Learned Best-Effort LLM Serving

Siddharth Jha [1] **Coleman Hooper** [1] **Xiaoxuan Liu** [1] **Sehoon Kim** [1] **Kurt Keutzer** [1]

Abstract

Many applications must provide low-latency LLM service to users or risk unacceptable user experience. However, over-provisioning resources to serve fluctuating request patterns is often prohibitively expensive. In this work, we present a best-effort serving system that employs deep reinforcement learning to adjust service quality based on the task distribution and system load. Our best-effort system can maintain availability with over $10\times$ higher client request rates, serves above 96% of peak performance $4.1\times$ more often, and serves above 98% of peak performance $2.3\times$ more often than static serving on unpredictable workloads.

1. Introduction

The widespread adoption of large language models (LLMs) in chat agents and virtual assistants has underscored the importance of low-latency responses for user experience. However, the unpredictability of online request patterns presents a challenge for efficient LLM serving. Over-provisioning GPU resources to serve bursty request windows is not economically viable for smaller entities. Furthermore, while downsizing to smaller models presents a cost-effective alternative, it risks compromising output quality if done without careful consideration.

In response to the challenge of delivering low-latency LLM services without additional hardware costs, we introduce a learned best-effort serving framework that dynamically selects models of varying sizes to match client requests, guided by a model routing mechanism as illustrated in Figure 1. Leveraging the lower memory footprint of smaller models, best-effort serving does not inhibit the ability to run the largest desired model in the system, similar to speculative inference (Leviathan et al., 2023) methods.

Our system is designed to handle requests across a spectrum of tasks, balancing between model size for accuracy and meeting deadlines, categorized as hard or soft, to optimize overall performance. Performance is quantified by accuracy for hard deadlines, and decreases as soft deadlines are exceeded. The ultimate objective is to maximize cumulative performance, with "peak performance" considered as the ideal state of meeting all deadlines using the largest model. We show that effective request routing is influenced by the distribution and difficulty of tasks, as well as system load. We employ deep reinforcement learning through the DQN algorithm to manage these dynamics efficiently.

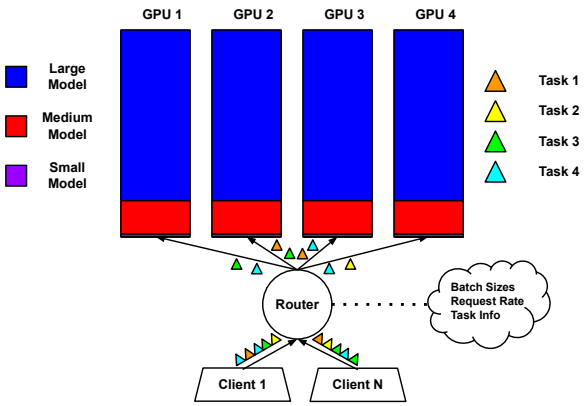

Figure 1: Learned best-effort serving consists of multiple models serving multiple tasks, with a router that keeps track of system load and task information in order to route requests to models.

In summary, learned best-effort serving provides a variety of benefits over traditional static serving methods for low-latency LLM applications:

- **Performance:** It maintains over 96% of peak performance $4.1\times$ more frequently, and exceeds 98% of peak performance $2.3\times$ more frequently than static serving with a large model on unpredictable workloads.
- **Availability:** It meets client deadlines at over $10\times$ higher system load than static serving with a large model. Compared to statically serving a medium sized model, it achieves at least 94% of peak performance $28.21\times$ more often, in addition to higher availability.

[1]UC Berkeley. Correspondence to: Siddharth Jha <sidjha@berkeley.edu>.

Presented in Efficient Systems for Foundation Models workshop in the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

2. Preliminaries

2.1. Efficient LLM Serving

The use of multiple models to serve LLM requests has been explored via speculative inference (Leviathan et al., 2023; Spector & Re, 2023), which uses a small draft model to generate tokens to be verified by a large model. Big Little Decoder (Kim et al., 2023) is a speculative inference technique that allows clients to adjust quality and latency by changing hyper-parameters. However, this requires a hyper-parameter search for every task and does not adjust under load. Dynamically adjusting serving in response to load has been explored though autoscalers. Autoscalers such as Ray (Moritz et al., 2018) dynamically increase GPU instances under load. However, acquiring on-demand GPU instances is expensive and not instantaneous. Model switching has been explored in (Zhang et al., 2020; Eccles et al., 2024). However, (Zhang et al., 2020) only considers CNN models running on CPUs and does not consider the task when selecting a model. Furthermore, (Eccles et al., 2024) necessitates pruned models and can only have one active model at a time.

2.2. Deep Reinforcement Learning

Deep RL is a promising technique for learning to control systems, and it has been successfully applied in a variety of areas such as continuous controls (Brockman et al., 2016) and games (Mnih et al., 2013). There are three core components in any RL problem: states, actions, and rewards. The RL policy aims to maximize the total rewards it sees as it takes actions and transitions between states. Deep Q-learning methods learn a Q-function, represented as a neural network, that map state-action pairs to the expected return of taking the action in the state and then following the policy. After fitting the Q-function of the optimal policy, the Q-function may be used to select actions with the highest expected reward. Popular algorithms in this area include DQN (Mnih et al., 2013), Double Q-learning (Van Hasselt et al., 2016), and PER (Schaul et al., 2015).

3. Best-Effort Serving

3.1. Problem Formulation

We envision a system where multiple clients dispatch requests for LLM inference. These requests are aligned with predefined tasks (e.g. summarization, question answering, etc.) tailored to the application, each tagged with a designated latency requirement. Service requirements are determined at the task granularity, meaning that latency requirements are equivalent for requests belonging to the same task. The system's infrastructure categorizes deadlines into two types: hard and soft. For hard deadlines, the utility of a response is gauged by its accuracy if it meets the request's latency requirement. Conversely, for soft deadlines, a response's utility diminishes proportionally with the extent of delay past the deadline. The overarching objective of the system is to optimize the cumulative utility across all client requests.

Suppose the best-effort system is serving T tasks using M model choices. Let $A \in \mathbb{R}^{T \times M}$ be a matrix so that A_{tm} denotes the client utility (e.g. accuracy) of serving a request from task t using model m. Each client request is tagged with a task and deadline. Given a sequence of requests $(r_n)_{n \in \mathbb{N}}$, let request r_n's task be denoted by a one-hot encoded vector $t_n \in \mathbb{R}^{1 \times T}$ and its latency requirement be denoted as $d_n \in \mathbb{R}$. If we let $m_n \in \mathbb{R}^{1 \times M}$ be a one-hot encoded vector representing the model assigned to request r_n, the goal of the serving system is to optimize

$$\max_{m} \sum_{n \geq 1} w_n t_n A m_n^T \quad \text{subject to} \quad \|m_n\|_1 = 1, \forall n \quad (1)$$

where w_n is dependent on the deadline being satisfied. For applications with hard deadlines, $w_n \in \{0, 1\}$ is binary. When using soft deadlines, $w_n \in [0, 1]$ decreases as the assigned deadline is further violated. The decay function that governs this may be set by application developers to best meet their requirements. Optimizing Equation 1 for online serving is difficult as the system does not have access to the behavior of future requests when making a model selection decision for a present request. Static serving methods assign the same model to each request. In contrast, best-effort serving employs dynamic routing to various models, aiming to leverage the latency-quality trade-off most effectively. This methodology seeks to enhance the system's ability to meet diverse client demands while maximizing overall performance.

3.2. Dynamic Router Agent

To address the challenge of optimizing Equation 1 in online settings, we employ deep reinforcement learning to learn a router agent that dynamically dispatches client requests to models. Since the agent cannot have access to information about future request patterns when making a routing decision, it must rely only the present and past. Thus the agent keeps track of the present batch size at each model and an approximation of the current arrival rate of the client request process using past requests. In sum, the state that the agent conditions on when routing consists of the task (numbered 0 though $T - 1$), the batch size at each model, and the current arrival rate. The action space consists of M options, with each action corresponding to a separate model. The reward for picking an action in a given state is the exact same as in Equation 1. Thus the router agent is optimizing a proxy of

Equation 1. To learn an effective policy for our router agent, we utilize the DQN algorithm with Double Q-learning to prevent over-estimation of Q-values and represent the policy as a 2-layer MLP with hidden size 256. As we do not assume prior information about the production workload, we train the policy by randomly switching between random arrival rates with a uniform task distribution. It is possible for application developers to exploit knowledge about their workloads to change the training distribution to closely match deployment, but our experiments in section 4 show that random training works well in practice. Additionally, as the policy is a small MLP, it can be run in less than a millisecond.

4. Evaluation

4.1. Experiment Setup

We use one set of hyper-parameters, further detailed in Appendix A, for all trained policies. As baselines, we evaluate against static serving with just one model size. When running the baselines, we give all the GPU memory to each model. Models are served with vLLM (Kwon et al., 2023). Graphs with uncertainty regions represent one standard deviation over three trials.

Prior work on model serving (Li et al., 2023; Zhang et al., 2023; Gujarati et al., 2020) uses Microsoft's Azure Function (MAF) traces (Shahrad et al., 2020; Zhang et al., 2021) to model behavior of clients in a serving system. The MAF1 trace (Shahrad et al., 2020) consists of stable and steady request periods. On the other hand, the MAF2 trace (Zhang et al., 2021) has much more unpredictable client behavior and the arrival rates rapidly change. Based off of these observations, we evaluate our system on three types of synthetic workloads that capture a wide range of client behavior. One workload is stable, while the other two are unpredictable. The synthetic workloads are generated in similar ways to (Yu et al., 2022; Zhang et al., 2023), which use Poisson processes and Markov-modulated Poisson processes. We list further details about the three synthetic workloads and arrival rate estimation in Appendix B.

4.2. System Setup

To evaluate our routing policy, we consider a serving system with 4 GPUs. Each GPU contains an instance of OPT-125M, OPT-1.3B, and OPT-6.7B and there are 4 tasks in the serving system. Thus the serving system is equivalent to the one shown in Figure 1. Smaller models can fit in device memory with the large model because the memory required for both their model parameters, key-value cache, and activations is significantly smaller than those of the large model. Specifically, we give 5% of GPU memory to OPT-125M, 20% of GPU memory to OPT-1.3B, and the rest to OPT-6.7B. When the router chooses a model size for the request, we automatically load balance by sending to the replica with the smallest batch size for the model. We set the latency guarantee to be 40 milliseconds/token. In Appendix D, we show the results of experiments when assigning different latency deadlines to tasks. Additionally we use zero-shot HellaSwag (Zellers et al., 2019), COPA (Roemmele et al., 2011), PIQA (Bisk et al., 2020), and OpenBookQA (Mihaylov et al., 2018) as the four tasks in the system. We use each model's average accuracy on each task as a measure of its quality. For each task we normalize the accuracy of each model to OPT-6.7B's accuracy to get the rewards shown in Table 1. We train the policy for 1.2 million iterations using hard deadlines.

Table 1: Rewards for tasks served in the system.

TASK	OPT-125M	OPT-1.3B	OPT-6.7B
HELLASWAG	0.45	0.78	1.00
COPA	0.80	0.95	1.00
PIQA	0.82	0.96	1.00
OPENBOOKQA	0.70	0.94	1.00

4.3. Stable Workload

4.3.1. HARD DEADLINES

For the stable workload, we vary the arrival rate of the arrival Poisson process from 0.25 to 48 requests per second and serve for 40 seconds. We show the results with hard deadlines in Figure 2. As baselines, we show the performance when only serving one of OPT-6.7B, OPT-1.3B, or OPT-125M.

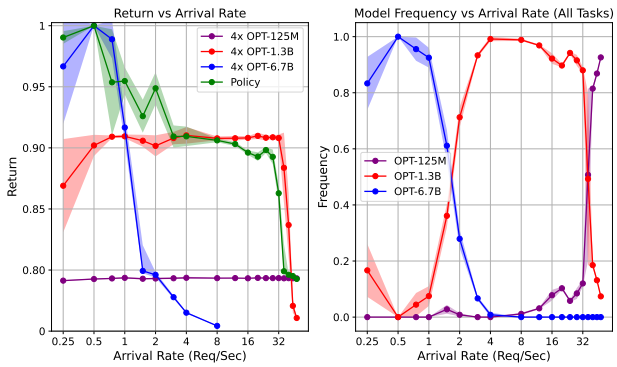

Figure 2: The left figure shows the performance with hard deadlines. The right figure shows the distribution of model selection from the policy.

As Figure 2 shows, in typical systems that serve all requests to OPT-6.7B, the performance is near the peak possible performance at low arrival rates. However, once the arrival rate

increases past a threshold (2 requests per second), many latency deadlines are missed and performance sharply declines. While OPT-1.3B can serve requests at much higher arrival rates, its quality cannot match OPT-6.7B even when the arrival rate is low. Additionally, there is also a point at which OPT-1.3B cannot keep up with client requests. Serving only with OPT-125M leads to significant performance degradation at all but extremely high arrival rates.

In contrast, the policy dynamically adjusts which model to send requests to. When the arrival rate is low, the policy primarily sends to OPT-6.7B and achieves similar performance. However, as the arrival rate increases, the policy correctly routes more requests to OPT-1.3B and eventually even OPT-125M at the extreme end. Therefore the policy allows the system to remain available for over 10x faster arrival rates than just using OPT-6.7B while still providing equal quality to OPT-6.7B at low arrival rates. Furthermore we notice that there are regions where the policy even performs better than just taking the maximum of each of the baseline's curves in Figure 2 as it is able to multiplex between models at a given arrival rate.

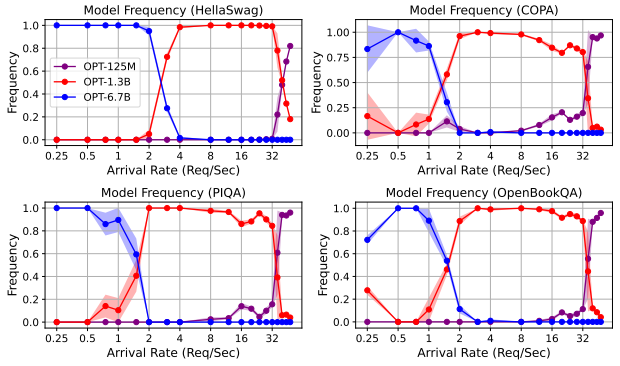

Figure 3: Model selection frequency for each individual task with hard deadlines.

We now examine how the routing varies for different tasks, as shown in Figure 3. We see that the policy sends HellaSwag requests to OPT-6.7B much more often than the other three tasks. Taking a look at Table 1, we see that OPT-125M and OPT-1.3B have a significant quality gap compared to OPT-6.7B for HellaSwag. This quality gap is much larger than the gap between models on COPA, PIQA, and OpenBookQA. Therefore the policy appropriately learns to prioritize sending HellaSwag to the large model when possible. Furthermore, even when the arrival rate is higher, HellaSwag is sent to OPT-1.3B more often than the other three tasks, which are more frequently sent to OPT-125M. Thus the router learns a complex relationship not only depending on the task's quality across models in isolation, but with respect to the quality of other tasks in the system and their distribution.

4.3.2. Soft Deadlines

We pick a specific soft deadline decay function and fine-tune the policy from the policy trained with the hard deadlines. It is also possible to train the soft policy from scratch. When fine-tuning, we adjust the reward function to be soft and train for an additional 685,000 iterations. Specifically, this soft deadline decays the reward by 1% for every millisecond past the deadline as long as the violation is less than 10% of the specified deadline. However, once the acceptable latency is violated by more than 10%, the client does not value the response and the reward is zero. We show the results in Figure 4. We see that the policy outperforms the baselines and sends more requests to larger models when using this soft deadline than when using hard deadlines. Compared to the hard policy's performance in Figure 2, we see that the soft policy more closely follows OPT-1.3B's performance before switching to OPT-125M's performance. With hard deadlines, the policy takes a slightly more conservative approach in this regime and sends a small set of requests to the small model in order to prevent missing deadlines, creating a small performance gap between the policy and OPT-1.3B before the policy switches to OPT-125M's performance. With soft deadlines, the policy is less conservative and is able to almost exactly match the performance of OPT-1.3B at these arrival rates.

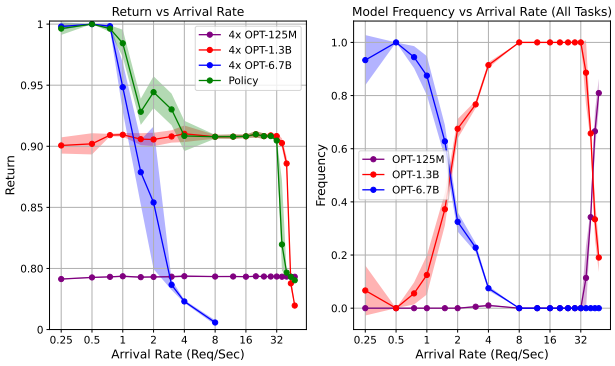

Figure 4: The left figure shows the performance with soft deadlines. The right figure shows the distribution of model selection from the policy.

We show how tasks are routed to models when using this soft deadline in Figure 5 and observe similar trends to Figure 3. When measuring the usage of OPT-6.7B via a Riemann sum of the selection distribution, we see that HellaSwag's OPT-6.7B usage increases by 52% with soft deadlines compared to hard deadlines. In contrast, PIQA, COPA, and OpenBookQA's OPT-6.7B usage increases by just 9%, 7%, and 3%, respectively. Thus the policy is able to exploit the leniency given by the soft deadline to reap the large gains in quality by sending HellaSwag to OPT-6.7B instead of OPT-1.3B.

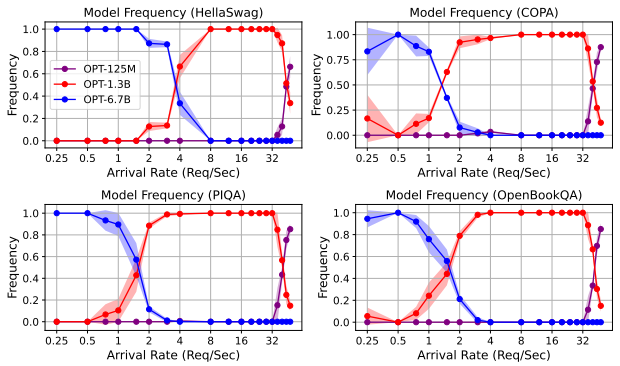

Figure 5: Model selection frequency for each individual task with soft deadlines.

4.4. Unpredictable Workloads

We evaluate on an unpredictable workload using hard deadlines with large bursts. Figure 6 shows the performance of the routing policy as well as the baselines, in addition to the changing arrival rate for the unpredictable workload. We show both the running average of performance across all served requests and the running average of the performance across the last 20 requests. The serving system that only uses OPT-6.7B fails to meet latency deadlines during many of the bursts and thus its performance is highly variable. Even though OPT-6.7B has more windowed averages at peak performance, the policy is able to perform at near-peak performance significantly more often. We quantify this in Table 2.

Figure 6: Running total and windowed average over the last 20 requests of performance on the unpredictable workload. The arrival rate at each step is also shown.

As shown in Table 2, Compared to the policy, OPT-6.7B is able to achieve more windowed averages with peak performance. However, when analyzing the number of windows

Table 2: Number of request windows of size 20 that meet average performance thresholds on the unpredictable workload.

THRESHOLD	POLICY	OPT-6.7B	OPT-1.3B
$= 1.00$	142	**307**	0
≥ 0.99	**470**	307	0
≥ 0.98	**713**	307	0
≥ 0.96	**1264**	307	0
≥ 0.94	**1723**	625	154

which meet high performance thresholds such as 0.99, 0.98, 0.96, and 0.94, the policy achieves more such windows than OPT-6.7B and OPT-1.3B. For example, it achieves $1.53\times$ more windows past 99% performance, $2.32\times$ more windows past 98% performance, and $4.11\times$ more windows past 96% performance compared to OPT-6.7B. Additionally, it achieves at least 94% of peak performance $2.75\times$ more often than OPT-6.7B and $11.18\times$ more often than OPT-1.3B. This shows that the policy is able to correctly balance between OPT-6.7B, OPT-1.3B, and OPT-125M, even while faced with an unpredictable workload. We see similar results on another unpredictable workload, detailed in Appendix C.

5. Discussion

Viewed through another lens, learned routing allows higher quality during low system load than statically serving a smaller model to handle periods of high loads. There are also a wide range of environments in which best-effort serving is applicable. By formulating model serving as a reinforcement learning problem, application developers have the flexibility to adapt the reward function in order to meet their application requirements. For example, they may choose to up-weight the reward on prioritized requests coming from paid users. In Appendix. D, E, and F, we show that our learned router can support different deadlines per task, be robust in the face of shifts in the task distribution, and facilitate serving with less compute resources.

6. Conclusion

Rather than serving LLMs at a fixed model size, we propose a best-effort serving paradigm with a learned router that maximizes holistic performance, which jointly captures quality and latency. Best-effort serving with dynamic routing is an efficient approach for developers looking to scale their latency-sensitive applications. We train our router using deep reinforcement learning methods and outperform static serving baselines in a variety of workloads. We imagine best-effort serving with dynamic routing to be a cheap and efficient paradigm for latency-sensitive applications.

References

Bisk, Y., Zellers, R., Bras, R. L., Gao, J., and Choi, Y. Piqa: Reasoning about physical commonsense in natural language. In *Thirty-Fourth AAAI Conference on Artificial Intelligence*, 2020.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Eccles, B. J., Rodgers, P., Kilpatrick, P., Spence, I., and Varghese, B. Dnnshifter: An efficient dnn pruning system for edge computing. *Future Generation Computer Systems*, 152:43–54, 2024.

Gujarati, A., Karimi, R., Alzayat, S., Hao, W., Kaufmann, A., Vigfusson, Y., and Mace, J. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 443–462, 2020.

Kim, S., Mangalam, K., Malik, J., Mahoney, M. W., Gholami, A., and Keutzer, K. Big little transformer decoder. *arXiv preprint arXiv:2302.07863*, 2023.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Leviathan, Y., Kalman, M., and Matias, Y. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pp. 19274–19286. PMLR, 2023.

Li, Z., Zheng, L., Zhong, Y., Liu, V., Sheng, Y., Jin, X., Huang, Y., Chen, Z., Zhang, H., Gonzalez, J. E., et al. Alpaserve: Statistical multiplexing with model parallelism for deep learning serving. *arXiv preprint arXiv:2302.11665*, 2023.

Mihaylov, T., Clark, P., Khot, T., and Sabharwal, A. Can a suit of armor conduct electricity? a new dataset for open book question answering. In *EMNLP*, 2018.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pp. 561–577, 2018.

Roemmele, M., Bejan, C. A., and Gordon, A. S. Choice of plausible alternatives: An evaluation of commonsense causal reasoning. In *2011 AAAI Spring Symposium Series*, 2011.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

Shahrad, M., Fonseca, R., Goiri, I., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., and Bianchini, R. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX annual technical conference (USENIX ATC 20)*, pp. 205–218, 2020.

Spector, B. and Re, C. Accelerating llm inference with staged speculative decoding. *arXiv preprint arXiv:2308.04623*, 2023.

Van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, 2022.

Zellers, R., Holtzman, A., Bisk, Y., Farhadi, A., and Choi, Y. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*, 2019.

Zhang, H., Tang, Y., Khandelwal, A., and Stoica, I. {SHEPHERD}: Serving {DNNs} in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pp. 787–808, 2023.

Zhang, J., Elnikety, S., Zarar, S., Gupta, A., and Garg, S. {Model-Switching}: Dealing with fluctuating workloads in {Machine-Learning-as-a-Service} systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

Zhang, Y., Goiri, Í., Chaudhry, G. I., Fonseca, R., Elnikety, S., Delimitrou, C., and Bianchini, R. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pp. 724–739, 2021.

A. Training Details

We use a discount rate of 0.99, a learning rate of 0.0001, and a batch size of 1024. In our Double Q-learning implementation, the target network is updated every 500 iterations. For exploration, we use an epsilon-greedy strategy. We use an NVIDIA TITAN RTX for GPUs and an Intel Xeon Gold 6126 processor as our CPU.

B. Workload Details

There are three workloads we use - 1 stable and 2 unpredictable. The first represents a stable workload in which client requests arrive in the system as a Poisson process with a fixed rate for a set period of time. The second and third workload represent unpredictable workloads in which the arrival rate of requests rapidly switches due to an underlying stochastic process that controls the arrival rate and its duration. The difference between the second and third workload is that the second workload assumes that the system spends the same amount of time (in expectation) in each arrival rate, while the third workload assumes that the system serves the same amount of requests (in expectation) in each arrival rate before switching. On the unpredictable workloads, we estimate the arrival rate by taking a running average of the last 5 arrivals. Since a near-zero variance estimate of the arrival rate may be obtained on the stable workloads, we give the agent the true arrival rate.

For the first unpredictable workload, we randomly vary the arrival rate and the number of requests served at that arrival rate before switching to the next arrival rate. With 90% probability, we randomly pick an arrival rate between 0.25 and 2. With 8% probability, we randomly pick an arrival rate between 2 and 40. With the remaining 2% probability, we randomly pick an arrival rate between 40 and 48. The number of requests served at the arrival rate is a geometric random variable with mean 20 $\times$ arrival rate. There are 10,000 requests in total.

In the second unpredictable workload, the arrival rate randomly fluctuates between 1 request per second and 48 requests per second. The first unpredictable workload assumed that the expected time spent in each arrival rate is the same. With the new workload, we assume that the number of expected requests seen in each arrival rate is the same, meaning large arrival rates occupy less real time in the system than small arrival rates. We use a geometric random variable with mean 500 to determine the number of requests to serve at an arrival rate before picking the next arrival rate. There are 10,000 requests in total.

C. Additional Unpredictable Workload Experiment

Figure 7: Running total and windowed average over the last 20 requests of performance on the second unpredictable workload. The arrival rate at each step is also shown.

THRESHOLD	POLICY	OPT-6.7B	OPT-1.3B
$= 1.00$	447	**1378**	0
≥ 0.99	**1977**	1378	0
≥ 0.98	**3121**	1378	0
≥ 0.96	**4141**	1378	0
≥ 0.94	**4740**	2810	168

Table 3: Number of request windows of size 20 that meet average performance thresholds on the second unpredictable workload.

We also show performance on the second unpredictable workload. The results are shown in Figure 7. As the results in Figure 7 show, the policy outperforms both baselines of OPT-1.3B and OPT-6.7B and is able to adapt to the large changes in arrival rate. Additionally, as shown in Table 3, the policy has significantly more windows which meet high performance

thresholds. It achieves $1.43\times$ more windows past 99% performance, $2.26\times$ more windows past 98% performance, and $3\times$ more windows past 96% performance compared to OPT-6.7B. Additionally, it achieves at least 94% of peak performance $1.68\times$ more often than OPT-6.7B and $28.21\times$ more often than OPT-1.3B.

D. Different Deadlines Experiment

Certain applications may want different deadlines to be associated with different tasks. To see if the policy can handle this setting, we double the deadline for OpenBookQA and tighten the deadline for COPA by 20%. We train the policy for 800,000 iterations using hard deadlines and show both the performance and model selection in Figure 8 and Figure 9. We see that the policy is able to learn with different deadlines and outperforms the static serving baselines as before. Additionally, compared to the task-specific selection distribution in Figure 3, the policy sends significantly more OpenBookQA requests to OPT-6.7B, as the deadline is now looser. Compared to the other tasks in the system, the policy waits for a higher arrival rate before it switches sending OpenBookQA requests to OPT-125M instead of OPT-1.3B. Similarly, the policy significantly reduces OPT-6.7B usage for COPA in favor of OPT-1.3B due to the tighter deadlines.

Figure 8: Running total and windowed average over the last 20 requests of performance on the second unpredictable workload. The arrival rate at each step is also shown.

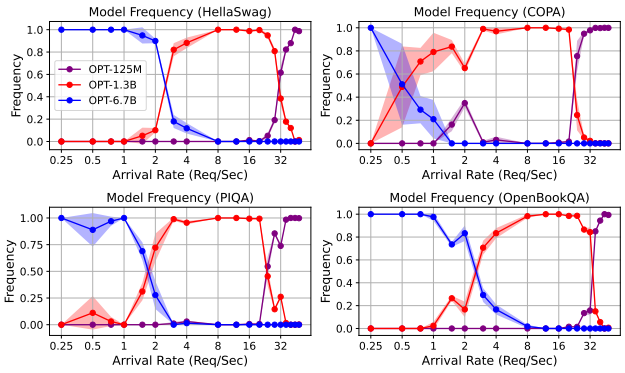

Figure 9: Model selection frequency for each individual task with loose OpenBookQA and tight COPA deadlines.

E. Task Distribution Shift Experiment

When training the policy we assume that tasks are picked uniformly at random in the application's workload. For some classes of applications, it is possible that the distribution of tasks changes after the policy has been deployed. It is important that the policy still perform well when the task distribution shifts. We show the performance of our policy in the hard deadline setting while drastically changing the task distribution and without performing any additional training iterations. Specifically, we evaluate performance when the serving system is only receiving requests from one class of tasks at a time. The results on the stable workload are shown in Figure 10. Even though the task distribution is drastically different than the training distribution, the policy is still able to achieve better performance than OPT-6.7B OPT-1.3B, and OPT-125M.

We notice that when serving only HellaSwag, the policy closely follows OPT-6.7B's performance at low arrival rates. In contrast, when only serving COPA, PIQA, or OpenBookQA, the policy is able to outperform OPT-6.7B at these low arrival rates. When analyzing the policy's model selection distribution for each of the tasks in Figure 11, we see that this is because the policy favors OPT-6.7B on HellaSwag more than the other tasks. Although this was optimal when tasks were picked uniformly at random, it is slightly less optimal now because the policy may benefit from sending some HellaSwag requests to OPT-1.3B due to randomness in the arrival process. For COPA, PIQA, and OpenBookQA, the policy is able to multiplex better with OPT-6.7B and OPT-1.3B and thus is able to beat OPT-6.7B at lower arrival rates. The out-of-distribution performance outperforms the baselines and highlights the policy's robustness.

We also evaluate the soft policy's resistance to shifts in the task distribution on an unpredictable workload. The workload

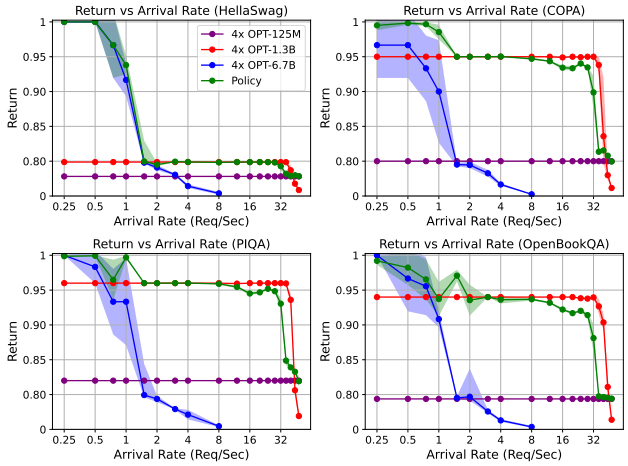

Figure 10: Performance for each of the out-of-distribution workloads, corresponding to sending only one task to the serving system.

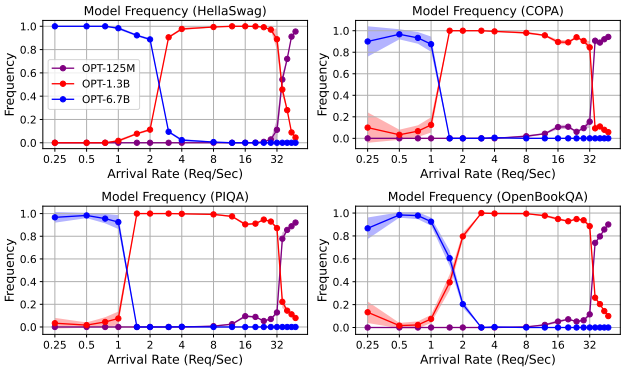

Figure 11: Model selection frequency for each of the out-of-distribution workloads, corresponding to sending only one task to the serving system.

is the first unpredictable workload but requests are chosen uniformly at random between only HellaSwag and COPA. As shown in Figure 12, the policy is still able to outperform the baselines.

Figure 12: Running total and windowed average over the last 20 requests of performance on the first unpredictable workload while only serving HellaSwag and COPA with soft deadlines. The arrival rate at each step is also shown.

THRESHOLD	POLICY	OPT-6.7B	OPT-1.3B
$= 1.00$	170	**369**	0
≥ 0.99	551	**696**	0
≥ 0.98	**812**	727	0
≥ 0.96	**1163**	727	0
≥ 0.94	**1497**	1302	0

Table 4: Request windows of size 20 that meet average performance thresholds.

F. Hardware Utility Experiment

Our learned router leads to increased performance per hardware unit, which we call hardware utility. To demonstrate this, we compare the performance of running OPT-6.7B replicated on 8 GPUs against running our policy on 4 GPUs. We show the results in Figure 13 on the stable workload using both hard and soft deadlines. When showing the performance, we normalize by the number of GPUs to capture the hardware utility. As Figure 13 shows, the policy running on 4 GPUs significantly outperforms both the 4 and 8 GPU OPT-6.7B system on a per-GPU basis. Additionally, we compare the policy's hardware utility against the 8 GPU system on the first unpredictable workload and show the results in Figure 14. The policy achieves higher hardware utility than the 8 GPU system 97.03% of the time. On average, the policy's hardware

utility is $3.94\times$ higher than the 8 GPU system. Additionally, the policy serves past 90% of peak performance $1.51\times$ more often than the 8 GPU OPT-6.7B system, when running on just 4 GPUs. Since reserving many GPU instances and having the budget to pay for them are both difficult tasks, these results highlight an important advantage for best-effort serving.

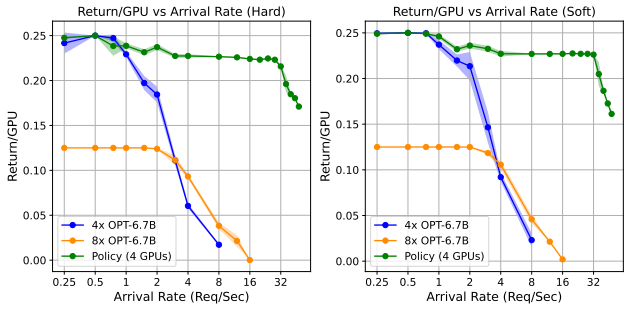

Figure 13: Hardware utility of the policy running on 4 GPUs compared to an OPT-6.7B system running on 8 GPUs. Both hard deadline (left) and soft deadline (right) performance is shown.

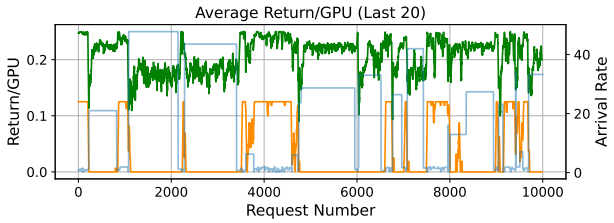

Figure 14: Windowed average over the last 20 requests of the hardware utility of the policy running on 4 GPUs compared to an OPT-6.7B system running on 8 GPUs, on the first unpredictable workload.