_m(q) \geq \max_{m' \in \mathcal{M}_{\text{prior}}} Q_{m'}(q) - \delta(q) \big\}. \tag{7}$$

4.4 Workload Routing inside the Feasible Set

Once the quality-eligible set $\mathcal{F}_Q(q)$ is fixed by the admission stage, the gateway still needs to address the second interface from Section 3. This interface provides the workload state along with the cost and time terms required by the utility function in Eq. 1. We continuously track the workload state to reflect the current unfinished work, while dynamically assembling the cost and time terms from the predicted outputs. We define these quantities below and then select the final model by the restricted $\arg\max$ over $\mathcal{F}_Q(q)$ specified in Eq. 2. This ensures that the routing decision is made only among quality-qualified models, effectively balancing the trade-off between quality, cost, and latency.

Unfinished Work Dynamics. Each model keeps unfinished work $W \geq 0$, so the gateway does not need per-request first-in, first-out queues. This abstraction avoids the overhead of explicit queue management for each incoming request, thereby keeping online routing lightweight. The unfinished work W and the service time t_m share the same virtual-second unit, ensuring state updates and time predictions align on the same temporal scale. Under Poisson arrivals $\{a_i\}$ of rate λ, a query that arrives at time a first advances the clock. For every LLM, the gateway drains the existing unfinished work by subtracting the elapsed virtual time since the last clock update, namely $a - t_{\text{now}}$. This subtraction is lower-bounded at zero, which prevents idle LLMs from accumulating negative work. Afterward, the clock is advanced to the current arrival time $t_{\text{now}} \leftarrow a$, guaranteeing that subsequent updates are processed on a monotonic timeline. Consequently, idle LLMs stay at zero, and busy LLMs shed work as the arrival clock advances. This state transition is applied to every LLM and can be summarized as follows:

$$W = \max\big(0,\, W - (a - t_{\text{now}})\big), \qquad t_{\text{now}} \leftarrow a. \tag{8}$$

Cost-aware Selection Inside the Feasible Set. After the workload state is drained according to Eq. 8, the frozen predictors provide the service time $t_m(q)$ and the predicted output length $L_{\text{out},m}(q)$ for each candidate model. We then combine these predicted quantities with the refreshed unfinished work W and the per-token prices from the model pool p_m^{in} and p_m^{out}. This combination gives the latency time and the monetary cost for each model, which are formally defined as:

$$T = W + t_m(q), \qquad C_m(q) = L_{\text{in}}(q)\, p_m^{\text{in}} + L_{\text{out},m}(q)\, p_m^{\text{out}}, \tag{9}$$

where T represents the latency time accounting for the current backlog, while $C_m(q)$ captures the estimated cost of processing the query on model m. These are precisely the latency and cost arguments required by the utility function in Eq. 1. Scoring is confined strictly to LLMs whose model names lie in the feasible set $\mathcal{F}_Q(q)$, under the locked preference weights $(w_q, w_c, w_l) = (0.35, 0.30, 0.35)$. Within this restricted set, $C_{\max}$ and $T_{\max}$ normalize the cost and time terms, and the gateway returns the optimal model $m^\star$ by Eq. 2. Because T already includes the backlog, a smaller W naturally improves the latency score and pulls traffic toward less work-loaded LLMs near the best LLMs. At the same time, models outside $\mathcal{F}_Q(q)$ cannot be selected even if they are idle, ensuring that the quality admission gate remains strictly enforced throughout the routing decision.

Booking and Prediction-only Feedback. After the choice is made, the gateway immediately books the predicted work and returns the decision without waiting for ground-truth latency or quality feedback. The reported latency uses the same T that entered S_m, and neither the service-time coefficients $(\kappa_m, \eta_m, \zeta_m)$ nor the predictor weights are updated online. Consequently, the only state carried across queries is the unfinished work vector across the model pool, which is updated via the drain, score, and book operations on every arrival. The booking step adds the predicted service time of the selected model to its unfinished work:

$$W = W + t_{m^\star}(q). \tag{10}$$

4.5 IMPLEMENTATION OF RTROUTER

Algorithm 1 lists the full RTRouter gateway. The inputs are an offline training set $\mathcal{Q}_{\text{train}}$, a candidate backend pool $\mathcal{M}$ with prices and sizes, an online arrival stream $\{(q_i, a_i)\}$, a Transformer predictor g_θ, the locked hyperparameters α and $(u_{\text{lo}}, u_{\text{hi}})$, the narrow and wide bandwidth endpoints $(\beta^{\text{n}}, \beta^{\text{w}})$, $(\delta^{\text{n}}_{\min}, \delta^{\text{w}}_{\min})$, $(\delta^{\text{n}}_{\max}, \delta^{\text{w}}_{\max})$, and the preference weights (w_q, w_c, w_l). The output is the set of routed backends $\mathcal{M}^\star$. The gateway first operates offline to fit the service-time coefficients $(\kappa_m, \eta_m, \zeta_m)$ on $\mathcal{Q}_{\text{train}}$, trains and freezes the predictor g_θ, and initializes unfinished work and clock to zero to establish an initial zero-load state. It then processes each arrival $(q, a) \in \{(q_i, a_i)\}$. For each, it drains the unfinished work of all $m \in \mathcal{M}$ using the virtual clock and bounds it at zero, then updates the clock accordingly. Subsequently, it runs the frozen predictor to obtain the predicted bundle $\{Q_m(q), L_{\text{out},m}(q), t_m(q)\}$. To construct the quality-feasible set, it applies a prior threshold with α to filter a prior subset, computes crowding $u(q)$ and spread $s(q)$ from the predicted distribution, and interpolates the locked bandwidth endpoints to build the feasible set $\mathcal{F}_Q(q)$. If empty, the gateway falls back to the prior subset. Finally, for scoring and routing, it evaluates only models in the feasible set using (w_q, w_c, w_l), selects $m^\star$ by Eq. 2, and books its predicted service time into the unfinished work of the selected model. After handling each arrival, it returns the set of routed backends $\mathcal{M}^\star$, and carries only the unfinished work vector across queries. This keeps online routing lightweight while dynamically adapting to real-time workloads.

Algorithm 1 Implementation of RTRouter

Input:
i) $\mathcal{Q}_{\text{train}}$, offline training query dataset;
ii) $\mathcal{M}$, LLM pool with prices and sizes;
iii) $\{(q_i, a_i)\}$, online stream of the queries q_i reaching the gateway at arrival time a_i;
iv) $g_\theta(\cdot)$, predictor parameterized by θ;
v) α, $(u_{\text{lo}}, u_{\text{hi}})$, $(\beta^{\text{n}}, \beta^{\text{w}})$, $(\delta^{\text{n}}_{\min}, \delta^{\text{w}}_{\min})$, $(\delta^{\text{n}}_{\max}, \delta^{\text{w}}_{\max})$, (w_q, w_c, w_l), hyperparameters.
Output:
The routed backend set $\mathcal{M}^\star$.

/* Offline predictors (Sec. 4.2) */
$(\kappa_m, \eta_m, \zeta_m)$ in $\mathcal{Q}_{\text{train}}$; train and freeze g_θ
/* State update and drain (Sec. 4.4) */
$W \leftarrow 0,\ t_{\text{now}} \leftarrow 0$
$\mathcal{M}^\star \leftarrow \emptyset$
for $(q, a) \in \{(q_i, a_i)\}$ **do**
  for $m \in \mathcal{M}$ **do**
    $W_m \leftarrow \max(0, W_m - (a - t_{\text{now}}))$
  end for
  $t_{\text{now}} \leftarrow a$
/* Prediction and admission (Sec. 4.3) */
  $\{Q_m(q), L_{\text{out},m}(q), t_m(q)\} \leftarrow g_\theta(q, \mathcal{M})$
  $u(q) \leftarrow \Phi(\cdot),\ s(q) \leftarrow \mathcal{S}(u(q))$
  $(\beta, \delta_{\min}, \delta_{\max}) \leftarrow s(q), (u_{\text{lo}}, u_{\text{hi}})$
  $\mathcal{F}_Q(q) \leftarrow (\delta_{\min}, \delta_{\max})$
  $\mathcal{F}_Q(q) \leftarrow \mathcal{F}_Q(q) \neq \emptyset\ ?\ \mathcal{F}_Q(q) : \mathcal{M}_{\text{prior}}$
/* Scoring and Routing (Sec. 4.4) */
  $S_m(q; a) \leftarrow w_q Q_m(q) + w_c(1 - C_m(q)/C_{\max}) + w_l(1 - T_m(q; a)/T_{\max})$
  $m^\star \leftarrow \arg\max_{m \in \mathcal{F}_Q(q)} S_m(q; a)$
  $\mathcal{M}^\star \leftarrow \mathcal{M}^\star \cup \{m^\star\}$
  $W_{m^\star} \leftarrow W_{m^\star} + t_{m^\star}(q)$
end for
return $\mathcal{M}^\star$

5 EXPERIMENTS

To evaluate RTRouter's effectiveness, we implemented the routing process in PyTorch (Paszke et al., 2019). We conducted all experiments on a workstation with a single NVIDIA RTX 4090 GPU. Appendix A provides comprehensive experimental details and dataset settings, covering the LLM-RouterBench benchmark and the candidate model pool.

5.1 EXPERIMENTAL SETUP

Dataset Settings. All experiments used LLMRouterBench (Li et al., 2026), a comprehensive multi-LLM routing benchmark with a shared pool of candidates. The benchmark provides quality labels, output lengths, and pricing information for each query, covering a wide range of domains.To simulate concurrent multi-LLM serving evaluation, the benchmark compares routers across diverse backend models using the same evaluation data. All compared routing methods used a shared Poisson arrival process to simulate request arrivals, ensuring identical input information between routers and allowing any differences to be attributed to the routing methods themselves. Each comparison ran a

sample of n=300 evaluation queries at each of three arrival rates, $\lambda \in \{1, 2, 8\}$. All reported results are averaged over 10 independent runs with different random seeds.

Baseline Settings. We compared RTRouter with the following routers. i) workload-unaware routers: MLPRouter (Hu et al., 2024), MFRouter (Hu et al., 2024), KNNRouter (Hu et al., 2024; Li et al., 2026), GraphRouter (Feng et al., 2025), AlwaysLargest (Hu et al., 2024; Feng et al., 2025), and AlwaysSmallest (Hu et al., 2024; Feng et al., 2025). These methods select an LLM based on query features, latent query-model compatibility, historical similarity, and other signals. However, they lack load-state awareness and ignore backlog-induced latency caused by concurrent query arrivals; ii) RouteBalance (Da & Kalyvianaki, 2026): a router that is aware of LLM states but lacks a quality admission mechanism; and iii) Oracle: a router that always selects the quality-maximizing backend for each query based on ground-truth labels, without considering cost and latency.

Evaluation Metrics. We evaluated RTRouter using the following metrics. i) Accuracy: the fraction of queries for which the selected LLM's quality label matches the Oracle's top-quality label, which measures the average quality performance of LLMs under the experimental conditions; ii) Cost: the total spending on backend LLMs over the run, in USD, based on per-million-token prices; iii) Mean Latency: the average predicted latency over all evaluation queries, which measures the backlog state during the routing process; and iv) p99: the 99th percentile of predicted latency during each experimental run, which measures tail latency under concurrent traffic.

Candidate Model Pool. We adopted the LLMRouterBench model pool (Li et al., 2026), which contains 20 heterogeneous open-source LLMs with price and size information. It covers general instruction, reasoning, code-specialized, and domain-adapted models across multiple families, reducing reliance on a single developer or task domain. All routers use the same pool, queries, and Poisson arrivals but have their own selection mechanisms. We evaluated mean latency, p99, and cost under a common scoring rule. Appendix C lists the model and its selection count.

5.2 Experimental Results

Table 1 summarizes the routing results of the routers compared under the above experimental setup. Workload-unaware routers achieve moderate accuracy, but perform markedly worse on mean latency and p99. This is because workload-unaware routers only balance cost and quality and ignore latency. As a result, only a few LLMs are selected most of the time, and backlog-induced latency markedly increases p99. RouteBalance performs worse than most load-unaware routers. This degradation becomes more pronounced under high workload, indicating that poor admission reduces route accuracy. Although Oracle is quality-optimal by construction, it offers little practical value once cost and latency are considered. RTRouter is the only method that achieves high accuracy while keeping Cost, Mean Latency, and p99 relatively low.

Table 1: Quality-cost-latency comparison on LLMRouterBench under concurrent Poisson arrivals.

Method	Accuracy↑	Cost↓ ($)	Mean↓ (s)	p99↓ (s)
Oracle	1.00 / 1.00 / 1.00	0.039 / 0.039 / 0.039	1886 / 1907 / 1959	4899 / 5018 / 5135
AlwaysSmallest	0.61 / 0.61 / 0.61	0.034 / 0.034 / 0.034	2438 / 2531 / 2591	4816 / 4992 / 5109
AlwaysLargest	0.75 / 0.75 / 0.75	0.064 / 0.064 / 0.064	2425 / 2520 / 2580	4809 / 4975 / 5093
MLPRouter	0.77 / 0.77 / 0.77	0.040 / 0.040 / 0.040	2360 / 2583 / 2640	5514 / 5875 / 5992
MFRouter	0.76 / 0.76 / 0.76	0.042 / 0.042 / 0.042	2865 / 3102 / 3160	6111 / 6463 / 6580
KNNRouter	0.77 / 0.77 / 0.77	0.041 / 0.041 / 0.041	2481 / 2710 / 2766	5665 / 6030 / 6146
GraphRouter	0.72 / 0.72 / 0.72	0.055 / 0.055 / 0.055	1768 / 1984 / 2043	4141 / 4550 / 4667
RouteBalance	0.74 / 0.72 / 0.73	0.035 / 0.032 / 0.033	15 / 20 / 54	138 / 181 / 252
RTRouter	0.82 / 0.79 / 0.79	0.019 / 0.020 / 0.021	9 / 11 / 25	25 / 33 / 98

5.3 Ablation Study

The complete RTRouter integrates workload-awareness and quality admission mechanisms. To assess their distinct contributions, we ablated workload-awareness and quality admission separately. We conducted the ablation experiments under the same experimental setup as in the main experiment. Table 2 summarizes the ablation results and compares them with the baseline routers. The detailed analysis is provided below.

RTRouter (without workload-awareness). When we remove the workload-awareness mechanism, Mean Latency and p99 of RTRouter increase markedly because, without W, RTRouter estimates latency based only on the current query's processing time and ignores the backlog induced by concurrent requests in high-concurrency scenarios. Moreover, Cost slightly decreases in this ablated setting, as the router begins selecting cheaper, smaller models. This behavior matches workload-unaware routers in the baselines. In addition, RouteBalance even outperforms RTRouter (without workload awareness) in Mean Latency and p99. This advantage arises because RouteBalance employs queues to distribute queries, reducing backlog-induced latency. This analysis shows why RTRouter needs workload awareness.

RTRouter (without quality admission). When we remove the quality admission mechanism, RTRouter's accuracy drops sharply. Without quality admission, RTRouter selects LLMs based on predicted quality, cost, and latency, while ignoring quality degradation in high-concurrency scenarios. Compared with other routers, RTRouter improves Cost, Mean Latency, and p99. However, this shows that the router degrades quality and pushes routing toward fast but unreliable responses. Specifically, idle and weak models outperform other routers because of their lower latency and cost. This analysis shows why RTRouter needs a quality admission mechanism.

RTRouter (fixed δ). We further ablate the crowding-adaptive threshold by fixing δ. A small δ enforces a strict quality gate, preserving accuracy but forcing selection of stronger, more expensive models, which raises cost, mean latency, and p99. As δ increases, the gate relaxes, lowering cost and latency but degrading accuracy by admitting weaker models. No single fixed δ matches the adaptive router: each value trades accuracy for efficiency, and the trade-off depends on workload. The adaptive router adjusts the threshold with observed crowding, being permissive at low load and conservative when queues build. A fixed threshold is thus unnecessarily conservative at low load and insufficiently selective at high load. This shows that the quality gate must be dynamic rather than static: only by adapting δ to real-time congestion can RTRouter balance accuracy, cost, and latency across arrival rates. A fixed δ inevitably fails on one side of the workload spectrum. For instance, fixing $\delta = 0.04$ yields accuracy comparable to RTRouter but incurs higher cost, mean latency, and p99, while a smaller δ would further inflate latency and a larger δ would degrade accuracy. This confirms that adaptive thresholds are essential.

Table 2: Ablation and baseline comparison on LLMRouterBench.

Method	Accuracy↑	Cost↓ ($)	Mean Latency↓ (s)	p99↓ (s)
Oracle	1.000	0.0395	1949.77	5066.79
KNNRouter	0.790	0.0402	2554.49	5832.85
MFRouter	0.770	0.0406	2944.74	6278.82
MLPRouter	0.780	0.0382	2439.13	5681.92
RouteBalance	0.740	0.0331	18.21	172.38
RTRouter	0.813	0.0190	10.10	30.21
RTRouter (without workload-awareness)	0.813	0.0186	79.58	391.71
RTRouter (without quality admission)	0.667	0.0038	3.68	10.01
RTRouter (fixed $\delta = 0.04$)	0.817	0.0243	13.67	55.12

6 Conclusion

In this paper, we introduced RTRouter, a quality-cost-aware LLM routing for low-latency concurrent multi-LLM serving. By incorporating workload-awareness and quality admission mechanisms, RTRouter effectively resolved backlog-induced latency and quality degradation in high-concurrency scenarios. Extensive experiments on LLMRouterBench under a shared Poisson arrival process and multiple arrival rates show that RTRouter consistently improves the joint quality, cost, and latency tradeoff over competitive baselines, achieving higher accuracy while lowering cost, mean latency, and p99 tail latency. The ablation study shows that workload awareness and quality admission mechanisms are indispensable components of RTRouter. A primary limitation is that we depend on predicted latency, rather than actual multi-backend latency measured in wall-clock time. Future work will couple quality admission with system-level feedback in live serving stacks.

AI USE STATEMENT

In this work, we used generative AI tools only to polish the language and correct grammar in the manuscript. The authors reviewed all AI-assisted edits and take full responsibility for the final content of this work.

ETHICS STATEMENT

This work does not involve human subjects, personal data, or sensitive information. All experiments use publicly available benchmarks, and we collect, use, or release no private or identifiable data.

REPRODUCIBILITY STATEMENT

We describe the proposed RTRouter framework and its online routing procedure in detail. We fully document the experimental setup, hyperparameters, model pool, prices, sizes, and routing distributions. We also provide baseline implementation details and evaluation limitations. We will release code and configuration files upon publication. All datasets are publicly available through LLM-RouterBench.The supplementary material provides implementation details.

REFERENCES

Pranjal Aggarwal, Aman Madaan, Ankit Anand, Srividya Pranavi Potharaju, Swaroop Mishra, Pei Zhou, Aditya Gupta, Dheeraj Rajagopal, Karthik Kappaganthu, Yiming Yang, Shyam Upadhyay, Manaal Faruqui, and Mausam. Automix: Automatically mixing language models. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in LLM inference with sarathi-serve. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation(OSDI)*, pp. 117–134, 2024.

Lingjiao Chen, Matei Zaharia, and James Zou. Frugalgpt: How to use large language models while reducing cost and improving performance. *Trans. Mach. Learn. Res.*, 2024, 2024.

Wei Da and Evangelia Kalyvianaki. Routebalance: Fused model routing and load balancing for heterogeneous LLM serving. *arXiv preprint arXiv:2606.17949*, 2026.

Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Rühle, Laks V. S. Lakshmanan, and Ahmed Hassan Awadallah. Hybrid LLM: cost-efficient and quality-aware query routing. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

Tao Feng, Yanzhen Shen, and Jiaxuan You. Graphrouter: A graph-based router for LLM selections. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025.

Qitian Jason Hu, Jacob Bieker, Xiuyu Li, Nan Jiang, Benjamin Keigwin, Gaurav Ranganath, Kurt Keutzer, and Shriyash Kaustubh Upadhyay. Routerbench: A benchmark for multi-llm routing system. *arXiv preprint arXiv:2403.12031*, 2024.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pp. 611–626, 2023.

Hao Li, Yiqun Zhang, Zhaoyan Guo, Chenxu Wang, Shengji Tang, Qiaosheng Zhang, Yang Chen, Biqing Qi, Peng Ye, Lei Bai, Zhen Wang, and Shuyue Hu. Llmrouterbench: A massive benchmark and unified framework for LLM routing. In *Proceedings of the Association for Computational Linguistics (ACL)*, pp. 37733–37754, 2026.

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M. Waleed Kadous, and Ion Stoica. Routellm: Learning to route llms with preference data. *arXiv preprint arXiv:2406.18665*, 2024.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, pp. 8024–8035, 2019.

Tuo Zhang, Asal Mehradfar, Dimitrios Dimitriadis, and Salman Avestimehr. Leveraging uncertainty estimation for efficient LLM routing. *arXiv preprint arXiv:2502.11021*, 2025.

A EXPERIMENTAL SETUP AND DATASET

Table 3: Evaluation dumps and configurations on LLMRouterBench.

Dump	n	λ	Seeds	Notes
`router_snapshot`	300	2	$\{0,\dots,9\}$	peer snapshot
`prune_ablation`	300	2	$\{0,\dots,9\}$	prune / W ablation

Dataset. All quantitative claims use LLMRouterBench (Li et al., 2026), a unified multi-LLM routing benchmark with per-query labels (performance, output tokens, prices) over a shared candidate pool. We sample n=300 evaluation queries under a Poisson arrival process on a predicted-latency simulator; we freeze predictors before comparative routing so peers share identical Q, L_{out}, and t.

Locked RTRouter. We instantiate RTRouter with a Transformer-based predictor and enable unfinished-work tracking. The quality admission stage uses a prior cut with $\alpha = 0.10$, followed by an IQR-based near-best bandwidth. The near-top analysis uses $k = 5$. For the crowding-adaptive bandwidth, the crowding index $u(q) \in [0,1]$ is mapped through a monotone calibration $s(q) = \mathcal{S}(u(q))$ using data-calibrated anchors $(u_{\text{lo}}, u_{\text{hi}})$. The bandwidth parameters are then interpolated between two locked endpoints according to Eq. 5:

$$(\beta_{\text{narrow}}, \beta_{\text{wide}}) = (0.30, 0.80),$$
$$(\delta_{\text{min,narrow}}, \delta_{\text{min,wide}}) = (0.04, 0.06),$$
$$(\delta_{\text{max,narrow}}, \delta_{\text{max,wide}}) = (0.045, 0.12).$$

Here, the narrow setting corresponds to high crowding and the wide setting corresponds to low crowding, so that large $u(q)$ shrinks the admissible near-best band and small $u(q)$ widens it. The predictors are calibrated offline without ground-truth feedback, and the preference weights are set to

$$(w_q, w_c, w_l) = (0.35, 0.30, 0.35).$$

Main experiments in Section 5.2 use these locked preference weights.

B BASELINE ROUTERS

We evaluate all non-workload-aware routers through the LLMRouter codebase on the same Bench dumps and candidate pool, and none maintain unfinished work W_m.

MLPRouter. A multi-layer perceptron maps query features, including text embeddings and hand-crafted features following RouterBench-style pipelines (Hu et al., 2024), to a score per candidate LLM and selects $\arg\max_m$ score. It is a lightweight learned ranker with no concurrency state.

MFRouter. Matrix factorization embeds queries and models in a shared latent space and scores compatibility by inner product (Hu et al., 2024). It improves multi-model coverage relative to purely preference-based ranking but still concentrates heavily under concurrency.

KNNRouter. A k-nearest-neighbor estimator retrieves similar training queries and aggregates their labeled backend performances (Hu et al., 2024). It is a strong non-workload-aware accuracy baseline in our tables, yet still defaults to DeepSeek-R1 under workload.

GraphRouter. A graph neural router over query-model interaction structure (Feng et al., 2025) predicts edge scores and selects the best neighbor. It is structurally richer than MLP or KNN but remains non-workload-aware with no W_m.

AlwaysLargest and AlwaysSmallest. These size heuristics fix one catalog endpoint for all queries (Hu et al., 2024; Feng et al., 2025). AlwaysLargest always picks the largest model, and AlwaysSmallest always picks the smallest model, so they stress pure size bias without query-adaptive ranking.

RouteBalance. A concurrent balancer that fuses model choice with token-level workload spreading (Da & Kalyvianaki, 2026). It reduces latency time compared to non-workload-aware routers but does not build a query-specific near-best set $\mathcal{F}_Q$.

Oracle and other heuristics. Oracle selects, for each query, the backend with the highest ground-truth quality label, and it is quality-optimal with no cost or latency objective. BestSingle fixes one strong catalog model for all queries and cycles uniformly over the 20-model pool.

C MODEL ROUTING DISTRIBUTIONS

Table 4 lists the top-5 routed LLMs for KNNRouter and RTRouter at λ=2. KNN sends $258/300$ queries to DeepSeek-R1, while RTRouter's top backend (gemma-2-9b-it) receives only $44/300$, indicating a much flatter routing distribution. RTRouter routes to 17 distinct LLMs in total; we show only the top-5 here for brevity.

Table 4: Routing distributions of KNNRouter and RTRouter.

KNNRouter		RTRouter	
DeepSeek-R1-0528-Qwen3-8B	258	gemma-2-9b-it	44
gemma-2-9b-it	19	NVIDIA-Nemotron-Nano-9B-v2	40
granite-3.3-8b-instruct	9	Qwen2.5-Coder-7B-Instruct	31
NVIDIA-Nemotron-Nano-9B-v2	3	Intern-S1-mini	30
Qwen2.5-Coder-7B-Instruct	3	Qwen3-8B	25

D DISCUSSION AND LIMITATIONS

RTRouter targets concurrent heterogeneous multi-LLM gateways where Poisson arrivals create backlog. Decoupling $\mathcal{F}_Q$ from unfinished-work dispatch prevents traffic from concentrating on strong models and keeps idle weak LLMs available. Relative to traditional routers, RTRouter spreads workload across 17 LLMs while improving accuracy; relative to RouteBalance, it preserves near-best admission with lower latency and cost (Table 1). Reported latency is predicted latency, not wall-clock multi-backend serving time (Kwon et al., 2023; Agrawal et al., 2024). At very low concurrency, unfinished work and crowding-adaptive admission effects attenuate; always-wide bands can be faster but worse at ranking.