Beyond Accuracy and Cost: Latency-Aware LLM Query Routing for Dynamic Workloads

Shivam Patel[*1], Akaash R. Parthasarathy[*1], Ankur Mallick[2], and Gauri Joshi[1]
[1]Carnegie Mellon University, [2]Microsoft
[*]Equal contribution.

Abstract. Modern language query routers improve inference efficiency by assigning each query to a model that balances response quality and monetary cost. However, current query routers are largely latency-agnostic and do not consider the generation latency experienced by queries at model instances. In practice, latency is often controlled by load-balancing policies such as round-robin or join-the-shortest-queue, which do not account for model accuracy or inference cost. Incorporating query latency into routing is challenging as it depends not only on the query's prompt length, but also on the current prefill and decode workload at the model instance and the scheduling and batching policy of the serving framework. We design a lightweight latency estimator that simulates autoregressive token batch processing in the serving framework and estimates the time-to-first-token (TTFT) of queries. We incorporate this latency estimator into *a latency-aware router that jointly optimizes latency, accuracy, and cost when assigning queries to model instances*. Our experimental results indicate that this joint optimization yields up to 40% improvement in accuracy–cost utility while maintaining the same latencies as standard load-balancing approaches.

Code: https://github.com/akaashrp/sfs

1 Introduction

Routing user queries across a pool of large language models (LLMs) is a promising solution towards efficient utilization of LLMs. Given a prompt or a query, a language query router selects a model instance to maximize expected response quality while controlling monetary cost (the price per token charged to the user). For instance, easier queries are sent to smaller and cheaper models, while harder queries are sent to larger and more expensive models. A substantial body of work [11, 12, 24, 36, 37] studies query routing to strike a good accuracy–cost tradeoff using calibrated correctness and cost predictors that are trained on query evaluations on models in the pool.

Need for latency-aware query routing. In addition to balancing accuracy and cost, response generation should also meet *latency constraints* based on downstream usage or service-level objectives (SLOs). Most existing query routers [10–12, 24, 36, 37] perform model selection based on accuracy and cost, but are latency-agnostic. Therefore, they may route queries to overloaded model instances, causing long queueing delays and latency constraint violations. Latency optimization via load-balancing of queries across model instances is generally left to the underlying systems layer. System-level latency optimization considers a fixed choice of language model for a query and focuses on assigning queries to replicas of the same language model. Works such as [9, 23, 45] perform load balancing of requests across model deployments on heterogeneous hardware. Other works design scheduling policies that prioritize requests based on latency constraints, predicted output lengths, or remaining decode work at the model instance [15, 20, 39, 41, 43]. These system-level routing methods improve latency and resource utilization, but they operate under a fixed query-to-model assignment and thus are accuracy- and cost-agnostic. In this work, we aim to bridge the gap between accuracy-aware and cost-aware routing, which is latency-agnostic, and system-level latency optimization, which is accuracy- and cost-agnostic. *We design a routing framework based on joint accuracy–cost–latency optimization.*

Challenges in latency estimation. Extending accuracy–cost routing to *latency-aware* routing is not straightforward. While existing query routers [11, 24, 37] provide reliable accuracy and cost estimates, latency-aware routing requires reliable estimates of the latency that will be experienced by an incoming query when assigned to a model instance. Unlike cost (the amount charged to the user, which is often a deterministic function of token counts) or accuracy (which can be learned from historical query–model evaluations), *generation latency depends on the workload of queries that are currently being served at a model instance.* Modern LLM serving frameworks (e.g., vLLM

[26], Sarathi-Serve [2]) execute *many requests concurrently* using continuous batching, resulting in simultaneous processing of sequences in the prompt token KV-cache computation stage (prefill stage), and the sequential decode token generation stage (decode stage). The response generation rate and latencies experienced by queries at a model instance depend on (i) the hardware specifications and language model architecture, (ii) the current prefill versus decode workload composition of queries, and (iii) the batching and scheduling policy of the serving framework.

Our latency-aware routing framework. We propose a routing framework that estimates query latencies and makes routing decisions by incorporating information about the query workload at each model instance, and the scheduling policies used by the underlying serving framework. Our proposed Serving Framework Simulation (SFS) based *time-to-first-token* (TTFT) latency estimator simulates the evolution of token batch composition during autoregressive generation at model instances, and predicts latencies incurred by new queries. Experiments show that our latency-aware routing framework strikes a favorable accuracy–cost–latency tradeoff, and outperforms existing latency-agnostic routers as well as accuracy- and cost-agnostic load balancing methods. The closest prior work to our setting is [27], which considers an output-length-based latency estimator and jointly optimizes response quality and generation latency. However, it does not account for varying prefill and decode compositions, serving framework policies, and the present workload at model instances. We provide a more detailed discussion of prior work in Section A.

Paper organization. The remainder of the paper is organized as follows: Section 2 introduces relevant background on autoregressive generation and serving frameworks, Section 3 formally presents our routing objective and notation, Section 4 proposes our latency estimation approach, Section 5 presents an empirical evaluation of the latency-aware router, and Section 6 concludes our work and discusses future directions.

2 Background on LLM serving frameworks

Autoregressive generation. Most transformer-based language models generate sequences autoregressively, predicting each token from the input prompt and all previously generated tokens through attention [47]. Computationally, generation consists of two phases [40]: the *prefill* phase, where the input prompt tokens are processed in parallel to compute the key-value states and initialize the KV cache; and the *decode* phase, where output tokens are generated sequentially, with new key-value states appended to the cache for future attention computations. The prefill phase is typically compute-bound since prompt tokens can be processed in parallel, whereas the decode phase is typically memory-bound since each step reads the cached key-value states for previous tokens [2, 26]. While other generation paradigms exist, including diffusion-based generation and non-autoregressive refinement [18, 28, 30], our analysis focuses on the popular attention-based autoregressive setting.

Batch composition and generation throughput. Batching of computational workload for generating responses to queries is a crucial aspect of utilizing the parallel computation capacity of GPUs. Naive sequence-level batching collects queries and generates new tokens for each query in the batch until completion. This leads to inefficient hardware utilization as shorter sequences in the batch need to wait for longer sequences to finish generation. To overcome this, modern serving frameworks utilize continuous batching (ORCA [50], vLLM [26], etc.), where new prompts are admitted in the generation stage at token iteration (often simply denoted as "token batch") boundaries, unlike sequence-level batch boundaries in naive batching. This implies that autoregressive generation for a new sequence can begin while other sequences are still being generated. Continuous batching leads to better hardware utilization and higher throughput.

Memory and scheduling mechanisms in serving frameworks. Continuous batching demands careful management of the varied composition of prefill and decode tokens in sequences, making memory allocation crucial to ensure efficient compute utilization. Early solutions, such as ORCA [50], reserve memory for the KV cache up to the maximum possible context length of each sequence being served. This leads to over-provisioning since most sequences do not achieve the maximum context length, resulting in suboptimal hardware utilization and reduced throughput. PagedAttention (used in vLLM) [26] applies virtual memory-style paging to the KV cache, avoiding large contiguous pre-allocation and instead allocating fixed-size KV cache blocks to sequences dynamically as generation proceeds. The heterogeneous compute and memory requirements of the prefill and decode stages allow further refinement of the batching policy. Chunked prefills [2] balance prefill and decode work within a token batch by splitting long prefills into smaller chunks, improving hardware utilization and throughput while avoiding decode stalls. Our latency estimator

factors in these aspects to produce accurate TTFT estimates for queries.

3 Problem formulation

System model and notation. We refer to each deployed LLM, together with its associated hardware allocation and serving framework, as a *model instance*, indexed by $j \in \mathcal{J}$. For each newly arriving query q_i, the router estimates the accuracy, cost, and time-to-first-token (TTFT) latency associated with each candidate model instance, and selects an instance $m(i) \in \mathcal{J}$ according to the desired routing objective. We consider TTFT latency as it captures the responsiveness perceived by users and interactive applications.

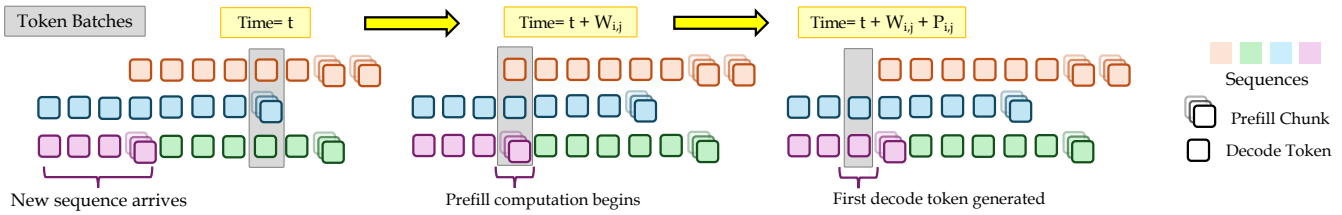

Figure 1: TTFT (time-to-first-token) for autoregressive generation. A new sequence (query) q_i arrives at time t to model instance j (*left*), its prefill computation begins at time $t+W_{i,j}$ (*center*), and the first decode token is generated at time $t+W_{i,j}+P_{i,j}$ (*right*). New sequences are queued until the memory at the model instance is released by sequences already being generated.

Generation latency. The time between a sequence (query) arriving in the system and completing generation (by either generating an `<eos>` token or reaching the model's context length limit) comprises three stages: queueing delay, prefill time, and decode time. Queueing delay, or waiting time, is the idle period before a query's prefill computation begins, during which it waits for compute and memory resources occupied by preceding queries to become available. Prefill time denotes the time required for query processing and populating the KV cache, and decode time denotes the autoregressive generation time. We examine time-to-first-token (TTFT), which denotes the time from query arrival to first output-token generation. Thus for query q_i at model instance j, TTFT can be represented as

$$L_{i,j}^{\text{ttft}} \;=\; W_{i,j} \;+\; P_{i,j} \tag{1}$$

where $W_{i,j}$ represents the waiting time spent in the queue and $P_{i,j}$ is the prefill and first decode token computation time. We illustrate the latency decomposition in Figure 1 for continuous batching policies. Estimates ($\widehat{L}_{i,j}^{\text{ttft}}$) of TTFT for a given query q_i at each model instance j are required to perform latency-aware routing decisions across model instances.

Accuracy and cost estimators. Using historical evaluations of queries on language models, we train estimators for the response quality and serving cost of previously unseen queries (details presented in Section B.4). For each query q_i and model instance j, we produce an accuracy estimate $\widehat{\text{acc}}_{i,j}$ and a cost estimate $\widehat{\text{cost}}_{i,j}$. Response accuracy refers broadly to a task-dependent quality measure (such as correctness, ROUGE [32], or BARTScore [51]). We utilize a normalized LLM-as-a-judge [29] quality score in $[0, 1]$ for our analysis. We compute generation cost using the per-token pricing of each model. We define the accuracy–cost utility [19, 24, 37] as:

$$\widehat{U}_{i,j}(\lambda) \;=\; \widehat{\text{acc}}_{i,j} - \lambda \widehat{\text{cost}}_{i,j}, \tag{2}$$

where $\lambda \geq 0$ governs the tradeoff between response quality and monetary cost.

Routing objectives and evaluation protocols. In practical deployments, queries are subject to generation latency requirements based on downstream tasks. We therefore define the primary routing objective as maximizing the accuracy–cost utility $\widehat{U}_{i,j}$, while satisfying a per-query time-to-first-token (TTFT) constraint. Let τ_i denote the TTFT target for query q_i, and define the estimated feasible set of model instances as $\mathcal{J}(i) = \{j \in \mathcal{J} : \widehat{L}_{i,j}^{\text{ttft}}(t) \leq \tau_i\}$. The router selects the feasible model instance with the highest estimated utility (if $\mathcal{J}(i) = \phi$, select the instance with the least estimated latency):

$$m(i) \leftarrow \underset{j \in \mathcal{J}:\, \widehat{L}_{i,j}^{\text{ttft}}(t) \leq \tau_i}{\arg\max} \widehat{U}_{i,j}(\lambda). \tag{3}$$

To evaluate this constrained routing objective, we report the average utility achieved by the routed model instances on queries, assigning zero utility to queries that violate their latency constraints. We refer to this metric as *OnTimeUtility*, capturing the average accuracy–cost utility over latency-constraint satisfying responses (where N denotes the total number of queries evaluated):

$$\text{OnTimeUtility}(\lambda) = \frac{1}{N}\sum_{i=1}^{N} U_{i,m(i)}(\lambda)\,\mathbf{1}\Big\{L^{\text{ttft}}_{i,m(i)} \le \tau_i\Big\}, \tag{4}$$

where $U_{i,m(i)}(\lambda)$ and $L^{\text{ttft}}_{i,m(i)}$ denote the realized utility and TTFT after routing, and $\mathbf{1}\{\cdot\}$ denotes the indicator function. We also characterize the underlying tradeoff between utility and latency by considering a Lagrangian relaxation of the constrained objective in eq. (3):

$$m(i) \;\leftarrow\; \arg\max_{j\in\mathcal{J}} \Big(\widehat{U}_{i,j}(\lambda) - \delta\,\widehat{L}^{\text{ttft}}_{i,j}(t)\Big), \tag{5}$$

where $\delta \ge 0$ controls the relative emphasis placed on latency. By varying δ, we obtain a utility–latency tradeoff curve in terms of $(\mathbb{E}[U_{i,m(i)}(\lambda)], \mathbb{E}[L^{\text{ttft}}_{i,m(i)}])$. This enables a characterization of the router's ability to trade off utility and latency independently of per-query service-level objectives. A better router attains higher utility at the same average latency, or equivalently achieves lower latency while preserving the same accuracy–cost utility.

4 Estimating query latencies across model instances

Accurate estimates of *time-to-first-token* (TTFT) latencies are essential for routing queries across language models for generating responses while satisfying latency constraints. Over-estimating latencies leads to under-utilization of model instances, whereas under-estimating can cause frequent constraint violations. In this section, we introduce our proposed *Serving Framework Simulation* (SFS)-based latency estimator. SFS uses query workload information at model instances to simulate token batch composition according to the serving framework's batching and scheduling policies. We design a token-batch processing time estimator that leverages the resulting token batch compositions until first decode token of the new query to estimate TTFT latencies. Our proposed latency estimator is accurate and computationally efficient, achieving $<5\%$ mean absolute percentage error on TTFT predictions with sub-millisecond test-time overhead. We also present a queueing-theoretic analysis of autoregressive generation to estimate average latencies at model instances in cases where real-time workload information is unavailable to the router.

Setup. Let $\mathcal{Q}_j(t)$ denote the set of requests queued at model instance j at time t, i.e., requests assigned to the instance whose prefill processing has not yet begun. Let $\mathcal{A}_j(t)$ denote the set of active requests currently being served, either in the prefill or decode stage. We define the resident set of requests at model instance j as

$$\mathcal{R}_j(t) \triangleq \mathcal{Q}_j(t) \cup \mathcal{A}_j(t).$$

For each resident request $q \in \mathcal{R}_j(t)$, let $\text{tok}^{\text{pre}}_{q,j}(t)$ and $\text{tok}^{\text{dec}}_{q,j}(t)$ denote the remaining number of prefill tokens to be processed and decode tokens to be generated at time t, respectively. For a new query q_i arriving at time t, the goal of our latency estimator is to compute the TTFT estimate $\widehat{L}^{\text{ttft}}_{i,j}(t)$ for each model instance j.

4.1 Serving Framework Simulation (SFS) for latency estimation

We propose *Serving Framework Simulation* (SFS), a latency estimator that simulates the serving framework's batching and scheduling policies at each candidate model instance using the remaining prefill workload and estimated decode workload of resident sequences. SFS accumulates the predicted processing times of successive token batches until the incoming query q_i produces its first decode token, resulting in the TTFT estimate $\widehat{L}^{\text{ttft}}_{i,j}(t)$. We first motivate this design by presenting a simple throughput-based latency predictor and highlighting its limitations. We then elaborate how SFS incorporates predicted decode lengths together with a calibrated token-batch processing time model.

Throughput-based latency estimator. An elementary approach to TTFT estimation for incoming queries computes the remaining prefill workload at each model instance and utilizes the prefill throughput of the model instance to predict latencies. For each model instance j, we define the prefill throughput θ^{pre}_j, measured in tokens

per second, as the ratio of the number of prefill tokens processed in a token batch containing a prefill chunk to the time required to process that batch. We also compute $\bar{T}_j^{\text{dec}}$, the average decode token batch processing time, which approximates the first decode computation time for q_i. At model instance j, the throughput-based TTFT estimate (eq. (1)) for a query q_i arriving at time t is

$$\widehat{L}_{i,j}^{\text{ttft}}(t) = \underbrace{\frac{\sum_{q \in \mathcal{R}_j(t)} \text{tok}_{q,j}^{\text{pre}}(t)}{\theta_j^{\text{pre}}}}_{\text{Waiting time } \widehat{W}_{i,j}(t)} + \underbrace{\frac{\text{tok}_{q_i,j}^{\text{pre}}(t)}{\theta_j^{\text{pre}}} + \bar{T}_j^{\text{dec}}}_{\text{Prefill and first-token time } \widehat{P}_{i,j}(t)} , \tag{6}$$

where the estimated waiting time $\widehat{W}_{i,j}(t)$ is the sum of the remaining prefill tokens of resident queries at model instance j, divided by the prefill throughput θ_j^{pre}, and $\widehat{P}_{i,j}(t)$ is the estimated prefill computation time plus first decode token generation time for query q_i.

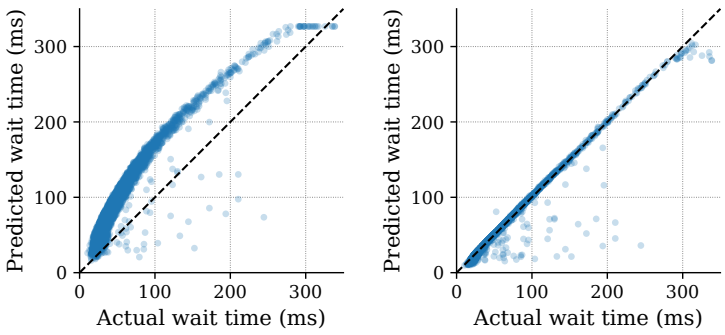

Figure 2: TTFT estimation using the prefill-throughput estimator *(left)* and the proposed Serving Framework Simulation (SFS) estimator *(right)* for Qwen3-0.6B (hosted on one H100 GPU). The prefill throughput estimator does not consider decode interference, while SFS simulates autoregressive generation to calculate latencies.

The prefill-throughput-based estimator approximates the experienced TTFT latencies using the prefill workload at the model instance. This approximation aligns with prefill-prioritizing and prefill-chunk-based serving frameworks, where either prefills are computed eagerly upon query arrivals [26, 50], or chunks of prefill tokens [2] are interleaved with decode computations to improve hardware utilization. However, this estimator does not account for decode-workload interference and its impact on memory and compute utilization at model instances. Consequently, as shown in Figure 2 *(left plot)*, the throughput-based estimator suffers bias when the evaluation-time prefill-decode workload distribution differs from the throughput profiling workload (note that the few outliers in both plots in Figure 2 arise from CPU and kernel overheads).

Serving Framework Simulation (SFS). To improve beyond the throughput-based latency estimator described above, we propose the *Serving Framework Simulation* (SFS) estimator, which jointly considers the prefill and decode workloads at model instances. SFS considers the remaining prefill workload $\{\text{tok}_{q,j}^{\text{pre}}(t)\}_{q \in \mathcal{R}_j(t)}$ for queries in the current resident set $\mathcal{R}_j(t)$ at model instance j, and estimates the remaining decode workload $\{\widehat{\text{tok}}_{q,j}^{\text{dec}}(t)\}_{q \in \mathcal{R}_j(t)}$ by using a decode length estimator (Section B.4). The workload of the resident query set is augmented with prefill token length, and decode length estimates $\left(\text{tok}_{q_i,j}^{\text{pre}}(t), \widehat{\text{tok}}_{q_i,j}^{\text{dec}}(t)\right)$ for the new query q_i. Subsequently, the latency predictor simulates the serving framework's batching and scheduling policies until the first decode token for sequence q_i is generated. Concretely, when estimating TTFT generation latency at model instance j for the arrival of query q_i at time t, SFS forms the augmented workload

$$\mathcal{X}_{i,j}(t) = \left\{ \left(\text{tok}_{q,j}^{\text{pre}}(t), \widehat{\text{tok}}_{q,j}^{\text{dec}}(t)\right) \ : \ q \in \mathcal{R}_j(t) \right\} \cup \left\{ \left(\text{tok}_{q_i,j}^{\text{pre}}(t), \widehat{\text{tok}}_{q_i,j}^{\text{dec}}(t)\right) \right\}.$$

Using $\mathcal{X}_{i,j}(t)$, the estimator then deterministically simulates the serving framework, resulting in consecutive token batches for autoregressive generation. Here, a *token batch* refers to the set of token computations to be performed, consisting of at most one decode token per active decode sequence and possibly a prefill chunk for one or more sequences [2].

Let $\mathcal{B}_j^{(\ell)}(t)$ denote the ℓ^{th} token batch from time t at model instance j. For each sequence q being processed in $\mathcal{B}_j^{(\ell)}(t)$, let $\text{tok}_{q,j}^{\text{pre}}(t,\ell)$ be the number of prefill tokens processed, and let $\text{tok}_{q,j}^{\text{dec}}(t,\ell)$ be the number of decode tokens generated. In the standard autoregressive generation setting, each sequence produces at most one decode token per batch, so $\text{tok}_{q,j}^{\text{dec}}(t,\ell) \in \{0,1\}$. The decode stage for a sequence begins only after all prefill tokens have been processed. Thus, we define the first decode token generation batch of query q_i as

$$\ell_j^{\text{TTFT}}(t,i) \triangleq \min \left\{ \ell : \text{tok}_{q_i,j}^{\text{dec}}(t,\ell) = 1 \right\}. \tag{7}$$

Figure 1 illustrates the model instance state and token-batch composition when a new sequence q_i arrives (*left*), when its prefill computation begins (*center*), and when its first decode token is generated (*right*), marking the TTFT for q_i. SFS estimates TTFT by summing the predicted processing times of the simulated token batches up to and including the first decode token generation for q_i:

$$\widehat{L}^{\mathrm{SFS}}_{i,j}(t) \triangleq \sum_{\ell=1}^{\ell_j^{\mathrm{TTFT}}(t,i)} \widehat{T}^{(\ell)}_j(t), \tag{8}$$

where $\widehat{T}^{(\ell)}_j(t)$ (defined next) denotes the time required to process the token batch $\mathcal{B}^{(\ell)}_j(t)$. Unlike the prefill-throughput estimate in eq. (6), SFS does not separately approximate waiting, prefill, and first decode token computation using model instance throughputs, instead capturing these components directly via the simulated token batches. Figure 2 demonstrates that SFS generates better estimates of TTFT latency compared to the prefill throughput-based estimator (5% vs 85% mean absolute percentage error) by jointly considering prefill and decode workloads and modeling autoregressive generation at model instances.

Token-batch processing time estimator $\widehat{T}^{(\ell)}_j(t)$. Processing a token batch at a model instance primarily involves MLP-layer computation, attention computation, and KV-cache memory reads. Modeling each low-level kernel and memory transfer separately is fragile and difficult to generalize across serving workloads. We therefore design a higher-level estimator based on token batch composition. For each sequence $q \in \mathcal{B}^{(\ell)}_j(t)$, let $c_{q,j}(t,\ell)$ denote the context length of sequence q before token batch ℓ is processed. We estimate the processing time of token batch $\mathcal{B}^{(\ell)}_j(t)$ as

$$\begin{aligned}\widehat{T}^{(\ell)}_j(t) = \beta_{0,j} &+ \beta_{1,j} \sum_{q\in\mathcal{B}^{(\ell)}_j(t)} \left(\mathrm{tok}^{\mathrm{pre}}_{q,j}(t,\ell) + \mathrm{tok}^{\mathrm{dec}}_{q,j}(t,\ell)\right) + \beta_{2,j} \sum_{q\in\mathcal{B}^{(\ell)}_j(t)} c_{q,j}(t,\ell)\,\mathrm{tok}^{\mathrm{dec}}_{q,j}(t,\ell) \\ &+ \beta_{3,j} \sum_{q\in\mathcal{B}^{(\ell)}_j(t)} \left(\mathrm{tok}^{\mathrm{pre}}_{q,j}(t,\ell) c_{q,j}(t,\ell) + \frac{\mathrm{tok}^{\mathrm{pre}}_{q,j}(t,\ell)\left(\mathrm{tok}^{\mathrm{pre}}_{q,j}(t,\ell)+1\right)}{2}\right).\end{aligned} \tag{9}$$

The intercept term ($\beta_{0,j}$) captures fixed per-batch overheads. The first workload term (with coeff. $\beta_{1,j}$) captures token-wise dense-layer computation, which scales linearly with the number of tokens in a batch. The second term (with coeff. $\beta_{2,j}$) accounts for the attention computation and KV-cache reads for each decode sequence, which scales linearly with increasing context. The third term (with coeff. $\beta_{3,j}$) models prefill attention cost: each prefill token attends both to the existing context and to the preceding tokens within the same prefill chunk, yielding a contribution that grows with the product of the prefill chunk size and prior context length, plus the quadratic attention cost within the chunk.

The coefficients $(\beta_{0,j}, \beta_{1,j}, \beta_{2,j}, \beta_{3,j})$ are calibrated for each model instance using observed token-batch processing times. Figure 3 shows the estimated token-batch processing time from eq. (9) against the observed batch processing time, demonstrating high accuracy with $\approx$4% mean absolute percentage error.

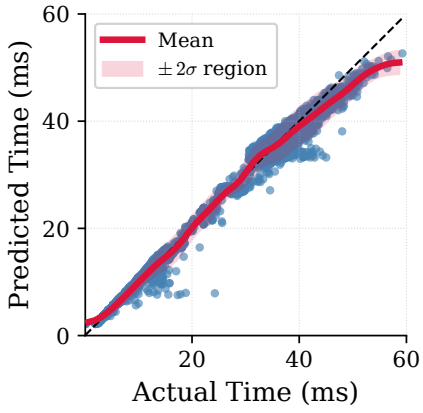

Figure 3: Token-batch processing time estimator. The estimator in eq. (9) closely matches observed batch times, with $\approx$4% mean absolute percentage error across token-batch compositions (for Qwen3-0.6B hosted on one H100 GPU).

SFS is also robust to moderate errors in predicted remaining decode lengths. The TTFT estimation only requires simulating the autoregressive serving process up to the token batch in which the new query generates its first decode token, rather than until all resident sequences finish generation. Since each active sequence contributes at most one decode token per token batch, the number of active decode sequences is more important for TTFT estimation than the exact remaining decode length of every resident sequence.

4.2 Average-case TTFT estimation without real-time workload information

In many practical deployments, for example, when an enterprise is routing queries to third-party cloud servers, the router may not have access to real-time query workload information, such as the number of resident queries at each model instance and their prefill and decode token composition. In this scenario, routing decisions need to rely on time-average quantities such as query arrival rates and model instance service capacities instead of real-time workload knowledge at model instances. We therefore construct an average-case estimator based on a queueing-theoretic abstraction of model instances and autoregressive generation.

Modern serving frameworks leverage hardware parallelism via continuous batching [50], enabling multiple sequences to be served concurrently (refer Section 2). However, this concurrency is limited by memory and hardware constraints and is often controlled as a hyperparameter [26]. We model this behavior using the Limited Processor Sharing (LPS) queueing system model [52], where up to k queries can be served in parallel by the server, and the rest are placed in a first-come-first-serve (FCFS) queue behind the k queries until one of them is served. Because compute capacity is shared equally among active queries, lighter workloads yield faster per-query processing. This abstraction is consistent with autoregressive generation on GPUs, where fewer concurrent queries generally result in faster token generation [2].

We estimate the service capacity μ_j (in queries per second) of model instance j from its peak generation output throughput. Consider the average arrival rate α_j queries per second, with $\rho_j = \alpha_j/\mu_j < 1$. Under Poisson arrivals and assuming memorylessness property of workload distribution across sequences, the expected waiting time under the LPS scheme is

$$\widehat{W}^{\text{avg}}_{i,j} = \mathbb{E}[W_{i,j}] = \frac{(\alpha_j/\mu_j)^k}{\mu_j - \alpha_j}. \tag{10}$$

The corresponding average-case TTFT estimator includes the query's prefill computation time and the average first decode token generation time:

$$\widehat{L}^{\text{avg}}_{i,j} = \widehat{W}^{\text{avg}}_{i,j} + \frac{\text{tok}^{\text{pre}}_{q_i,j}}{\theta^{\text{pre}}_j} + \bar{T}^{\text{dec}}_j. \tag{11}$$

We defer the derivation to Section E. Figure 4 shows that this abstraction captures the qualitative growth of waiting time with load and that the observed system behavior is consistent with a finite-parallelism serving model (the observed time average parallelism for the setup in Figure 4 is $k \approx 25$).

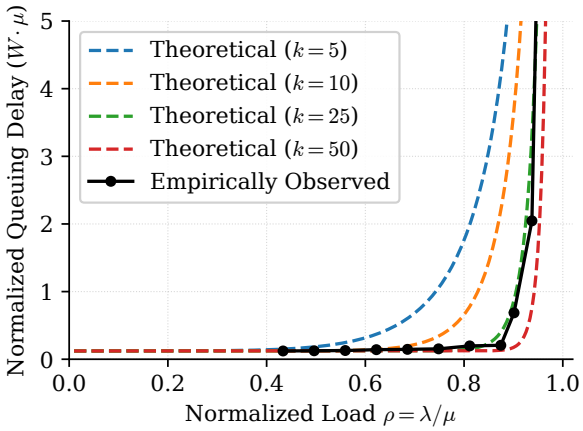

Figure 4: Observed TTFT for sequence generation on Qwen3-0.6B (on single H100 GPU) compared with theoretical TTFT under the Limited Processor Sharing (LPS) model where k queries can be served in parallel.

5 Experiments

Experimental setup. In our analysis, we sample queries from four tasks: **Alpaca** [46], **HotpotQA** [49], **GovReport-Summarization** [21], and **WritingPrompts** [13]. The four tasks represent varying (input, output) token length distributions for queries, with (short, short), (long, short), (long, long), and (short, long) types, respectively. We sample 2.5K queries from each task to estimate per-instance prefill throughput, calibrate the token-batch processing time estimators, and train output length and response quality predictors. We simulate query arrivals based on a Poisson process (Figures 5 and 6), and also generalize to bursty arrivals based on a Markov-Modulated Poisson process (Figure 7).

We consider three language models: **Qwen3-0.6B**, **Qwen3-8B**, and **Qwen3-32B** from the Qwen3 family [42], covering a range of sizes and capabilities. We deploy Qwen3-0.6B and Qwen3-8B on one H100 GPU [35] each, and Qwen3-32B on two H100 GPUs (using tensor parallelism), characterizing three model instances. We use the vLLM [26] serving framework for our experiments; however, our presented latency estimators can be easily extended to other serving frameworks as well. We evaluate the responses generated by models on queries via LLM-as-a-judge [29] using Gemini 3.1 Pro Preview [16] (evaluation prompt provided in Section D). We discuss further experimental details in Section B.

Baselines. We compare the proposed latency-aware router, using the *Serving Framework Simulation (SFS)* based TTFT estimator against three baselines. *Round Robin* policy assigns queries cyclically across model instances, independent of workload, utility, or latency. The *Shortest Queue* policy routes each incoming query to the instance with the fewest resident requests, explicitly targeting low workload to minimize generation latencies. Finally, the *Latency-Agnostic* policy routes solely based on maximum utility, ignoring generation latencies.

Overview of results. We examine latency-aware routing from two complementary perspectives. First, we vary the offered query load under the latency-constrained routing objective in eq. (3). Second, we vary the latency penalty parameter δ under the Lagrangian relaxation of latency-constrained routing (eq. (5)). For both these scenarios, we keep the cost coefficient λ fixed (at $5{\times}10^{-4}$) and defer sweeps over λ to Section G. We further present the performance of our approach across bursty query arrivals and comment on the overhead introduced by the serving framework simulation-based latency estimation.

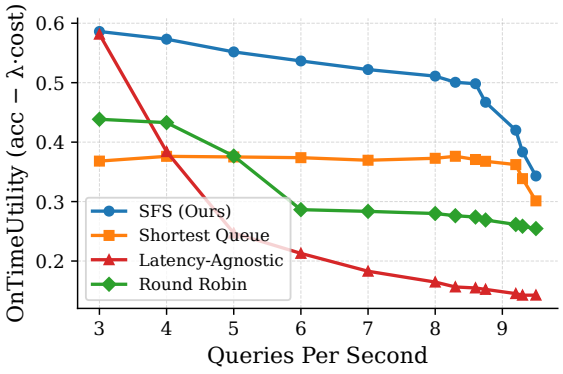

Figure 5: OnTimeUtility across varying arrival rate (in qps). Latency-aware routing (SFS) balances utility and latency to maintain high OnTimeUtility, demonstrating 33% higher AUC.

Varying offered load. As the arrival rate of queries increases, the model instances become saturated and the feasible set of instances that satisfy queries' latency constraints shrinks. In such scenarios, a better router will yield a higher *OnTimeUtility* value (eq. (4)) for latency-constrained routing (eq. (3)) across a wide range of query workloads. Figure 5 shows that our proposed Serving Framework Simulation (SFS) for latency estimation achieves the highest *OnTimeUtility* across all query workloads, yielding a **33% gain** in terms of area-under-the-curve (AUC) over the best baseline, and **46% improvement** in OnTimeUtility (at 5qps). The *Shortest Queue* policy achieves low latencies, but it is agnostic to the query's model preferences. The *Round Robin* policy is oblivious to both model instance saturation and query preferences. *Latency-Agnostic* routing prioritizes query preferences and yields high utility when desired model instances are available, but its performance degrades with increasing workloads.

Varying latency emphasis. To analyze the trade-off between utility and latency, we vary δ values in eq. (5) and obtain average utility and latency values for queries (following the Poisson arrival process with rate $=8$ qps). Figure 6 shows that our proposed approach consistently improves the utility–latency performance over the baselines, resulting in **40% higher utility** than *Shortest Queue* policy, and importantly, provides a controllable tradeoff which can be set by choosing the corresponding δ value. Setting $\delta = 0$ recovers the *Latency-Agnostic* routing policy. The *Round Robin* and *Shortest Queue* baselines balance model instance workload in some regimes but do not exploit the heterogeneous accuracy–cost structure of the model instance pool and therefore attain substantially worse realized utility than our method for comparable latency values.

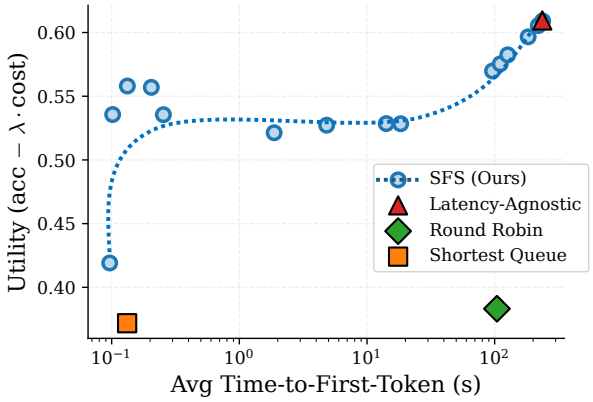

Figure 6: Utilities plotted against TTFT latencies by varying δ in eq. (5). Latency-aware routing provides a controllable utility–latency tradeoff.

Evaluation across arrival processes. In practical deployments, query arrivals are often correlated and bursty, which may not be captured well by the standard Poisson arrival process. We extend our analysis to query arrivals based on a two-state Markov Modulated Poisson Process (MMPP-2) [14] (Section C), where the low-state denotes standard query arrivals (rate λ_L) and a high-state indicates a duration of heavy workload (with arrival rate λ_H). The ratio of the high- and low-state arrival rates indicates the level of burstiness in query arrivals. Figure 7 illustrates consistently higher performance of our approach across arrival processes and average arrival rates.

Estimator overhead. Since estimating latencies for queries at model instances lies on the critical path of routing, latency estimators should not induce high computational overhead. Our proposed SFS latency estimator simulates batching policies from the current workload state (including the newly arrived query) and halts at the token batch

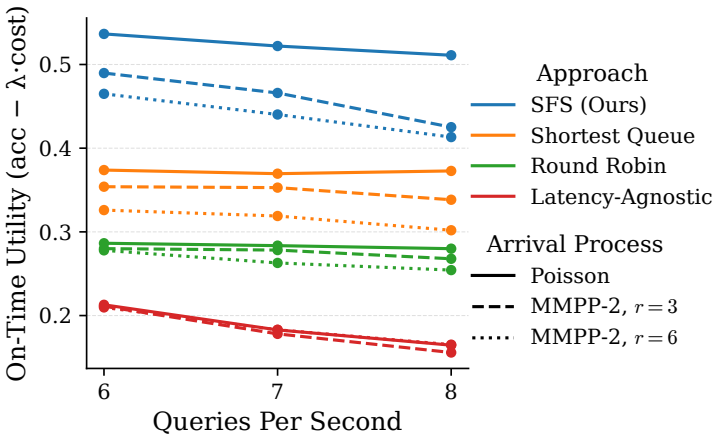

Figure 7: OnTimeUtility for routers under query arrivals from a Markov-modulated Poisson process with high-to-low arrival-rate ratio $r = 3, 6$. Our proposed approach improves OnTimeUtility across query arrival processes.

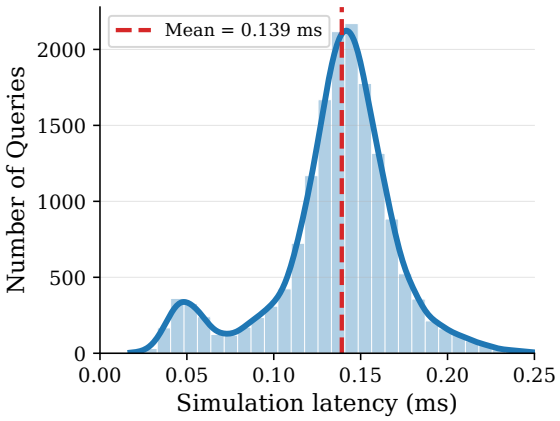

Figure 8: Wall-clock overhead of the SFS (ours) latency estimator. Mean simulation latency of $\approx 10^{-4}$s indicates negligible routing-time overhead for latency estimation (Section 4.1).

corresponding to first token generation for this new sequence. This avoids full simulation till completion of all resident requests at the model instance. Figure 8 depicts an average latency estimation **time overhead of $\approx 10^{-4}$s**, which is negligible relative to query generation TTFTs (usually in seconds [26]). Further implementation details of the SFS estimator are provided in Section F.

6 Concluding remarks

Most existing language query routers optimize the accuracy–cost trade-off, but do not account for latencies experienced by queries. We proposed a latency-aware routing framework that jointly optimizes accuracy, cost, and time-to-first-token (TTFT) latency. The core component of our approach is a lightweight Serving Framework Simulation (SFS) latency estimator that predicts TTFT by simulating prefill and decode token batches under the serving framework's policies. By incorporating these estimates into routing decisions, our approach avoids overloaded instances while preserving strong accuracy–cost performance. Our experiments show that joint accuracy–cost–latency optimization can substantially improve utility (by over **40**%) as compared to standard load-balancing baselines at comparable latencies. Future directions include joint optimization of routing and model placement, and studying the interactions between routers that independently send queries to shared model instances.

Acknowledgments

This work was partially supported by NSF grants CCF 2045694, CNS-2112471, CPS-2111751, CCF-2428569, ONR grant N00014-23-1-2149, an AI2C Seed grant, and a Gemini Academic Program Award. This work used Bridges-2 GPU at the Pittsburgh Supercomputing Center through allocation CIS250429 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by NSF grants #2138259, #2138286, #2138307, #2137603, and #2138296 [6].

References

[1] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav S. Gulavani, Ramachandran Ramjee, and Alexey Tumanov. Vidur: A large-scale simulation framework for llm inference. In *MLSys*, May 2024.

[2] 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 18th USENIX Conference on Operating Systems Design and Implementation*, OSDI'24,

USA, 2024. USENIX Association. ISBN 978-1-939133-40-3.

[3] Alibaba Cloud. Alibaba cloud model studio: Model invocation pricing, 2026. URL https://www.alibabacloud.com/help/en/model-studio/model-pricing. Alibaba Cloud Documentation Center, accessed Apr. 17, 2026.

[4] Baris Askin, Shivam Patel, Anupam Nayak, Andrea Vigano, Jiin Woo, Gauri Joshi, and Carlee Joe-Wong. Federate the router: Learning language model routers with sparse and decentralized evaluations, 2026. URL https://arxiv.org/abs/2601.22318.

[5] Agrim Bari, Parikshit Hegde, and Gustavo de Veciana. Optimal scheduling algorithms for llm inference: Theory and practice. *Proc. ACM Meas. Anal. Comput. Syst.*, 9(3), December 2025. doi: 10.1145/3771574. URL https://doi.org/10.1145/3771574.

[6] Timothy J. Boerner, Stephen Deems, Thomas R. Furlani, Shelley L. Knuth, and John Towns. Access: Advancing innovation: Nsf's advanced cyberinfrastructure coordination ecosystem: Services & support. In *Practice and Experience in Advanced Research Computing 2023: Computing for the Common Good*, PEARC '23, page 173–176, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399852. doi: 10.1145/3569951.3597559. URL https://doi.org/10.1145/3569951.3597559.

[7] Jianhao Chen, Chenxu Wang, Gengrui Zhang, Peng Ye, Lei Bai, Wei Hu, Yuzhong Qu, and Shuyue Hu. Learning compact representations of llm abilities via item response theory, 2025. URL https://arxiv.org/abs/2510.00844.

[8] Lingjiao Chen, Matei Zaharia, and James Zou. Frugalgpt: How to use large language models while reducing cost and improving performance. *Transactions on Machine Learning Research*, 2024.

[9] Wei Da and Evangelia Kalyvianaki. Block: Balancing load in llm serving with context, knowledge and predictive scheduling, 2025. URL https://arxiv.org/abs/2508.03611.

[10] Jasper Dekoninck, Maximilian Baader, and Martin Vechev. A unified approach to routing and cascading for LLMs. In *Forty-second International Conference on Machine Learning*, 2025.

[11] 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 *The Twelfth International Conference on Learning Representations*, 2024.

[12] Dujian Ding, Ankur Mallick, Shaokun Zhang, Chi Wang, Daniel Madrigal, Mirian Del Carmen Hipolito Garcia, Menglin Xia, Laks V. S. Lakshmanan, Qingyun Wu, and Victor Rühle. BEST-route: Adaptive LLM routing with test-time optimal compute. In *Forty-second International Conference on Machine Learning*, 2025.

[13] Angela Fan, Mike Lewis, and Yann Dauphin. Hierarchical neural story generation. In Iryna Gurevych and Yusuke Miyao, editors, *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 889–898, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1082. URL https://aclanthology.org/P18-1082/.

[14] Wolfgang Fischer and Kathleen Meier-Hellstern. The markov-modulated poisson process (mmpp) cookbook. *Performance Evaluation*, 18(2):149–171, 1993. ISSN 0166-5316. doi: https://doi.org/10.1016/0166-5316(93)90035-S. URL https://www.sciencedirect.com/science/article/pii/016653169390035S.

[15] Yichao Fu, Siqi Zhu, Runlong Su, Aurick Qiao, Ion Stoica, and Hao Zhang. Efficient LLM scheduling by learning to rank. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=wlLjYl0Gi6.

[16] Gemini Team. Gemini: A family of highly capable multimodal models, 2025. URL https://arxiv.org/abs/2312.11805.

[17] Ruihao Gong, Shihao Bai, Siyu Wu, Yunqian Fan, Zaijun Wang, Xiuhong Li, Hailong Yang, and Xianglong Liu. Past-future scheduler for llm serving under sla guarantees. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, page 798–813, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400710797. doi: 10.1145/3676641.3716011. URL https://doi.org/10.1145/3676641.3716011.

[18] Jiatao Gu, James Bradbury, Caiming Xiong, Victor O.K. Li, and Richard Socher. Non-autoregressive neural machine translation. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=B1l8BtlCb.

[19] 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. In *Agentic Markets Workshop at ICML 2024*, 2024. URL https://openreview.net/forum?id=IVXmV8Uxwh.

[20] Jinqi Huang, Yi Xiong, Xuebing Yu, Wenjie Huang, Entong Li, Li Zeng, and Xin Chen. Slo-aware scheduling for large language model inferences, 2025. URL https://arxiv.org/abs/2504.14966.

[21] Luyang Huang, Shuyang Cao, Nikolaus Parulian, Heng Ji, and Lu Wang. Efficient attentions for long document summarization. In Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tur, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou, editors, *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1419–1436, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.112. URL https://aclanthology.org/2021.naacl-main.112/.

[22] Saki Imai, Rina Nakazawa, Marcelo Amaral, Sunyanan Choochotkaew, and Tatsuhiro Chiba. Predicting llm inference latency: A roofline-driven ml method. In *ML for Systems Workshop, Annual Conference on Neural Information Processing Systems*, 2024.

[23] Kunal Jain, Anjaly Parayil, Ankur Mallick, Esha Choukse, Xiaoting Qin, Jue Zhang, Íñigo Goiri, Rujia Wang, Chetan Bansal, Victor Rühle, Anoop Kulkarni, Steve Kofsky, and Saravan Rajmohan. Performance aware llm load balancer for mixed workloads. In *Proceedings of the 5th Workshop on Machine Learning and Systems*, EuroMLSys '25, page 19–30, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400715389. doi: 10.1145/3721146.3721947. URL https://doi.org/10.1145/3721146.3721947.

[24] Wittawat Jitkrittum, Harikrishna Narasimhan, Ankit Singh Rawat, Jeevesh Juneja, Congchao Wang, Zifeng Wang, Alec Go, Chen-Yu Lee, Pradeep Shenoy, Rina Panigrahy, Aditya Krishna Menon, and Sanjiv Kumar. Universal model routing for efficient LLM inference. In *The Fourteenth International Conference on Learning Representations*, 2026.

[25] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf.

[26] 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 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

[27] Javid Lakha, Minlan Yu, and Rana Shahout. Faster, cheaper, just as good: Cost- and latency constrained routing for llms. In *Sparsity in LLMs (SLLM): Deep Dive into Mixture of Experts, Quantization, Hardware, and Inference*, 2025. URL https://openreview.net/forum?id=pZFJLsIY2m.

[28] Jason Lee, Elman Mansimov, and Kyunghyun Cho. Deterministic non-autoregressive neural sequence modeling by iterative refinement. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii, editors, *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1173–1182, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1149. URL https://aclanthology.org/D18-1149/.

[29] Dawei Li, Bohan Jiang, Liangjie Huang, Alimohammad Beigi, Chengshuai Zhao, Zhen Tan, Amrita Bhattacharjee, Yuxuan Jiang, Canyu Chen, Tianhao Wu, Kai Shu, Lu Cheng, and Huan Liu. From generation to judgment: Opportunities and challenges of LLM-as-a-judge. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng, editors, *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 2757–2791, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.138. URL https://aclanthology.org/2025.emnlp-main.138/.

[30] Xiang Lisa Li, John Thickstun, Ishaan Gulrajani, Percy Liang, and Tatsunori Hashimoto. Diffusion-LM improves controllable text generation. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=3s9IrEsjLyk.

[31] Yueying Li, Jim Dai, and Tianyi Peng. Throughput-optimal scheduling algorithms for llm inference and ai agents, 2025. URL https://arxiv.org/abs/2504.07347.

[32] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics. URL https://aclanthology.org/W04-1013/.

[33] Michael Mitzenmacher and Rana Shahout. Queueing, predictions, and llms: Challenges and open problems, 2025. URL https://arxiv.org/abs/2503.07545.

[34] Moonmoon Mohanty, Gautham Bolar, Preetam Patil, Umamaheswari Devi, Felix George, Pratibha Moogi, and Parimal Parag. Deferred prefill for throughput maximization in llm inference. In *Proceedings of the 5th Workshop on Machine Learning and Systems*, EuroMLSys '25, page 100–106, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400715389. doi: 10.1145/3721146.3721962. URL https://doi.org/10.1145/3721146.3721962.

[35] NVIDIA Corporation. Nvidia h100 pcie gpu — product brief. Technical Report PB-11133-001, NVIDIA, September 2022. URL https://www.nvidia.com/content/dam/en-zz/Solutions/gtcs22/data-center/h100/PB-11133-001_v01.pdf.

[36] 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 from preference data. In *The Thirteenth International Conference on Learning Representations*, 2025.

[37] Shivam Patel, Neharika Jali, Ankur Mallick, and Gauri Joshi. Proxrouter: Proximity-weighted llm query routing for improved robustness to outliers, 2025. URL https://arxiv.org/abs/2510.09852.

[38] Shivam Patel, William Cocke, and Gauri Joshi. Locus: Low-dimensional model embeddings for efficient model exploration, comparison, and selection, 2026. URL https://arxiv.org/abs/2601.21082.

[39] Archit Patke, Dhemath Reddy, Saurabh Jha, Haoran Qiu, Christian Pinto, Chandra Narayanaswami, Zbigniew Kalbarczyk, and Ravishankar Iyer. Queue management for slo-oriented large language model serving. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, SoCC '24, page 18–35, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712869. doi: 10.1145/3698038.3698523. URL https://doi.org/10.1145/3698038.3698523.

[40] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. In D. Song, M. Carbin, and T. Chen, editors, *Proceedings of Machine Learning and Systems*, volume 5, pages 606–624. Curan, 2023. URL https://proceedings.mlsys.org/paper_files/paper/2023/file/c4be71ab8d24cdfb45e3d06dbfca2780-Paper-mlsys2023.pdf.

[41] 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, pages 1–7, San Diego, CA, USA, 2024. Association for Computing Machinery.

[42] Qwen. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

[43] 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. URL https://openreview.net/forum?id=7JhGdZvW4T.

[44] Wei Song, Zhenya Huang, Cheng Cheng, Weibo Gao, Bihan Xu, GuanHao Zhao, Fei Wang, and Runze Wu. IRT-router: Effective and interpretable multi-LLM routing via item response theory. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15629–15644, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.761. URL https://aclanthology.org/2025.acl-long.761/.

[45] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: dynamic scheduling for large language model serving. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, OSDI'24, USA, 2024. USENIX Association. ISBN 978-1-939133-40-3.

[46] 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.

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[48] Ronald W. Wolff. Poisson arrivals see time averages. *Operations Research*, 30(2):223–231, 1982. ISSN 0030364X, 15265463. URL http://www.jstor.org/stable/170165.

[49] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

[50] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/yu.

[51] Weizhe Yuan, Graham Neubig, and Pengfei Liu. BARTScore: Evaluating generated text as text generation. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021. URL https://openreview.net/forum?id=5Ya8PbvpZ9.

[52] Jiheng Zhang. *Limited Processor Sharing Queues and Multi-Server Queues*. PhD thesis, Georgia Institute of Technology, August 2009. URL https://people.orie.cornell.edu/jdai/thesis/Jiheng_Zhang_Thesis.pdf.

[53] Richard Zhuang, Tianhao Wu, Zhaojin Wen, Andrew Li, Jiantao Jiao, and Kannan Ramchandran. EmbedLLM: Learning compact representations of large language models. In *The Thirteenth International Conference on Learning Representations*, 2025.

A Related Works

A broad range of research directions tackle efficient response generation from varying perspectives. Language Query Routing considers utilizing a pool of language models for extracting collective capabilities to balance performance and cost for query response generation. The most suitable model for generating a response to a query is chosen based on a desired tradeoff, through a router that maps queries to suitable models. Multiple response aggregation, dynamic test-time resource allocation, and online feedback-driven model routing form promising extensions of the model selection setting. Additionally, systems-driven works emphasize latency-constrained response generation through scheduling, workload-aware load balancing, autoscaling, and queue management. Crucially, these directions do not provide a structured understanding of balancing response accuracy-cost performance along with generation latency, which is of practical interest in a wide range of scenarios. In our work, we explicitly bridge this gap and present practical latency estimators that are lightweight, accurate, and interpretable, and that integrate latency into routing decisions. Below, we elaborate on related works across concurrent directions of research.

LLM routing for quality–cost trade-offs. Early work on model selection uses historical model behavior on queries to estimate task-specific model abilities, allowing new queries to be routed to suitable models based on a desired quality–cost trade-off. FrugalGPT [8] introduced rule-based selection and cascading across language models, demonstrating quality gains. HybridLLM [11] proposed routing between a large and a small language model, using thresholds on model-correctness estimates to maintain response quality while reducing cost. RouteLLM [36] introduced human-preference-based routing over generated responses to choose the preferred model for response generation. Simple nonparametric routing approaches, such as clustering and nearest neighbors, were formally examined by [19, 24], and extended in [37] to generalize across query distributions. Additional approaches include interpretable routing mechanisms [7, 44], multi-response aggregation [12], cascading language models [10], and methods that learn compact model representations through embeddings [38, 53]. Practical aspects such as sample efficiency and distributed training have also been studied [4].

Dynamic scheduling and queue management. Response generation for queries with heterogeneous latency constraints requires moving beyond the usual first-come-first-served (FCFS) scheduling policy toward priority-based scheduling that can meet constraints and improve throughput [20]. Model switching and query-group scheduling across instances [39] allow resource orchestration to improve generation latency by solving a linear program that minimizes latency violations. Llumnix [45] explores runtime query scheduling for improved load balancing and isolation of response generation across model instances, mitigating resource fragmentation and improving throughput. Other works examine scheduling algorithms from a queueing-theoretic perspective and highlight open problems [5, 31, 33].

Workload-aware routing and latency modeling. Load balancing and workload routing across model instances of the same LLM can reduce experienced latency and improve hardware utilization. [23] propose a reinforcement-learning-based router for workload- and query-distribution-aware scheduling. For optimizing deployment hyperparameters for model serving, Vidur [1] estimates operator-level computation times and memory latencies, modeling GPU generation through a CPU simulator. Analytical roofline estimators [22] also provide lightweight operator-level latency models for LLM generation. Thus, latency modeling can be studied at varying levels of hardware abstraction; in our work, we propose lightweight estimators across different workload-observability scenarios. [27] examine a latency-aware routing objective with an output-length predictor and subsequent cost and latency estimator, demonstrating improved response quality while respecting latency constraints.

Output-length prediction and size-aware scheduling. Since autoregressive generation latency depends on the number of generated response tokens, predicted response lengths can provide valuable signals for designing non-FCFS scheduling policies. [41] propose speculative shortest job first, which uses a lightweight response-length predictor to estimate output sequence lengths. Learning to rank relative output lengths of queries in a batch provides a similar shortest-job-first scheduler when exact output-length predictions are unavailable [15]. [43] use internal LLM embeddings to predict the remaining number of decode tokens, eliminating the need for precomputed response lengths, which may be inaccurate and lead to suboptimal scheduling. [17] predict output-length distributions and future memory requirements based on historical query-length observations to improve latency-oriented serving. [9] present a distributed scheduling framework for load balancing and auto-provisioning across model instances.

B Query and Language Model Details

B.1 Model inference costs and average performance

Table 1 reports the tokenwise input and output costs for the language models used in our analysis. Table 2 presents the average performance of models across the tasks under consideration, evaluated using the LLM-as-a-judge scheme [29].

Table 1: Pricing for Qwen3 models (non-thinking mode) in USD per Million tokens on Alibaba Cloud Model Studio [3].

Model	Prompt	Response
Qwen3-0.6B	0.044	0.173
Qwen3-8B	0.072	0.287
Qwen3-32B	0.287	0.640

Table 2: Per-query task average scores as percentage (using LLM-as-judge [29]) for Qwen3-0.6B/8B/32B models [42] across Alpaca [46], Govreport-Summarization [21], HotpotQA [49], WritingPrompts [13] tasks.

Query Task	Qwen3-0.6B	Qwen3-8B	Qwen3-32B
Alpaca	53.49%	82.67%	88.85%
GovReport-Summarization	29.02%	86.62%	95.93%
HotpotQA (distractor)	40.98%	88.28%	92.69%
WritingPrompts	17.73%	57.47%	80.69%
Aggregate	35.31%	78.76%	89.54%

B.2 TTFT latency assignment for queries

For evaluations under the hard-constrained routing setting (eq. (3)), each request i is assigned a TTFT target τ_i. We choose realistic and practically achievable TTFT constraints based on our experimental setup, including language models and hardware. The chosen constraints are a simple linear function of the prompt length p_i and include a small amount of random variation. Specifically, we sample $u_i \sim \text{Unif}(1\text{ms}, 5\text{ms})$ and $v_i \sim \text{Unif}(0.98, 1.02)$, and set

$$\tau_i = \min\Big\{1120\text{ms}, \max\big\{150\text{ms}, u_i + \big(155 + 3.5 \times 10^{-3} p_i\big) v_i\big\}\Big\}.$$

Under this parameterization, the expected unclipped target is approximately

$$\mathbb{E}[\tau_i] \approx 158 + 3.5 \times 10^{-3} p_i \text{ ms}.$$

All TTFT targets are generated using a seeded pseudorandom number generator for reproducibility.

B.3 Prompt (prefill) and Response (decode) Length Distributions

We present the query token length distributions across the tasks in Figure 9, and the model response length distributions in Figure 10. The examined tasks and their responses present a wide range of workloads for a comprehensive evaluation of the presented latency estimators and routing scheme.

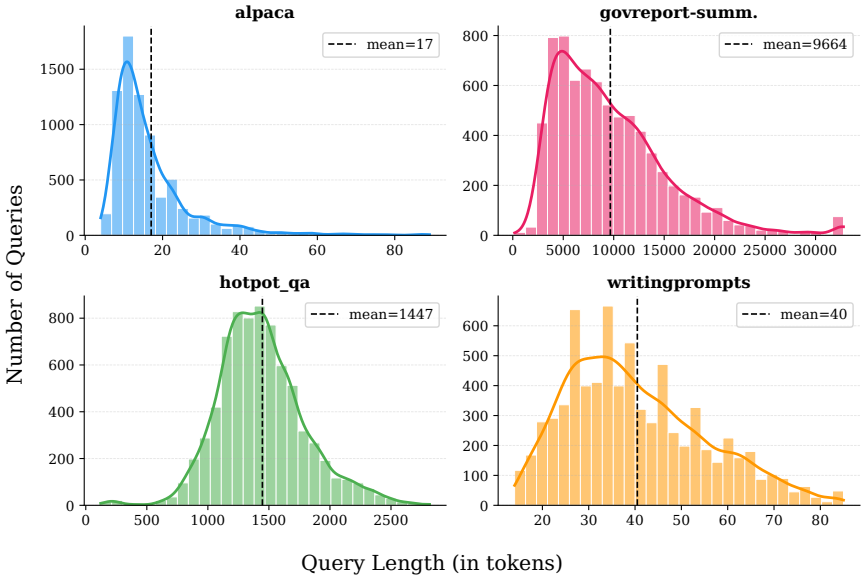

Figure 9: Query token length distributions across the tasks in our evaluations. Alpaca [46], GovReport-Summarization [21], HotpotQA [49], and WritingPrompts [13] tasks present a diverse range of input lengths (~ 10 to 10^4 tokens), simulating real workloads and a comprehensive evaluation of our routing scheme and latency estimator.

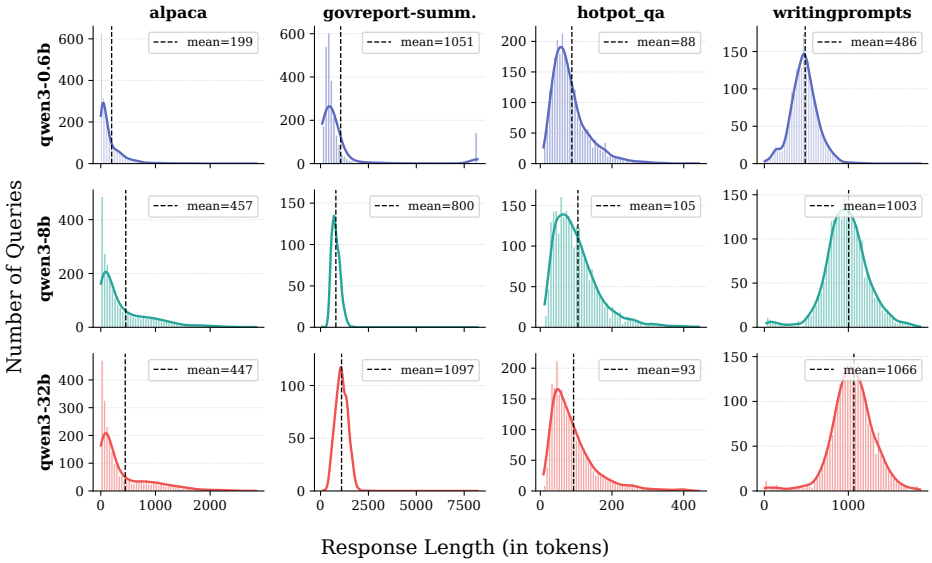

Figure 10: Response token lengths for examined Qwen3-0.6B/8B/32B language models [42] across tasks. Response lengths vary based on the nature of the task (creative generation, summarization, etc.), covering a broad range of $\sim 10^2$ to 10^3 tokens.

B.4 Accuracy, Cost, and Output Length Predictors

We utilize lightweight predictors for estimating model accuracy and output length for queries on arrival. We use LightGBM [25] regressors over features that can be computed directly from the prompt, without requiring additional sentence embedding models, to capture query semantics in a fixed-dimensional space. We rely on prompt tokens, and map them into a 512-dimensional signed hashing vector, normalize by token count, and project the vector to 16 dimensions using PCA down-projection calibrated on prompts from the initial modeling phase. These PCA components are concatenated with prompt-derived features such as prompt token count, character count, sentence count, average characters per token, length-constraint indicators, and task-type indicators (such as list, summarize, or explain).

The feature vector for the accuracy estimator contains the prompt features, a categorical model-id feature, and model characteristics such as logarithm of number of model parameters and maximum context length. We train a LightGBM [25] Huber regressor on quality scores generated by an LLM judge as described in Section D. The output-length predictor estimates the expected number of tokens in the generated response $\widehat{\text{tok}}^{\text{dec}}_{i,j}$ using the same prompt features and categorical model-id feature as the accuracy predictor. We train a LightGBM mean head directly on completion-token counts, and the runtime predictor uses this mean estimate for routing and cost prediction.

Given per-model prompt-token and output-token prices $\text{cost}^{\text{in}}_j$ and $\text{cost}^{\text{out}}_j$ in Table 1, the predicted request cost is derived from the output-length prediction:

$$\widehat{\text{cost}}_{i,j} = \text{tok}^{\text{pre}}_{q_i,j} \cdot \text{cost}^{\text{in}}_j + \widehat{\text{tok}}^{\text{dec}}_{q_i,j} \cdot \text{cost}^{\text{out}}_j,$$

where $\text{tok}^{\text{pre}}_{q_i,j}$ denotes the number of prefill tokens based on tokenization scheme at model instance j. These estimates are used for calculating utility in eq. (2).

C Modeling Correlated Query Arrivals

We evaluate robustness to correlated/bursty query arrivals using a two-state Markov-modulated Poisson process (MMPP-2). In an MMPP-2, arrivals are Poisson conditioned on a latent continuous-time Markov state $Z(t) \in \{L, H\}$, where L is a low-rate state and H is a high-rate burst state [14]. This model preserves a fixed long-run mean arrival rate while introducing temporally correlated periods of low and high load, making it suitable for bursty, time-varying request traffic. For each average arrival rate $\bar{\alpha} \in \{6, 7, 8\}$ qps, we parameterize the MMPP-2 by the rate ratio $r = \alpha_H/\alpha_L$, the high-state fraction $p_H = 0.2$, and correlation time $\tau = 2$s. Thus,

$$\alpha_L = \frac{\bar{\alpha}}{(1 - p_H) + p_H r}, \qquad \alpha_H = r\alpha_L.$$

The Markov transition rates are $z_{L\to H} = 0.1$ and $z_{H\to L} = 0.4$ per second, so the process spends 80% of time in the low-rate state and 20% in the high-rate state. We use homogeneous Poisson arrivals as a low-burstiness baseline, MMPP-2 with $r = 3$ as moderate burstiness, and MMPP-2 with $r = 6$ as high burstiness. Figure 7 shows the performance of our proposed latency-aware routing policy utilizing Serving Framework Simulation based latency estimator, against examined baselines.

Table 3: MMPP-2 arrival rates used in the arrival distribution ablation.

Mean QPS $\bar{\alpha}$	Arrival setting	Low-state QPS α_L	High-state QPS α_H
6	$r = 3$	4.29	12.86
7	$r = 3$	5.00	15.00
8	$r = 3$	5.71	17.14
6	$r = 6$	3.00	18.00
7	$r = 6$	3.50	21.00
8	$r = 6$	4.00	24.00

D Response Evaluation using LLM-as-a-judge

We utilize the Gemini 3.1 Pro Preview [16] LLM via the Gemini API to evaluate the responses generated by the models in Section 5. The evaluation prompt is provided below. We obtain raw scores in the 0-10 range and normalize them to $[0, 1]$.

You are a strict evaluator.

Evaluate each candidate response against the REFERENCE RESPONSE for the given PROMPT.

Multiple candidate responses are provided for the same prompt. Score each one independently, but use the other candidates as comparison context to better calibrate quality differences.

Focus on correctness and completeness. Ignore style, tone, and phrasing unless they affect meaning, accuracy, or instruction-following.

Penalize contradictions, hallucinations, unsupported claims, and missing key points.

Use the full 0 to 10 scale and assign scores granularly across the range to capture meaningful differences in quality.

Do not bunch most responses into a narrow band.

Use decimals when useful.

Scoring anchors:

0 = completely wrong, irrelevant, or unusable
2 = mostly wrong with very little useful content
4 = partially correct but with major mistakes or omissions
6 = substantially correct but missing important details or containing minor errors
8 = strong, correct, and mostly complete with only small gaps
10 = fully correct, complete, and faithful to the prompt and reference

Choose any value from 0 to 10, including decimal values, based on where each response falls between these anchors.

Scores should reflect absolute quality while preserving real differences between candidates answering the same prompt.

If two responses are extremely similar in quality, their scores may be close or equal.

If one response is clearly better or worse, reflect that in the scores.

Return ONLY valid JSON.

The JSON must contain exactly one top-level key, `"scores"`.

The `"scores"` object must contain exactly these keys: `"A"`, `"B"`, and `"C"`.

Use this exact structure:

```
{
    "scores":  {
        "A":  number from 0 to 10,
        "B":  number from 0 to 10,
        "C":  number from 0 to 10
    }
}
```

DO NOT return any other text since this is for an evaluation task.

E Limited Processor Sharing Abstraction for Autoregressive Generation

This appendix derives the queueing-delay estimator used in Section 4.2. We use a *limited processor sharing* (LPS) abstraction from queueing theory [52], in which at most k jobs are served simultaneously, and any additional jobs wait in a FIFO buffer. This model provides a close approximation for autoregressive response generation. Modern serving engines employ iteration-level or continuous batching, so multiple sequences can be decoded concurrently; however, the number of concurrent sequences is constrained by KV-cache memory, token budget, and hardware capacity. Moreover, when fewer sequences are active, each sequence typically experiences higher kernel parallelism, resulting in higher per-sequence throughput, whereas under heavier load this throughput is divided across more active sequences [2, 26]. Thus, a single model instance $j \in \mathcal{J}$ can be viewed as a server of total capacity μ_j whose service is shared among up to k active sequences, with any excess arrivals queued until capacity becomes available.

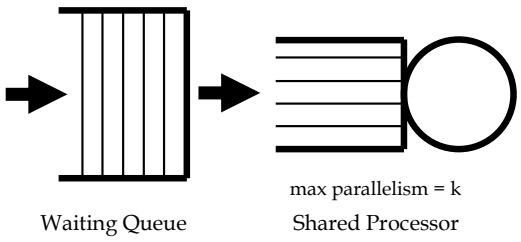

Figure 11: Limited Processor Sharing Scheme [52]. A new job directly begins serving by the processor if fewer than k jobs were already present, otherwise queued (in FCFS manner) until the number of preceding jobs at the server drops below k.

Limited processor sharing model. Consider a model instance $j \in \mathcal{J}$. Queries arrive according to a Poisson process of rate α_j queries/sec. Let $|\mathcal{R}_j(t)|$ denote the number of queries currently present/residing in the system (waiting or actively decoding), and let $|\mathcal{A}_j(t)|$ denote the queries that are being actively decoded. Under LPS with parallelism cap k, the first $|\mathcal{A}_j(t)| = \min\{|\mathcal{R}_j(t)|, k\}$ jobs are in service and share the server equally; any additional jobs wait in a FIFO queue. Each active job experiences an instantaneous service rate

$$\frac{\mu_j}{\min\{|\mathcal{R}_j(t)|, k\}},$$

implying that the *aggregate* service rate remains μ_j whenever the system is nonempty.

Assumptions. The derivation of the waiting time estimate uses the following assumptions.

1. **Poisson arrivals:** query arrivals form a Poisson process of rate α_j queries per second.
2. **Memoryless service requirement:** each query has an exponential service requirement, with mean as $1/\mu_j$ in the time average sense. Alternatively, generation work for queries follows the memoryless distribution [34].
3. **Limited parallelism:** at most k queries can be served simultaneously, and the remaining queries wait in a FIFO queue.
4. **Saturated aggregate capacity:** whenever the active decode batch is nonempty, the instance provides approximately constant aggregate service rate μ, while this rate is shared across active sequences.
5. **Stability condition:** incoming rate of queries does not exceed the service capacity, i.e., the utilization $\rho_j = \alpha_j/\mu_j < 1$.

State process and stationary distribution. Queries arrive according to a Poisson process of rate α_j, so whenever a new query arrives, the state $|\mathcal{R}_j(t)|$ increases by one. For $|\mathcal{R}_j(t)|$ resident queries in the system, with $|\mathcal{A}_j(t)|$ active queries, the total service capacity μ_j is shared equally among those active queries. Thus, the process $\{|\mathcal{R}_j(t)|\}_{t\geq 0}$ behaves as a birth–death chain with upward transition rate α_j and downward transition rate μ_j for every nonzero state. Hence, the stationary distribution of $\{|\mathcal{R}_j(t)|\}_{t\geq 0}$ is the same as that of an $M/M/1$ queue:

$$\pi_n \triangleq \Pr(|\mathcal{R}_j| = n) = (1-\rho_j)\rho_j^n, \qquad n = 0, 1, 2, \ldots,$$

where $\rho_j = \alpha_j/\mu_j < 1$.

Queueing delay of an arriving query. Let $W_{i,j}$ denote the queueing delay, i.e., the time from arrival until the query q_i first enters the active set at model instance j. By PASTA (Poisson Arrivals See Time Averages) [48], an arrival sees state $|\mathcal{R}_j| = n$ with probability π_n.

If the query arrives and sees $n < k$, it enters service immediately, hence

$$\mathbb{E}[W_{i,j} \mid |\mathcal{R}_j(t)| = n] = 0, \qquad n < k.$$

If it sees $n \geq k$, then k jobs are already active, and $n - k$ jobs are already waiting. Since the new query q_i joins the tail of the FIFO queue, it must wait for $n - k + 1$ service completions before entering service. While the query is waiting, the server always has k active jobs, so completions occur as a Poisson process of rate μ_j. Hence

$$\mathbb{E}\left[W_{i,j} \mid |\mathcal{R}_j(t)| = n\right] = \frac{n-k+1}{\mu_j}, \qquad n \geq k.$$

Averaging over the stationary state seen by arrivals gives

$$\begin{aligned}
\mathbb{E}[W_{i,j}] &= \sum_{n=0}^{\infty} \pi_n \, \mathbb{E}[W_{i,j} \mid |\mathcal{R}_j(t)| = n] = \sum_{n=k}^{\infty} (1-\rho_j)\rho_j^n \frac{n-k+1}{\mu_j} \\
&= \frac{(1-\rho_j)\rho_j^k}{\mu_j} \sum_{m=0}^{\infty} \rho_j^m (m+1) = \frac{(1-\rho_j)\rho_j^k}{\mu_j} \cdot \frac{1}{(1-\rho_j)^2} \\
&= \frac{\rho_j^k}{\mu_j(1-\rho_j)} = \frac{(\alpha_j/\mu_j)^k}{\mu_j - \alpha_j}.
\end{aligned}$$

Thus, under the Poisson + memoryless-service LPS model,

$$\boxed{\mathbb{E}[W_{i,j}] = \frac{(\alpha_j/\mu_j)^k}{\mu_j - \alpha_j}} \tag{12}$$

provided $\alpha_j < \mu_j$.

Interpretation. Equation (12) has the expected limiting behavior. For $k = 1$, LPS reduces to ordinary $M/M/1$, yielding $\mathbb{E}[W_q] = \rho_j/(\mu_j - \alpha_j)$, the usual FCFS waiting time. As $k \to \infty$, the waiting time vanishes: every query enters service immediately, which matches ordinary processor sharing. For fixed $\rho_j < 1$, the queueing delay decays geometrically in k through the factor ρ_j^k, reflecting the fact that waiting occurs only when all k active slots are already occupied.

Connection to TTFT estimation. In eq. (1), TTFT is decomposed as queueing delay plus prefill computation and first decode token generation time. If query i has prompt length $\text{tok}_{q_i,j}^{\text{pre}}$ and instance j has prefill throughput θ_j^{pre} tokens/sec, and average decode token batch processing time $\bar{T}_j^{\text{dec}}$, then

$$\widehat{P}_{i,j} = \frac{\text{tok}_{q_i,j}^{\text{pre}}}{\theta_j^{\text{pre}}} + \bar{T}_j^{\text{dec}}.$$

Combining this with (12) gives the estimator

$$\widehat{L}_{i,j}^{\text{ttft}} = \widehat{W}_{i,j} + \widehat{P}_{i,j} = \frac{(\alpha_j/\mu_j)^k}{\mu_j - \alpha_j} + \frac{\text{tok}_{q_i,j}^{\text{pre}}}{\theta_j^{\text{pre}}} + \bar{T}_j^{\text{dec}}.$$

Beyond Poisson arrivals and exponential/geometric service. For general response generation time distributions, the time until the next admission into generation depends on the *residual* decode sequence lengths of the k active jobs, requiring additional analysis beyond the standard Poisson Arrivals and exponentially distributed per query workload (i.e., geometrically distributed sequence length). The Limited Processor Sharing waiting time expression above admits a useful variability-corrected approximation. Let Q denote the per-query service requirement, with mean $\mathbb{E}[Q]$, and let $\rho_j = \alpha_j \mathbb{E}[Q]$, squared coefficients of variation of inter-arrival times and service requirement as c_a^2 and c_s^2 respectively. A simple approximation that is exact in the important special cases $k = 1$ and Poisson+exponential service is

$$\mathbb{E}[W_{i,j}] \approx \frac{c_a^2 + c_s^2}{2} \cdot \frac{\rho_j^k \, \mathbb{E}[Q]}{1 - \rho_j}. \tag{13}$$

When $k = 1$, this reduces to the standard $G/G/1$ waiting-time approximation, and for Poisson arrivals with exponential service ($c_a^2 = c_s^2 = 1$) it recovers (12). Hence, (13) can be viewed as a first-order extension of the exact LPS formula that accounts for arrival and service variability. In settings where the output-length distribution is strongly heavy-tailed or the arrival process is bursty, a more accurate alternative is to estimate queueing delay by direct simulation or by fitting more complex analytical representations of per query service requirement.

F Implementation of the SFS Latency Estimator

We describe the implementation of the Serving Framework Simulation (SFS) latency estimator in Section 4.1. Estimation is initiated by the router and uses periodically published query workload snapshots from model instances to predict the time-to-first-token (TTFT) latency a new request would experience at each candidate model instance. A desirable estimator should be computationally efficient and should not induce significant overhead for query routing.

Query workload snapshots. Each model instance captures and conveys to the router a compact summary of its current query computation workload after each token iteration through shared memory for computational efficiency. This snapshot contains the information utilized by the latency simulator at the router: the resident requests at the instance, their remaining prefill computation workload, predicted decode work, and the scheduler state needed for constructing token batches.

Robustness to incomplete snapshots. Since query workload snapshots are updated without pausing autoregressive generation at model instances, the router must avoid using partially written snapshots for simulation. We use a simple consistency mechanism in which each snapshot update is marked as either in-progress or complete. The router accepts a snapshot only when it observes that the update has completed and that the snapshot was not modified during the read. This allows the router to obtain a valid view of the instance state without blocking the serving engine. The router maintains a local cached copy of the most recent complete snapshot from each model instance, which is utilized for latency estimation upon arrival of new queries.

Latency estimation. When a new request q_i arrives, the router evaluates each candidate model instance j by simulating the batching and scheduling policies of the serving engine given the current workload, along with the new query. The resulting sequence of batches until the first output token generation of the new query (eq. (7)) determines the TTFT for q_i. We utilize a calibrated estimator (eq. (9)) to compute batch processing times, yielding the TTFT estimator (eq. (8)). This simulation accounts for queue order, admission limits, token-budget constraints, context limits, KV-cache allocation state, and preemption behavior when applicable.

Code structure. The implementation separates three components: the serving engine, workload snapshot collection, and routing-time evaluation. Each model instance periodically records its current query workload, and the router keeps a cached copy of the latest snapshot from each instance, using them for TTFT latency simulation when a new request arrives. The main routing logic, request dispatch, and policy evaluation are implemented using Python. Critical components such as snapshot parsing and batching policy simulation are implemented in C++ and exposed to Python endpoints. The simulation overhead (involving a loop over request states for construction of each token batch) for latency estimation is minimized through our C++ implementation.

The system consists of $\approx$3.6K lines of Python and C++. In particular, the token batch construction loop for the serving engine is implemented in $\approx$300 lines of C++ code. Adapting the estimator to a different serving engine or scheduling policy only requires modifying this batching simulation logic with minor modifications to router integration.

G Additional Experimental Results

Varying λ in utility eq. (2). Figure 12 denotes the OnTimeUtility of our approach as compared to *Latency-Agnostic* policy (*Shortest Queue* and *Round Robin* do not have λ parameter to control the accuracy–cost tradeoff). We observe that across different choices of λ values, our approach maintains higher performance. Note that the utility values naturally reduce as λ increases in eq. (2).

Query latency constraint attainment. We present the query latency attainment in Figure 13, which denotes that our approach maintains high SLO attainment for queries through accurate estimates of latencies across model instances that match with observed latencies. Combined with improved accuracy–cost utility, our approach shows the highest OnTimeUtility across all baselines as demonstrated in Figure 5.

Routing composition across model instances. Figure 14 shows the fraction of queries from each task routed to each model instance for our approach and the examined baselines.

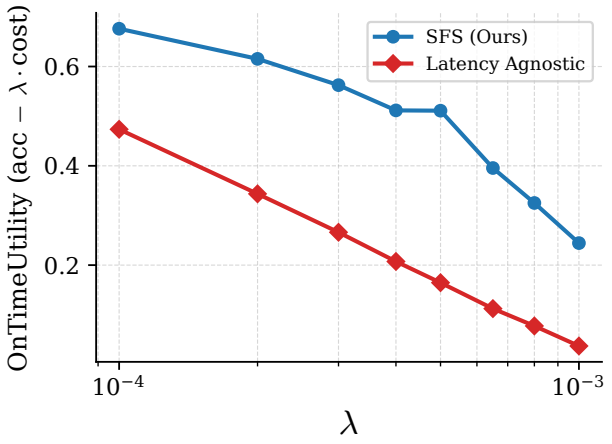

Figure 12: OnTimeUtility with varying accuracy–cost balance (λ) for our approach and latency agnostic routing. Consistent improvement across varying accuracy–cost balances indicates the generalizability of our approach.

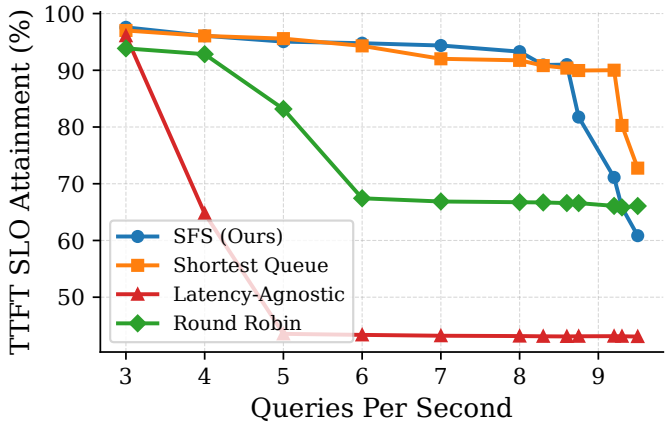

Figure 13: Latency constraint (SLO) attainment vs. query arrival rate. Our approach makes reliable latency estimates for queries across model instances, resulting in higher latency constraint satisfaction across baselines.

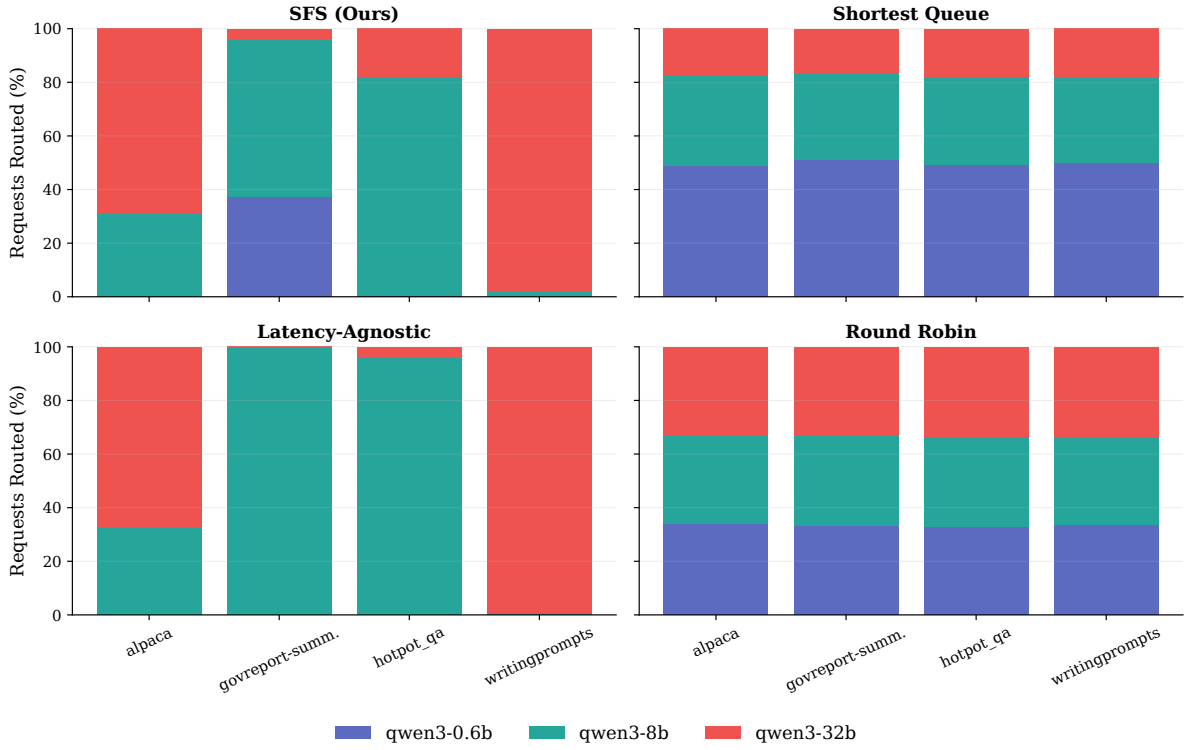

Figure 14: Fraction of queries routed to different model instances across approaches. *Shortest Queue* and *Round Robin* do not consider the accuracy–cost heterogeneity of models across queries while making routing decisions, hence routing composition is identical across each task. *Shortest Queue* routes more queries to smaller models as their faster throughputs result in smaller queues. In contrast, *SFS* jointly accounts for latency, accuracy, and cost, producing task-dependent routing decisions by assigning longer or latency-sensitive queries to faster models while still using larger models when their quality gains justify the cost.