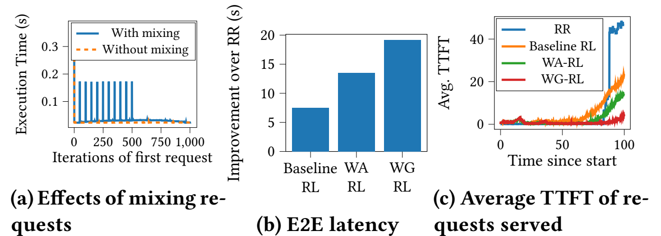
Performance Aware LLM Load Balancer for Mixed Workloads

Kunal Jain, Anjaly Parayil, Ankur Mallick, Esha Choukse, Xiaoting Qin, Jue Zhang, Íñigo Goiri,
Rujia Wang, Chetan Bansal, Victor Rühle, Anoop Kulkarni, Steve Kofsky, Saravan Rajmohan
Microsoft

Abstract

Large Language Model (LLM) workloads consist of distinct prefill and decode phases, each with unique compute and memory requirements that should be considered when routing input queries across cluster instances. However, existing load-balancing algorithms treat these workloads as monolithic jobs, ignoring the differences between the two phases. This oversight leads to suboptimal query distribution and increased response latency. In our work, we first characterize the factors affecting response latency during LLM inference. We show that balancing inference requests across available LLM instances can improve end-to-end latency more than simply optimizing the instance-level scheduler. Motivated by these findings, we propose a heuristic-guided, reinforcement learning-based router for data-driven, workload-aware scheduling. Our router distributes queries across LLM instances by using a trainable response-length predictor and a novel formulation for estimating the impact of mixing different workloads, achieving over 11% lower end-to-end latency than existing methods on mixed public datasets. Our framework represents a first step toward a holistic optimization framework and serves as a benchmark for deriving optimal load balancing strategies tailored to different reward functions and requirements. Beyond latency, we can extend the proposed framework to optimize for various performance criteria ensuring that the system meets diverse operational objectives.

ACM Reference Format:
Kunal Jain, Anjaly Parayil, Ankur Mallick, Esha Choukse, Xiaoting Qin, Jue Zhang, Íñigo Goiri, Rujia Wang, Chetan Bansal, Victor Rühle, Anoop Kulkarni, Steve Kofsky, Saravan Rajmohan, Microsoft . 2025. Performance Aware LLM Load Balancer for Mixed Workloads. In *The 5th Workshop on Machine Learning and Systems (EuroMLSys '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3721146.3721947



(a) Effects of mixing requests **(b) E2E latency** **(c) Average TTFT of requests served**

Figure 1: Key Results: (a) The red curve indicates trend in the execution time when a LLM instance serves a single request, while the blue curve shows spikes in the total execution time of a request due to the addition of additional requests to that instance at fixed intervals. (b) Our RL-based approaches improve upon Round-robin (RR) routing in terms of overall latency with Workload Guided RL (WG RL) reducing average end-to-end latency by 19.18 seconds. (c) The average Time-To-First-Token is the lowest for the proposed approach.

1 Introduction

The emergence of large language models (LLMs) and their generative ability has led to an increase in their usage in conversational engines, search engines, and code assistants Adiwardana et al. [1], Chen et al. [5], Roller et al. [35]. The widespread usage of these large models, coupled with the significant GPU compute required for inference, has made LLM inference the dominant GPU workload. Optimizing LLM inference is thus critical for improving user experience, lowering the pressure on GPU resources, and reducing the environmental impact of running LLM workloads, and so there has been a flurry of recent work looking at various aspects of LLM inference optimization Li et al. [21], Lin et al. [22], Spector and Re [36], Zhang et al. [44].

LLM inference is usually performed on the cloud by model instances hosted by commercial cloud providers [12, 26] or dedicated LLM inference platforms [14, 28] that serve inference requests from a variety of tenants. Owing to the widespread use of LLMs in chatbots, document summarization, and content creation, the requests vary in terms of their input and output characteristics. Each LLM instance that serves the inference request contains a scheduler, which is a batching system responsible for creating a batch of requests by retrieving requests from a queue and scheduling the execution engine. There exist multiple approaches in the literature that try to optimize the batching of these requests at a *single* LLM instance [2, 30, 43, 45] with various goals like reducing queueing delay of requests, maximizing the utilization of the serving infrastructure, etc. For similar reasons, works like [10, 27] have looked at routing requests across *multiple LLMs* (route easy requests to a small model and hard requests to a big model). However neither of these lines of work have considered *routing requests across multiple instances of a single LLM.*

This is a significant gap since all cloud providers host multiple instances of each model and need to design policies for assigning requests to instances such that they can be served with low latency. The wide variety in the size of input queries and LLM responses across scenarios implies that sub-optimal request assignment can significantly increase inference latency. Figure 1a shows spikes in execution time of a request when new requests are added while the LLM instance is still processing an initial request. The execution time of a request during each iteration is significantly impacted by the addition of new requests. We conducted an empirical study (see Figure 2) analyzing the disparity between optimal and random

This work is licensed under a Creative Commons Attribution 4.0 International License.
EuroMLSys '25, Rotterdam, Netherlands
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1538-9/2025/03
https://doi.org/10.1145/3721146.3721947

assignment of 8 requests arriving at a rate of 1 per second, with varying input and output lengths assigned to two LLM instances using set partitioning. Through exhaustive search of all possible combinations, we found that the best achievable latency was 27.03 seconds, the worst was 32.34 seconds, and a random assignment yielded a latency of 29.81 seconds. On average, a random assignment yields 10% higher end-to-end latency than the optimal assignment.

Given our optimization preferences, our objective is to identify the optimal set of requests to co-serve in each model instance, achieving the best performance metrics—in terms of both combination and distribution of requests—which will serve as a benchmark for future load-balancing strategies.

We begin by examining the interplay between routing strategies and instance-level scheduling. Poor request assignment at the instance level adversely affects latency; therefore, improved routing strategies that consider both incoming and ongoing request characteristics are needed. Optimization decisions for routing (load balancing) are critical and must be treated differently from those for instance-level scheduling to fully realize their benefits.

Our analysis starts with the prompt and decode phases of inference requests, where we estimate the execution time for a request given its prompt and decode lengths. We then study the factors influencing end-to-end latency in an LLM instance, including co-served requests and variations in prompt and decode distributions. By classifying requests based on these characteristics, we model the latency impact of their mixing and propose a latency impact estimator supported by a lightweight decode length predictor.

Finally, we introduce an intelligent router tailored to a specific combination of instance-level scheduler, LLM, and hardware. By strategically delaying routing and choosing the optimal model instance based on both current load and incoming requests, we reduce queuing at model instances, achieving an average latency improvement of 11.43% over 2,000 requests across four LLM instances. We also provide a preliminary evaluation of the framework's scalability, adaptability to different LLM-hardware combinations, and its performance on a real production trace.

Contributions: 1) We show that poor request assignment at the instance-level cannot improve end-to-end latency beyond a point and highlight the impact of concurrently serving diverse inference requests. 2) We introduce a novel formulation to model this impact. 3) We develop a lightweight decode length predictor that performs well across various prompt and decode characteristics. 4) We propose a heuristic-guided, workload-aware RL router that leverages prior knowledge of workload mixing to optimize the assignment of requests to LLM instances, thereby improving overall performance. Our intelligent router optimizes latency for a given hardware, model, and instance-level scheduler, setting a benchmark for future load balancers by finding the optimal distribution of requests across the instances. The framework also offers the flexibility to integrate additional optimizations, such as prefill chunking or prefix caching, to further enhance performance.

2 Preliminaries

Large Language Models (LLMs) go through prompt/ prefill and decode phases while serving a request

Scheduler block at LLM instance. Each LLM instance that serves the inference request contains a scheduler, which is a batching system responsible for creating a batch of requests by retrieving requests from a queue and scheduling the execution engine. The scheduler controls how many and which requests are processed in each iteration and may use techniques like iteration-level scheduling introduced in Yu et al. [43] to reduce queueing delay. The highlighted blocks in Figure 3 show the scheduler at each LLM instance. Often, the First-Come-First-Served policy is used for scheduling requests as online requests are latency-sensitive. We give a comprehensive review of the scheduling literature along with other relevant works in Appendix A.

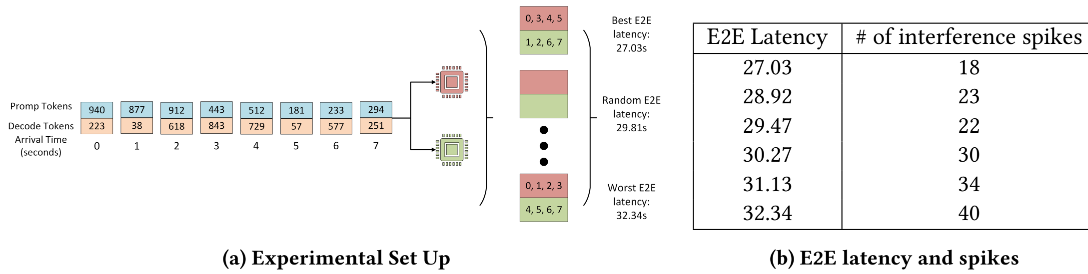
Problem Setup. We consider serving a stream of requests using multiple homogeneous LLM instances, each with a scheduler Yu et al. [43] to iteratively batch requests using a First-Come-First-Served policy. Requests vary in tasks like summarization, QnA, and translation, each with different prompt and decode characteristics. Requests queue centrally and are routed one at a time to an LLM instance with available capacity. Due to memory constraints, a request may be preempted mid-process if its response exceeds expected size. Our goal is to minimize end-to-end latency by assigning requests to LLM instances, assuming the request arrival rate maintains system equilibrium and given any optimization strategies present at the model-level scheduler.

3 Observation

Dataset. In general, LLM queries come from different tasks and differ in terms of their prompt and decode distribution. We simulate the prompt and decode distribution using data from five different tasks: sentiment analysis, entity recognition, in-context QnA, general QnA, and translation (prompt details in Appendix C.5). Input output distribution of the dataset and performance on our methods is summarized in the Appendix C.1 We see a clear distinction in the average length of prompt and decode tokens, and in the percentage of requests with heavy decode, across tasks.

Prompt-Decode characteristics of requests and their execution time. We start by characterizing the processing time of a request in terms of their prompt and decode token count. Figures 4a and 4b show that batch execution time increases linearly with the number of prompt tokens due to it's compute bound nature, and the growth is much slower during the decode phase. Thus, we estimate the processing time for a request with p prompt and d decode tokens as $p\times$`(time per prompt token)`$+d\times$`(average decode batch time)`. Similarly, the earliest time any model instance will become available is `(iterations left)` $\times$ `(average batch time)`. It is to be noted that Figures 4a and 4b correspond to the profiles for Llama-2-7b models on V100. A similar profiling approach can be followed for the processing time of a different LLM and hardware combination.

Consistency Across Hardware and Model Combinations: For consistency across different LLM and hardware combinations, we classify prompt and decode phases as either heavy or light based on their processing time for that LLM and hardware. Requests that take 0.5 seconds or more to complete their prompt phases are defined as heavy prompts, while requests that take 5 seconds or more in the decode phase are defined as heavy decodes. These values are hyperparameters that can be tuned to the provider's needs. It should



E2E Latency	# of interference spikes
27.03	18
28.92	23
29.47	22
30.27	30
31.13	34
32.34	40

(a) Experimental Set Up **(b) E2E latency and spikes**

Figure 2: (a) We assigned 8 requests, arriving at a rate of 1 per second, with input and output lengths varying from 10 to 100, to two LLM instances using set partitioning. Through exhaustive search of all possible combinations, we found that the best achievable latency was 27.03 seconds, the worst was 32.34 seconds, and random assignment resulted in an expected latency of 29.81 seconds. The ideal case is approximately 10% better than the average. (b) The figure shows the number of spikes and their corresponding E2E latency during the simulation. The ideal scenario has a maximum of seven spikes, but we observe a significantly higher number of spikes, suggesting request preemption

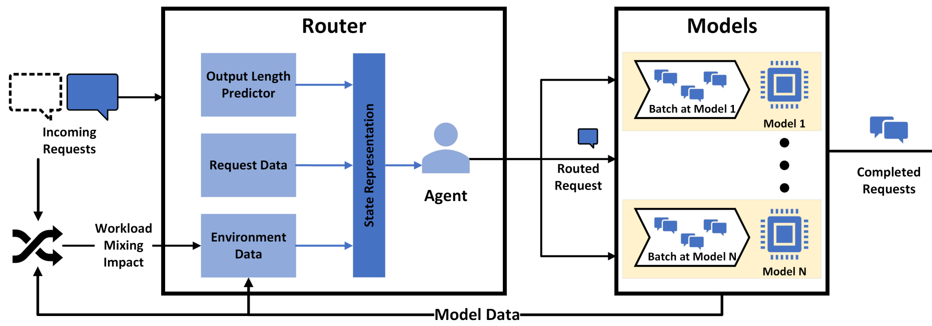
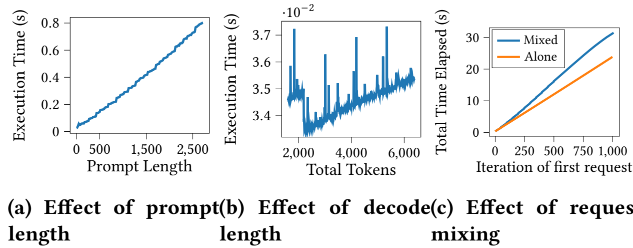


Figure 3: Our intelligent router optimizes request routing for end-to-end latency by using the output length predictor and workload impact estimator to route incoming requests to the appropriate model instance based on request characteristics and instance state. In contrast, current approaches focus on instance-level scheduling, as shown by highlighted regions around each model instance. Our router achieves optimal improvements regardless of the LLM instance's optimization strategy.



(a) Effect of prompt length **(b) Effect of decode length** **(c) Effect of request mixing**

Figure 4: Effects of prompt and decode tokens on batch execution time. (a) Execution time of batch with a request in prefill phase grows fast and linearly with the number of prefill tokens. (b) Execution time of batch with only decode tokens is affected to a much lesser degree with the number of tokens. (c) Increase in execution time on mixing requests with the arrival pattern of Figure 1a

be noted that input prompts with 1024 tokens may be heavy for a V100, but they may not be classified as heavy for an H100 due to the latter's better processing capability. We then divide all incoming requests into four categories: light prompt-light decode (LL), light prompt-heavy decode (LH), heavy prompt-light decode (HL), and heavy prompt-heavy decode (HH). Next, we quantify the factors affecting the latency and end-to-end performance of LLM inference.

Effect of co-serving requests in the prompt and decode phase on a single LLM instance. To analyze the effect of processing requests in their prompt and decode phase on a single machine, we served a single request on a LLM instance and added requests at fixed intervals while the first request is still in its decode phase. As shown in Figure 1a, the execution time of the first request experienced spikes when new requests were added while the LLM instance was still processing an initial request. The orange curve in Figure 4c shows the ideal latency for the first request, which is 17 seconds for a request with $p = 1000$ and $d = 1000$. However, the end-to-end latency increases to 31 seconds due to the arrival of new requests of $p = 500$ and $d = 500$ at every 50th iteration. The decode phase of the first request experienced increase in execution time due to the latency spikes caused by the prompt phase of each of the incoming requests [13].

Effect of serving requests with distinct prompt and decode characteristics on a single LLM instance. Recall that: (i) the

latency of the incoming request during the prompt phase increases rapidly and linearly with an increase in the number of prompt tokens, and (ii) the decode phase has a much lower impact, and the mean iteration time varies slowly with an increase in total tokens. As we can see from Figure 4 (Llama-2-7b models profiled on V100 GPUs), the gradient for our configuration can be calculated as 3.2×10^{-4} and 3.3×10^{-5}. Thus mixing requests with different prompt and decode characteristics in a batch at a model instance can impact overall latency due to mismatch and interference between the prompt and decode phase of the requests. We see a roughly linear increase in batch execution with an increase in token count. The same approach can be followed for other model and hardware combinations.

Factors affecting E2E Latency. In real-world scenarios, multiple LLM instances serve requests. We analyze how routing strategies and model-level scheduling algorithms affect end-to-end latency. Using two LLM instances and 3000 requests, we examine four arrival patterns: 1) random LH and HL mix; 2) random mix of all four classes; 3) LH requests first, then HL; and 4) HL first, then LH. We compare various batching and routing combinations (see Appendix) and find that their interplay determines overall latency.

For example, the bin-packing scheduler finishes Scenario I six seconds faster with round-robin routing than with decode balancer routing—but the same setup is four seconds slower in Scenario II. Suboptimal strategies, such as dedicating servers for specific request sizes, can severely harm performance. Notably, in Scenarios III and IV, all batching algorithms yield identical latency, with routing being the decisive factor. This underscores that scheduling effectiveness depends on the router's choices.

Insight: Request characteristics and arrival patterns must guide routing decisions. Since scheduling algorithms can only perform as well as the router's assignment, the next section proposes optimal routing strategies tailored to specific model-level schedulers, hardware, and model combinations, detailing the router's design and its key components.

4 Intelligent Router: Design

Figure 3 shows the overall design of the intelligent router. Based on the insights from section 3, an intelligent router should: a) classify requests based on prompt-decode characteristics and be able to estimate decode length, b) estimate the adverse effect of mixing diverse requests at a single model instance on end-to-end latency, c) leverage prior knowledge of these adverse effects for decision making, and d) possess lightweight modules for efficient processing. To achieve this, we develop an output length predictor and workload impact estimator for intelligent routing. Additionally, we propose a reinforcement learning framework to utilize accumulated context and prior knowledge, improving end-to-end latency.

4.1 Output length predictor

Similar to [18], we generate responses for each request from our dataset (see Section Observation) and bucket them based on the number of output tokens. These buckets serve as labels to fine-tune a DistillBERT model for predicting output token ranges for new requests. Instead of equal-sized buckets, we define ranges based on estimated completion times—e.g., 0–0.5 seconds, 0.5–2 seconds, 2–4 seconds—translating to roughly 0–250, 250–1000, and 1000–4000 tokens given our 500 tokens/sec throughput. This choice better distributes requests and aligns ranges with expected times.

We found that Jin et al.'s approach does not generalize to our dataset, with accuracy only 5.5% for unequal buckets and 9.3% for equal 250-token buckets. By appending the task type as a hint to the prompt—a reflection of our observation that input/output characteristics depend on task—we boost accuracy to 79.15% for unequal buckets and 68.23% for equal buckets, while task type prediction itself reaches 93.79

4.2 Workload impact estimator

Next, we use the profiling approach from Section 4 to obtain the analytic expression for the processing times of the prompt and decode phase. Let there be n requests within model instance m, and p_j^m and d_j^m indicate the number of prompt and decode tokens processed by the j-th existing request at model m. As the impact on the prompt phase is directly proportional to the number of prompt tokens in the request and the total number of tokens already running in the decode phase, we can model the impact on the prompt phase (time to process p_i, $T_{p_i}^m$) of an incoming request with p_i tokens when added to the model instance with n requests, and corresponding penalty as:

$$\begin{aligned} T_{p_i}^m &= \mathtt{grad}_1 * \left((p_i^2 + \textstyle\sum_{j=1}^n \left(p_j^m + d_j^m \right) \right) \\ r_{p_i}^m &= \begin{cases} 1 & \text{if } T_{p_i}^m \leq \epsilon \\ 1 - \frac{T_{p_i}^m}{\epsilon} & \text{otherwise} \end{cases} \end{aligned} \tag{1}$$

Here, we introduce a penalty if the latency impact exceeds ϵ. Similarly, the impact on existing requests beyond the prompt phase is directly proportional to the total number of requests in the model. We can model the penalty due to the impact of an incoming request with p_i prompt tokens and d_i decode tokens on the decode phase of already existing n requests as:

$$r_d^m = -\mathtt{grad}_2 * \sum_{j=1}^{n} \left(p_j^m + d_j^m \right) + p_i + d_i \tag{2}$$

With our selection of $\mathtt{grad}_1$ as 3.2×10^{-4} and $\mathtt{grad}_2$ as 3.3×10^{-5}, we expect the values r_d^m and $T_{p_i}^m$ to be in the ranges of $[-1, 1]$ and $[-1, 0]$ respectively when there are no requests waiting at the model instance. We combine Equation 1 and Equation 2 to get the final penalty of mixing requests: $r_{\mathtt{mixing}}(s_t, s_{t+1}) = \alpha r_{p_i}^m + (1 - \alpha) r_d^m$ where m is the action taken for the state transition $s_t \longrightarrow s_{t+1}$. Here, parameter $\alpha \in (0, 1)$ balance priority over the prompt and decode phases.

4.3 RL based router

We formulate the problem of routing incoming requests to the m model instances as discrete-time Markov Decision Process (MDP) and propose a reinforcement learning-based solution [39]. We assume an arrival rate of λ for the requests and the ideal estimated time to complete request i as $\hat{T}_i$. Let o_{jt} denote the total number of output tokens produced until time t by request j, and $\hat{d}_{jt}$ denote the estimated decode tokens for the j-th request. Therefore, we denote the fraction of request j completed at time t by $f_{jt} \coloneqq \frac{o_{jt}}{\hat{d}_{jt}}$.

Batching Algorithm	Routing Algorithm	Total End to End Latency (seconds)			
		(LH, HL random)	(Random)	(LH, then HL)	(HL, then LH)
Bin Packing [18]	Dedicated Small-Large	704.5	644.75	566.25	588.15
	Round Robin	**581.5**	559.3	424.8	**440.68**
	Decode Balancer	595.82	555.4	424.82	440.81
Least Work Left	Dedicated Small-Large	704.5	641.81	566.25	588.15
	Round Robin	585.14	**554.00**	**424.64**	440.82
	Decode Balancer	596.95	559.97	424.66	440.81
FCFS [43]	Dedicated Small-Large	704.5	648.66	566.25	588.15
	Round Robin	607.45	572.16	424.80	440.82
	Decode Balancer	605.65	573.17	424.82	440.81

Table 1: Performance of batching and routing algorithm combinations. We simulate arrival of requests with distinct characteristics using the request classification discussed in section 3 and test the combined affect of routing and batching strategies. Good routing algorithm on an average shows greater end-to-end latency improvement compared to the batching algorithm on all the scenarios with distinct characteristics and arrival sequence.

We assume state transition at every Δt, where Δt is the average time to generate a decode token.

State Space: At time t, the state of the system, which comprises m model instances and requests waiting in the queue, can be captured by the following: 1) The number of requests in the queue at time t, denoted by w_{q_t}; 2) The exact number of prompt tokens, denoted by $p_t \in \mathbb{R}$, and the estimated bucket for the decode tokens, denoted by $d_t \in \{0, \ldots, n_d\}$, corresponding to the next request in the queue. Here, the estimated bucket varies from zero to n_d; 3) Matrices, $\mathbf{P}_t \in \mathbb{R}^{mn_p}$ and $\mathbf{D}_t \in \mathbb{R}^{mn_d}$, capturing the prompt and decode distribution of requests at the model instances. We represent the prompt (decode) distribution by n_p (n_d) buckets. $(\mathbf{P}_t)_{i,j}$ ($(\mathbf{D}_t)_{i,j}$) denotes the number of requests in prompt (decode) phase at the i-th model instance that are present in the j-th bucket i.e. have j prompt (decode) tokens. We represent the prompt and decode distribution across the model instances as a matrix, which maintains the finite dimensionality of the state space. 4) The capacity available at the model instances at time t is denoted by $\mathbf{C}_t \in \mathbb{R}^m$, as a function $g(\texttt{batch size}, \mathbf{P}_t, \mathbf{D}_t)$; and 5) The estimated completion time for the earliest request in model j, denoted by $\hat{T}_{c_t}$.

Action Space: At any given point in time, the agent must decide whether to schedule incoming request i to any of the m model instances or choose to take no action. Therefore, $a \in \{0, \ldots, m\}$. Here, index m refers to no action being taken by the router.

Reward Design: Based on the insights from section 3, we include the following elements in the reward formulation: a) a negative penalty for requests in the queue, decreasing as requests are processed, to account for the autoregressive nature of requests, b) a positive reward for each completed request, and c) a workload impact estimator-based penalty, which encodes the adverse effect of routing specific requests to a model instance with existing requests, and prevents requests from being queued at each model instance due to a lack of memory. Note that adding the workload impact estimator-based penalty directly to the reward function might introduce bias. Therefore, we propose to augment the prior knowledge using a heuristic-guided formulation [6], and the reward at time t is given by:

$$r_t = -\sum_{j \in \mathcal{J}} \left(\frac{1}{\hat{T}_j} \left(1 - f_{jt}\right) \right) + \sum_{j=1}^{m} \sum_{i} r_w \times \mathbf{w}_{mi_t} - (\gamma - \tilde{\gamma}_k) h(\mathbf{s}_t, \mathbf{s}_{t+1}) \tag{3}$$

where

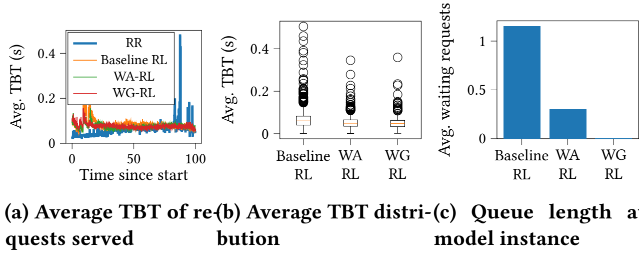
$$h(s_t, s_{t+1}) = r_{\texttt{mixing}}(\mathbf{s}_t, \mathbf{s}_{t+1}) - \max_{l=1,\ldots,m} (r_{\texttt{mixing}}(\mathbf{s}_t, \mathbf{s}^l_{t+1})) \tag{4}$$

Here, $\mathcal{J}$ includes the set of scheduled and unscheduled requests, and $\mathbf{w}_{mi_t} \in \{0, 1\}$ indicates whether the ith at the mth model completed at time t. $r_w \in \mathbb{Z}^+$ is the positive reward for completing a request. The function $h : \mathbf{S} \times \mathbf{S} \rightarrow \mathbb{R}$ represents the difference in penalty due to assigning the incoming request to a model other than the one for which the impact is minimum (a function of equations Equation 1 and Equation 2). The term "guidance discount" is given by $\tilde{\gamma}_k = \lambda_k \gamma$, where the subscript k denotes the k-th episode. Here, $\lambda_k \in [0, 1]$ is the mixing coefficient and settles to zero with an increase in episodes [6]. The discount factor in the MDP is set to $\tilde{\gamma}_e$ during training. The function $h()$ returns zero when the request is assigned to the model with the least workload mixing impact. Intuitively, the formulation introduces horizon-based regularization, and its strength diminishes as the mixing coefficient increases, which modifies the original MDP, $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, r, \gamma)$, to $\tilde{\mathcal{M}} = (\mathcal{S}, \mathcal{A}, P, \tilde{r}, \tilde{\gamma})$. Over the course of training, the agent interacts with the environment, and the effects of the heuristic in the MDP decrease, and the agent eventually optimizes for the original MDP. Guarantees on the boundedness of the reshaped MDP's value function directly translate from [6].

5 Experiments

We conduct experiments to evaluate the performance improvement of the intelligent router comapred to different heuristics on the dataset presented in Section 3.

Evaluation metrics: For all the experiments, we report the end-to-end latency, Time-To-First-Token (TTFT), which is the time taken for the user to see the initial response, and Time-Between-Tokens (TBT), which is the average token streaming latency [30]. Additionally, we report the throughput achieved by different approaches. We included three variants of the RL formulation, including the



(a) Average TBT of requests served **(b) Average TBT distribution** **(c) Queue length at model instance**

Figure 5: Experimental Results. We simulate 2000 requests with distinct characteristics arriving at 20 per second, averaging results over 20 episodes. Initially, Round-Robin yields better average TBT, but its performance degrades over time as the request pool diversifies. In contrast, Workload Guided RL minimizes variance in both waiting requests and TBT, keeping the waiting queue shorter than alternative methods.

baseline RL formulation with a reward function consisting of only the first and second terms from Equation 3, workload-augmented RL which simply adds the penalty from $r_{\texttt{mixing}}$ to the baseline RL (workload knowledge augmented), and workload-guided RL that uses the heuristic-guided formulation 3.

Setup We route requests between four instances of LLama-2-7b-hf model [41] on a cluster of four V100 GPUs using vLLM [20] with its default First-Come-First-Served (FCFS) scheduler for iteration-level scheduling. We assume an average request arrival rate of $\lambda = 20/s$, with requests uniformly sampled at random from the dataset in Section 3. The routing of requests to model instances is asynchronous, and we take actions every 0.02 seconds, which is the minimum decode batch execution time.

For baseline RL, we set $\gamma - \tilde{\gamma}_e = 0$ in the reward function from Equation 3. For workload-aware RL, we directly augment the penalty for mixing requests to the reward function. Therefore, we set $\gamma - \tilde{\gamma}_e = 1$. For all experiments, we give equal weight to the impact on the prompt and decode phase. Therefore, α for equations Equation 1 and Equation 2 is set to 0.5. For workload-guided RL, we use the guidance mechanism from Equation 3. We set $\lambda_k = e^{-\beta_d k}$ (exponential decay over each episode) with $\beta_d = 0.5$, and the guided discount factor for training γ as $\hat{\gamma} = (1 - e^{-\beta_d k})\gamma$. Additional details on the model training are added to Table C.1.

5.1 Performance evaluation of Intelligent router

Here we compare the performance improvements with respect to various heuristics.

End-to-end latency: We evaluate RL based approaches over 20 episodes, each comprising 2,000 requests with distinct characteristics. As shown in Figure 1b, our methods outperformed Round-Robin in terms of end-to-end latency for servicing all requests. Baseline RL surpassed Round Robin by an average of 7.53 seconds (4.35%). Incorporating the workload-aware penalty into the reward function enhanced this advantage to 13.50 seconds (7.79%), and utilizing the penalty as heuristic guidance for the RL agent improved the advantage to 19.18 seconds (11.43%). This is intuitive as heuristics should only be employed as a warm start and should be reduced as the agent collects more information about the environment.

Classical heuristics such as Join Shortest Queue, Maximum Capacity Usage, and Min-Min Algorithm [4] only marginally outperformed Round Robin by 0.46%, 2.60%, and 1.50%, respectively, in terms of end-to-end latency. These results are intuitive as classical heuristics do not translate well for LLM workloads due to their unique nature. Due to this, we only provide further results in comparison to Round Robin. We provide further details on these algorithms in the appendix subsection C.3.

Improvements in TTFT and TBT: RL-based approaches outperformed the Round-robin router in terms of average TTFT (Figure 1c), with significant improvement as the number of accumulated requests increased over time. Baseline RL halved the TTFT for late-arriving requests by finding a better assignment than Round-robin. Workload-aware penalty further enhanced these decisions, but not optimally, as it diluted the urgency to complete requests promptly and introduced constant bias. Workload-guided RL performed the best by selecting more optimal model instances and mitigating spikes in TBT of existing requests (Figure 5a). Although Round-robin performed better initially in terms of average TBT, the value increased over time as more requests with different characteristics accumulated. Workload awareness effectively reduced the number of outliers and the variance of the distribution (Figure 5b).

Queuing at Router and Model Instance: Figure 5c illustrates the average length of the waiting queue at the model instances. While Baseline RL exhibited a shorter average waiting time of 0.59 seconds at the router, the requests got preempted at the model instances and accumulated substantial delays. This approach was suboptimal since postponing the routing decision could have resulted in a better model instance getting assigned and resulted in faster processing of the request. In contrast, Workload-aware RL, with an average router wait time of 4.41 seconds, addressed this issue by incorporating a penalty based on the workload. Workload Guided RL further refined this strategy by utilizing the penalty as a guidance mechanism, resulting in an average router wait time of 2.05 seconds and improved overall performance.

6 Limitations and Conclusion

We introduce a heuristic-guided RL router to efficiently schedule requests across homogeneous LLM instances. By modeling the impact of concurrently serving diverse workloads, our approach leverages prior knowledge of workload mixing to improve end-to-end latency. Our formulation generalizes to enhance key metrics like Time-To-First-Token (TTFT) and Time-Between-Two-Tokens (TBT), with the flexibility to incorporate additional requirements such as serving throughput. Experimental evaluations show our method outperforms current baselines across various models, hardware setups, and datasets. While introducing some overhead compared to pure heuristics, this framework paves the way for identifying optimal load balancing strategies for any given model-hardware combination and instance-level scheduler. We envision it setting a new benchmark for future inference scheduling research. and lays the groundwork for a versatile and comprehensive scheduling framework that can be tuned to a wide range of models, hardware configurations, and performance objectives.

References

[1] Daniel Adiwardana, Minh-Thang Luong, David R So, Jamie Hall, Noah Fiedel, Romal Thoppilan, Zi Yang, Apoorv Kulshreshtha, Gaurav Nemade, Yifeng Lu, et al. 2020. Towards a human-like open-domain chatbot. *arXiv preprint arXiv:2001.09977* (2020).

[2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. arXiv:2403.02310 [cs.LG]

[3] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. 2023. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369* (2023).

[4] Huankai Chen, Frank Wang, Na Helian, and Gbola Akanmu. 2013. User-priority guided Min-Min scheduling algorithm for load balancing in cloud computing. In *2013 National Conference on Parallel Computing Technologies (PARCOMPTECH)*. 1–8. doi:10.1109/ParCompTech.2013.6621389

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[6] Ching-An Cheng, Andrey Kolobov, and Adith Swaminathan. 2021. Heuristic-guided reinforcement learning. *Advances in Neural Information Processing Systems* 34 (2021), 13550–13563.

[7] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems* 35 (2022), 16344–16359.

[8] Leon Derczynski, Eric Nichols, Marieke van Erp, and Nut Limsopatham. 2017. Results of the WNUT2017 Shared Task on Novel and Emerging Entity Recognition. In *Proceedings of the 3rd Workshop on Noisy User-generated Text*. Association for Computational Linguistics, Copenhagen, Denmark, 140–147. doi:10.18653/v1/W17-4418

[9] Dujian Ding, Sihem Amer-Yahia, and Laks VS Lakshmanan. 2022. On Efficient Approximate Queries over Machine Learning Models. *arXiv preprint arXiv:2206.02845* (2022).

[10] Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Ruhle, Laks VS Lakshmanan, and Ahmed Hassan Awadallah. 2024. Hybrid LLM: Cost-efficient and quality-aware query routing. *arXiv preprint arXiv:2404.14618* (2024).

[11] Angela Fan, Yacine Jernite, Ethan Perez, David Grangier, Jason Weston, and Michael Auli. 2019. ELI5: Long Form Question Answering. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, Anna Korhonen, David R. Traum, and Lluís Màrquez (Eds.). Association for Computational Linguistics, 3558–3567. doi:10.18653/v1/p19-1346

[12] Google. [n. d.]. Vertex AI. https://cloud.google.com/vertex-ai.

[13] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, et al. 2024. Inference without Interference: Disaggregate LLM Inference for Mixed Downstream Workloads. *arXiv preprint arXiv:2401.11181* (2024).

[14] HuggingFace. [n. d.]. Hugging Face Inference API. https://huggingface.co/inference-api.

[15] Neharika Jali, Guannan Qu, Weina Wang, and Gauri Joshi. 2024. Efficient Reinforcement Learning for Routing Jobs in Heterogeneous Queueing Systems. *arXiv preprint arXiv:2402.01147* (2024).

[16] Siddharth Jha, Coleman Hooper, Xiaoxuan Liu, Sehoon Kim, and Kurt Keutzer. 2024. Learned Best-Effort LLM Serving. *arXiv preprint arXiv:2401.07886* (2024).

[17] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088* (2024).

[18] Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. 2023. S^3: Increasing GPU Utilization during Generative Inference for Higher Throughput. In *Thirty-seventh Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=zUYfbdNl1m

[19] Anil Kag, Igor Fedorov, Aditya Gangrade, Paul Whatmough, and Venkatesh Saligrama. 2022. Efficient Edge Inference by Selective Query. In *The Eleventh International Conference on Learning Representations*.

[20] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[21] Jiamin Li, Le Xu, Hong Xu, and Aditya Akella. 2024. BlockLLM: Multi-tenant Finer-grained Serving for Large Language Models. *arXiv preprint arXiv:2404.18322* (2024).

[22] Bin Lin, Tao Peng, Chen Zhang, Minmin Sun, Lanbo Li, Hanyu Zhao, Wencong Xiao, Qi Xu, Xiafei Qiu, Shen Li, et al. 2024. Infinite-LLM: Efficient LLM Service for Long Context with DistAttention and Distributed KVCache. *arXiv preprint arXiv:2401.02669* (2024).

[23] Jiachen Liu, Zhiyu Wu, Jae-Won Chung, Fan Lai, Myungjin Lee, and Mosharaf Chowdhury. 2024. Andes: Defining and Enhancing Quality-of-Experience in LLM-Based Text Streaming Services. *arXiv preprint arXiv:2404.16283* (2024).

[24] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning Word Vectors for Sentiment Analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Portland, Oregon, USA, 142–150. http://www.aclweb.org/anthology/P11-1015

[25] Daniel Mendoza, Francisco Romero, and Caroline Trippel. 2024. Model Selection for Latency-Critical Inference Serving. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 1016–1038.

[26] Microsoft. [n. d.]. Azure AI Studio. https://ai.azure.com/.

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

[28] OpenAI. [n. d.]. OpenAI Platform. https://platform.openai.com/overview.

[29] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[30] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. 2023. Splitwise: Efficient generative LLM inference using phase splitting. arXiv:2311.18677 [cs.AR]

[31] Archit Patke, Dhemath Reddy, Saurabh Jha, Haoran Qiu, Christian Pinto, Shengkun Cui, Chandra Narayanaswami, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2024. One Queue Is All You Need: Resolving Head-of-Line Blocking in Large Language Model Serving. arXiv:2407.00047 [cs.DC] https://arxiv.org/abs/2407.00047

[32] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2024. vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention. *arXiv preprint arXiv:2405.04437* (2024).

[33] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew T Kalbarczyk, Tamer Başar, and Ravishankar K Iyer. 2024. Efficient Interactive LLM Serving with Proxy Model-based Sequence Length Prediction. *arXiv preprint arXiv:2404.08509* (2024).

[34] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100,000+ Questions for Machine Comprehension of Text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, Jian Su, Kevin Duh, and Xavier Carreras (Eds.). Association for Computational Linguistics, Austin, Texas, 2383–2392. doi:10.18653/v1/D16-1264 arXiv:1606.05250 [cs.CL]

[35] Stephen Roller, Emily Dinan, Naman Goyal, Da Ju, Mary Williamson, Yinhan Liu, Jing Xu, Myle Ott, Kurt Shuster, Eric M Smith, et al. 2020. Recipes for building an open-domain chatbot. *arXiv preprint arXiv:2004.13637* (2020).

[36] Benjamin Spector and Chris Re. 2023. Accelerating llm inference with staged speculative decoding. *arXiv preprint arXiv:2308.04623* (2023).

[37] Alessandro Staffolani, Victor-Alexandru Darvariu, Paolo Bellavista, and Mirco Musolesi. 2023. RLQ: Workload allocation with reinforcement learning in distributed queues. *IEEE Transactions on Parallel and Distributed Systems* 34, 3 (2023), 856–868.

[38] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: Dynamic Scheduling for Large Language Model Serving. *arXiv preprint arXiv:2406.03243* (2024).

[39] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA.

[40] Jörg Tiedemann. 2012. Parallel Data, Tools and Interfaces in OPUS. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC'12)*, Nicoletta Calzolari, Khalid Choukri, Thierry Declerck, Mehmet Uğur Doğan, Bente Maegaard, Joseph Mariani, Asuncion Moreno, Jan Odijk, and Stelios Piperidis (Eds.). European Language Resources Association (ELRA), Istanbul, Turkey, 2214–2218. http://www.lrec-conf.org/proceedings/lrec2012/pdf/463_Paper.pdf

[41] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]

[42] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920* (2023).

[43] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 521–538. https://www.usenix.org/conference/osdi22/presentation/yu

[44] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. 2024. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems* 36 (2024).

[45] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. *arXiv preprint arXiv:2401.09670* (2024).

A Related Work

A.1 LLM Serving Systems

Recent advancements in inference serving systems for LLMs have focused on optimizing throughput, latency, and resource management. ORCA Yu et al. [43], Sarathi Agrawal et al. [3], FlashAttention Dao et al. [7], and vAttention Prabhu et al. [32] are examples of systems that have achieved significant improvements in performance through techniques such as iteration-level scheduling, innovative batching, and IO-aware algorithms.

A.2 LLM Serving Algorithms

This space has also seen several algorithmic innovations. QLM (Patke et al. [31]) utilizes Bayesian statistics and stochastic programming to manage non-deterministic execution times inherent in LLMs. Similarly, Qiu et al. [33] advocates for speculative shortest-job-first scheduling, and Wu et al. [42] employs preemptive scheduling to improve performance. DistServe and Splitwise (Patel et al. [30], Zhong et al. [45]) optimize LLM serving performance by separating prefill and decoding computation for throughput enhancement while maintaining low latency. In addressing system load and request patterns, Jha et al. [16] and Mendoza et al. [25] utilize deep reinforcement learning to dynamically adjust service quality, increasing hardware utilization for cost-efficient serving. Additionally, Liu et al. [23] optimize Quality-of-Experience (QoE) for LLM serving systems, focusing on user-centric metrics to enhance individual experiences. Patke et al. [31], Sun et al. [38] proposes a multi-model queue management framework for LLM serving and orchestrate the actions such as model swapping, request eviction, GPU-CPU state swapping, load balancing, and warm model start. While these works optimize request scheduling at the instance level, they ignore the diversity in the prompt and decode characteristics across requests.

A.3 Hybrid LLM Inference

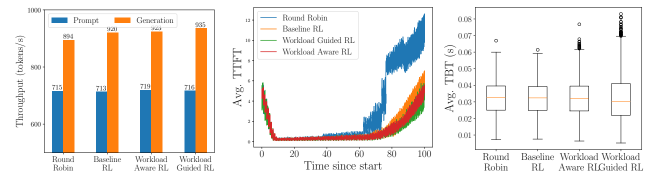
Recent works [9, 10, 19, 27] have introduced a hybrid inference paradigm which uses two *different models* instead of a single model for inference. The key idea is to save inference cost without compromising on response quality by routing easy queries to the smaller and less capable model (e.g. Mixtral [17]) while the difficult queries are routed to the larger and more capable model (e.g. GPT-4 [29]). The routing is typically achieved by training a query-difficulty classifier and is thus different from our reinforcement learning based router which seeks to find the optimal assignment of requests across different instances of the *same model.*

A.4 Reinforcement Learning for routing jobs

Reinforcement Learning (RL) has been a natural choice for routing jobs in multi-server queues owing to the challenges in deriving exact policies. While previous works [15, 37] have looked at general jobs, in this work we leverage the specific characteristics of LLM requests and insights from our workload-study to design novel workload aware RL approaches for routing inference requests across LLM instances.

Routing Algorithm	Prefill Chunking	Avg. E2E Latency (s)	Improvement
Round Robin	No	248.41	-
Baseline RL	No	240.58	3.15%
Workload Aware RL	No	231.66	6.74%
Workload Guided RL	No	221.80	10.71%
Round Robin	Yes	247.30	0.45%
Baseline RL	Yes	240.68	3.11%
Workload Aware RL	Yes	231.12	6.96%
Workload Guided RL	Yes	220.93	11.06%

Table 2: Intelligent router was able to generalize the approach across different model and hardware combinations, outperforms heuristics, and shows additional improvements even with chunked prefills.



(a) Throughput on A100s with Llama 3.1 **(b) Average TTFT with chunking** **(c) TBT distribution with chunking**

Figure 6: Model and hardware generalizability: Experiments on A100s with Llama 3.1 8B shows that intelligent router maintains prompt and generation throughput similar to Round Robin. Intelligent router still outperform Round Robin in the presence of optimizations such as chunked prefills.

B Additional Results

B.1 Different LLM and Hardware combination

We conducted experiments on different LLM and hardware combinations, specifically testing on A100 with Llama-3.1-8B. Due to better processing capabilities, we increased the arrival rate to 80 RPS, benchmarked the gradients again for the hardware/model combination, and retrained our agent with the same remaining hyperparameters. With more requests coming in, the router had many more decisions to make. Even then, our strategies were able to outperform Round Robin by similar margins (10.81%), as shown by the first four rows of Table 2. We observe in Figure 6a that our methods maintain prompt and generation throughput similar to Round Robin. Round Robin exhibits similar throughput over a longer period of time, indicating that it generates more tokens to service the same number of requests, highlighting the impact of request preemption. Additional experiments were conducted to validate the scalability of the proposed approach, and the results are presented in subsection C.11. The intelligent router outperformed Round Robin by 11.62% when evaluated on a setting with eight LLM instances.

B.2 Performance in the presence of SOTA Optimizations

Next, we will evaluate the performance of the intelligent router in the presence of chunked prefill tokens [3]. The aim is to assess the

performance improvements achieved by the intelligent router in the context of optimizations at the instance-level scheduler.

For Round-Robin, chunked prefill tokens only improves performance by 0.45%, which could be due to experimental noise. Chunking is not primarily intended to improve E2E latency but rather to enhance user experience by reducing TBT/decode throughput at the expense of TTFT. However, we observe that our method is able to adapt well to this new setting and maintain its lead over Round Robin. Figure 6b shows that the intelligent router still improves TTFT with chunking, despite the fact that chunking is supposed to harm TTFT. Figure 6c shows that TBT has much less variance now, and the average TBT across methods is the same. The performance gains are intuitive, as the intelligent router prevents preemptions of requests and selects the best suitable LLM instance for each request based on the request characteristics and other requests currently being served by each instance. Additional experiments that validate performance improvements on a different dataset, which is the real production trace from Cloud provider X, have been added to subsection C.12.

C Appendix / supplemental material

C.1 Performance on Dataset

C.2 Batching and Routing Algorithms

All the batching algorithms are non-preemptive in nature, meaning that once the processing of a request has started, it is prioritized over requests which have not. Next, we discuss different batching and routing algorithms defined in section 3.

C.2.1 Batching: Bin Packing Algorithm. When a new request's processing can be started, we select the largest request that can fit into the memory available. Ties are broken by FCFS.

C.2.2 Batching: Least Work Left. Among the requests available, we select the request with the smallest number of decode tokens.

C.2.3 Batching: FCFS. The request which arrives first is processed first.

C.2.4 Routing: Dedicated Small-Large. For the two LLM model instances, we dedicate one instance for servicing only the heavy-decode requests while the other model instance services only the light-decode requests.

C.2.5 Routing: Round Robin. Each of the two model is user alternatively by the router to send requests to.

C.2.6 Routing: Decode Balancer. We assume that the total number of output tokens is known beforehand for the request and we balance the sum of decode tokens on both the model instances.

C.3 Additional Baselines

We implemented three baselines other than Round Robin and the Light-weight Heuristic:

C.3.1 Join Shortest Queue. Each arriving request is routed to the model with the least number of prompt and decode tokens yet to be processed.

C.3.2 Maximum Capacity Usage. Request at the front of the queue is routed to the model with the maximum capacity available, given that it can process this particular request, at intervals of one second.

C.3.3 Min-Min Algorithm. We implemented the classical Min-min algorithm, using the number of prompt tokens and the upper bund of the predicted decode token buckets to calculate the time for finishing each job. Since we have homogenous model instances, this strategy becomes similar to shortest job first.

C.4 Overhead of the Router

For each decision, the router has to perform two additional steps in our approach: (i) inference from DistillBERT for output length bucket and (ii) inference from the neural network being used. The approach can parallelize these modules when the number of requests in the queue is large (and the request being routed has already been processed by the length predictor). (i) takes us 0.01 (on GPU) and 0.8 (on CPU) seconds per batch of size 64 and (ii) takes $< 10^6$ operations (within miliseconds) to process.

C.5 Details of Dataset

From each dataset, we take the subset of prompts that have a maximum prompt length of 1000 tokens.

C.5.1 Prompts. For each task, the prompt is created in the following manner:

Sentiment Analysis (IMDb dataset). For each review in the dataset, we randomly select one of the following sentences and add it to the review:

(1) "Based on this review, judge if the user liked this movie or not?"
(2) "Please identify if the review is positive or negative?"
(3) "Based on this review, should we recommend this movie to other users with similar tastes?"

We add these tasks either at the beginning or at the end of the prompts, again randomly.

QnA (Eli5 Reddit subset). We pick the question in the title as it is and provide it as the prompt to the LLM.

Entity Recognition (WNUT dataset). We add the suffix "Can you identify the <entity> mentioned in the above sentence?" where <entity> is selected ranomdly from "person", "place", and "object".

In context QnA (SQuAD dataset). We add the question as well as the four options of the answer to the prompt and ask the LLM to select the correct option and provide reasoning with it as well.

Translation (Books dataset). We provide the text and add the phrase "Please translate this text into <language>" either at the start or at the end of the text. <language> is selected from the ones provided in the dataset itself.

C.5.2 Task Hints. For providing a hint of the task to the model, we add the phrase "This is a <task> task" at the end of each prompt before providing it to the classifier.

Source	Task	Samples	Average Tokens		Heavy Decode	Accuracy	
			Prompt	Decode		SOTA	Ours
Books Tiedemann [40]	Translation	7351	29.09	61.76	9.18%	4.47%	93.10%
Eli5 (Reddit subset) Fan et al. [11]	QnA	6988	29.83	334.40	58.18%	5.91%	70.36%
IMDb Maas et al. [24]	Sentiment Analysis	6564	211.54	142.53	41.01%	6.81%	79.92%
SQuAD Rajpurkar et al. [34]	In-context QnA	7122	125.16	220.02	47.95%	6.22%	65.27%
WNUT Derczynski et al. [8]	Entity Recognition	3304	26.41	64.10	8.71%	2.76%	95.06%
Total	-	31329	89.03	175.71	35.54%	5.5%	79.15%

Table 3: Dataset and Performance of Output Length Predictor. Average prompt and decode tokens varies across data sources. The second last column indicates the percentage of requests with heavy decodes (≥ 5 seconds estimated completion time) and the last column indicates the accuracy of our decode length predictor model described in subsection 4.1 for each source.

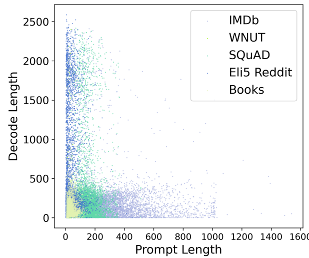
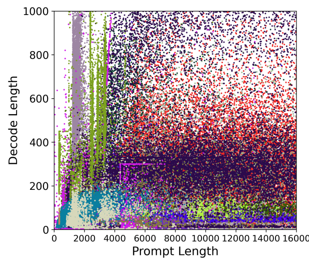
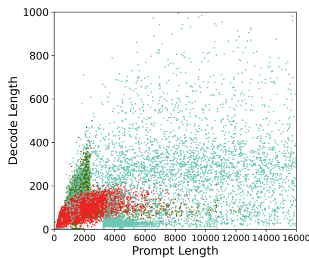


Figure 7: Prompt decode distribution for our dataset with responses generated from Llama 2 7B model.



(a) Distribution of entire trace



(b) Distribution of certain applications

Figure 8: Prompt-decode distribution of the production traffic from Cloud provider X.

C.6 Prompt-Decode Distribution

Figure 7 shows the distribution of prompts and decode tokens across the different datasets we mixed. We can clearly see the different distributions each dataset has. Prompts from Eli-5 Reddit subset are shorter in length and have longer responses than the rest of the dataset, while the IMDb distribution on the other hand has longer prompt lengths and shorter responses. Such a varied distriubtion contributes to the low accuracy of the current SOTA model by Jin et al. [18].

C.7 Training details of the output length predictor

We had a total of 31329 samples in our mixed dataset, from which we had an 80:20 train-test split. We had a train time accuracy of 81% after performing 6 epochs of fine-tuning with the entire training set.

C.8 Task Predictability

We predict the task of a prompt sampled from our dataset described in section 3 using DistillBERT, the same methodology we use to predict their output length bucket as discussed in subsection 4.1. We observe an accuracy of 93.79%.

This allows us to proceed safely with the assumption that we can provide task type as part of the prompt to the output length predictor.

C.9 Licenses

(1) WNUT Dataset: CC-by-4.0
(2) SQuAD dataset: CC-by-SA-4.0
(3) vLLM: Apache-2.0

C.10 Details of RL training

For our experiments, we use 4 LLM model instances to route among. This results in our state space having 27 dimensions (6 for each model instance and 4 for the request queue at the router). In order to bound our state space, we round the estimated capacity available at each model instance and the estimated completion time for the earliest request to two decimal places. We also upper bound the waiting queue length that we provide to the DQN to $4 \times (\texttt{max batch size}) = 4 \times 128 = 512$. We provide the DQN with 3 buckets: 0-256, 256-2048, ≥ 2048.

C.10.1 Q-Learning. Q-Learning yields poor performance for our task due to the size of the state space. If we upper bound the total number of requests that can be present at a model instance to 150

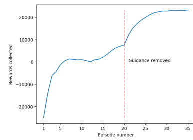
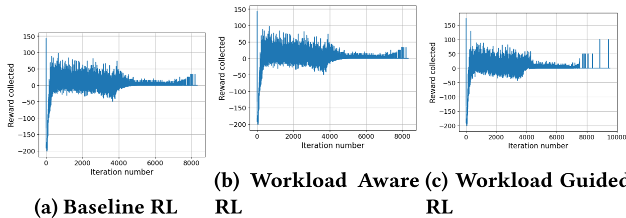


Figure 9: Training reward for workload guided RL



(a) Baseline RL **(b) Workload Aware RL** **(c) Workload Guided RL**

Figure 10: Rewards collected during testing for each strategy

(even though there can be infinitely many) and the prompt and decode length to 4096 (maximum content window of LLama-2 7B), each model instance can be in $150 \times 150 \times 100 \times (4096 \times 150 \times \texttt{grad}_2 \times 100) = 3.0405 \times 10^9$ different states. This would result in a total of $(3.0405 \times 10^9)^4 \times 512 \times 4096 \times 3 \approx 5 \times 10^{44}$. Even though we will never visit most of these states, the possible states that be visited are large enough to make Q-Learning infeasible.

C.10.2 Training Rewards. Figure 9 Shows the rewards collected during training of the RL model. We see that the guidance heuristic helps the agent converge. After episode 20, we no longer use explore with random actions and exploit this knowledge.

C.10.3 Double-DQN. We take a double DQN approach for our RL agent. We set the request completion reward to 60 and train our DQN with a batch size of 512. We use a neural network with layer sizes $(27, 64)$, $(64, 64)$, $(64, 5)$ and ReLU activation function for layers 1 and 2.

Figure 10 shows the rewards collected by each strategy during testing. Requests stop arriving at iteration number 4000, after which, we see the rewards tend to positive values due to the high request completion reward.

C.11 Additional experiments to validate the scalability of proposed framework

To further test the scalability of our approach, we tested our methods with eight model instances. We increased the number of processed requests to 4,000 and the request arrival rate to 40/s to remain consistent with previous experiments. To scale our approach, we needed to increase the parameters in our neural network. Our methods outperformed the Round-Robin approach in this setup as well. On average, Baseline RL, Workload Aware RL, and Workload Guided RL outperformed Round Robin by 5.84%, 6.64%, and 11.62%, respectively.

C.12 Experiments on Real Production Trace from Cloud Provider X

Next, we validate our approach using one hour production trace from Cloud provider X. We use 4000 requests for our experiments, with average prompt length of 5526.64 tokens and average decode length of 112.69 tokens. We do our experiments at 80 requests per second, again using Llama-3.1-8B model. We enable chunking for this experiment with maximum number of batched tokens set to 1024. Round robin takes 1005.31 seconds on average (across 20 random iterations). We see that the advantages of our algorithms are less pronounced when the prompt length becomes much longer than the decode length, with advantages of baseline RL, workload aware RL and workload guided RL reducing to 2.28% (982.38 seconds), 4.39% (961.17 seconds) and 7.84% (926.49 seconds) respectively. This can also be attributed to the lesser number of preemptions happening as the decode length has gotten shorter.

To reduce the overhead of output length prediction, we assume the unavailability of prompt content and only assume the availability of prompt token count. Therefore, for the bucket prediction module, we train a Random Forest which takes the prompt length of the request along with the application name associated with the request. Using the same bucket sizes as before, this module is able to achieve 79% accuracy (while 68.44% of the requests were in bucket 0) due to the predictable nature of production traffic. Figure 8 shows the prompt and decode distribution from the production trace. The prompt and decode distribution of applications from the production trace show distinct trend as shown in Figure 8b which makes the decode length predictable with prompt length and application type.

C.13 Additional Proofs

Reshaping the MDP ($\mathcal{M}$) with heuristic guided RL preserves the value bounds and linearity of the original MDP: 1) If $h(s) \in [0, \frac{1}{1-\gamma}]$, then value function corresponding to the policy, π, $\tilde{V}^{\pi}(s) \in [0, \frac{1}{1-\gamma}]$ for all π and $s \in \mathcal{S}$. 2) If $\mathcal{M}$ is a linear MDP with feature vector $\phi(s, a)$ (i.e. $r(s, a)$ and $\mathbb{E}_{s'|s,a}[g(s')]$ for any g that can be linearly parameterized in $\phi(s, a)$ [6].