Quality-of-Service Aware LLM Serving

Siddharth Jha* [1] **Rithvik Chuppala*** [1]

Abstract

Many applications must provide low-latency large language model (LLM) service to users or risk unacceptable user experience. However, over-provisioning resources to serve models is often prohibitively expensive. In this work, we identify various classes of LLM requests, each with different quality-of-service requirements. To best meet the requirements of all traffic classes, we design and implement scheduling algorithms for the QUIC network protocol. Moreover, we present a best-effort serving system that employs deep Q-learning to adjust service quality based on task distribution and system load. We train the Q-learning agent to optimize latency and model serving quality of higher priority classes while achieving fairness and best-effort response quality for lower priority requests. Our network schedulers show better request latency and completion rate performance compared to the standard QUIC protocol, as well as TLS/TCP. Overall, our end-to-end model-serving system effectively caters to QoS differentiation between requests and outperforms current model-serving standards in per-token latency and response quality metrics.

1. Introduction

Applications in the last decade have evolved from using machine learning in background functions such as data analytics and monitoring to being at the forefront of user experience. Over the past couple of years, many applications have adopted large language models (LLMs) to provide users with both custom and interactive experiences. The need for latency guarantees is critical for such applications as services cannot simply hang and become unavailable to users. LLM systems are faced with the challenge of serving a wide range of user demands, such as varying context lengths and request arrival rates, while simultaneously meeting application QoS requirements. This necessitates an architectural tradeoff as the simple solution of over-provisioning resources to serve dynamic application needs is prohibitively expensive for small businesses and independent developers.

Instead, we propose a best-effort system that dynamically switches between models of different latency and quality. In this best-effort setting, the serving system must serve at the highest possible quality while maintaining availability. Simply serving at the smallest model's quality all the time will be undesirable for a user, even though availability would be high. In this work, we show that routing queries to models is dependent on the set of tasks, the distribution of those tasks, and the load on the system. In order to learn a router that efficiently routes client requests to LLMs while meeting latency guarantees, we utilize deep reinforcement learning (RL) techniques with minimal hyper-parameter tuning.

State-of-the-art model serving systems utilize REST API/gRPC endpoints over base HTTP/2 (TCP/TLS) network protocols (Agarwal et al., 2023) (vLLM Team, 2023) (RayTeam, 2023). However, model-serving architectures are agnostic to application-specific network patterns and do not cater to QoS differentiation between requests. While most active research explores major bottlenecks in the speed of model inference, we find that effective network scheduling not only allows us to cater to different traffic classes but also meet per-token latency deadlines during periods of high request load and network resource contention. For example, one area of ongoing exploration is the use of LLMs in synthesizing large text corpora, such as collections of legal documents (Bornstein & Radovanovic, 2023). Common uses of the widely popular LLM platform, ChatGPT, involve providing the LLM with contextual information followed by a short question prompt (Raf, 2023). The former potentially requires higher model qualities to accurately synthesize domain-specific technical language, whereas the latter needs low latency of response to provide a fluid and interactive user experience.

To meet request SLOs, we leverage one of the key design components of the QUIC network protocol - multiplexed connection streams. This allows us to send different classes of data in separate streams, reducing multi-connection overheads. To efficiently manage these streams and meet the variety of application requirements detailed above, we propose the implementation of a scheduling abstraction in open-

*Equal contribution [1]UC Berkeley.

source QUIC. By considering LLM traffic at both the network level and model-serving level, we are able to schedule and serve requests at low latencies even in the presence of wide fluctuations in client behavior.

In summary, we make the following contributions in this work:

1. We implement QUIC stream schedulers for client requests and responses to meet the QoS requirements of various traffic classes
2. We train and employ an RL router agent to send client queries to appropriate models in order to maximize response quality while meeting user-defined latency guarantees
3. We design and implement an end-to-end model-serving framework utilizing both network stream scheduling as well as dynamic model selection
4. We evaluate our system, analyzing LLM application requirements, and running request loads to benchmark against existing model-serving techniques.

2. Background and Related Work

2.1. Large Language Models

LLMs have emerged as a powerful service for modern applications. There is a wide spectrum of LLMs which forms a trade-off of quality and latency. Larger models with more parameters can serve client requests at a higher quality but incur higher latency. There is also a wide spectrum of tasks that can be served using LLMs. Such tasks include summarization (Hermann et al., 2015; Narayan et al., 2018), translation (Cettolo et al., 2017), question answering (Rajpurkar et al., 2016), etc. Prior work on LLM serving (Li et al., 2023; Zhang et al., 2023; Gujarati et al., 2020) assumes that client requests are bound to a specific model. Our best-effort approach relaxes this, allowing for increased scalability. 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.

2.2. Deep Reinforcement Learning

Deep RL is a promising technique for learning to control systems and 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. Given the state, the RL policy chooses an action, which gives it a reward for that action and transitions the environment to the next state. The goal of RL algorithms is to maximize the total rewards seen by the policy as it takes actions and transitions to different 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).

2.3. QUIC

The QUIC network protocol (standardized in IETF RFC 9000, 9001, 9002) is a transport-level protocol built on UDP and offers endpoint-to-endpoint uni- or bi-directional connections, reliability semantics, congestion control mechanisms, encryption/security via TLS, low-latency connection establishment, and notably, stream multiplexing (Langley et al., 2017) (Iyengar & Thomson, 2021) (Thomson & Turner, 2021) (Iyengar & Swett, 2021). Currently, QUIC is primarily used in the HTTP/3 stack for web applications as a replacement for TCP/TLS in HTTP/2 (Bishop, 2022). However, its properties make it a compelling network protocol in several other use cases.

One of the key design components of the QUIC protocol is the use of time-multiplexed streams in a single point-to-point connection. While HTTP/2 (built over TLS/TCP) attempts intra-connection multiplexing, it suffers from head-of-line blocking, where all streams are blocked if just a single stream experiences packet loss (Langley et al., 2017).

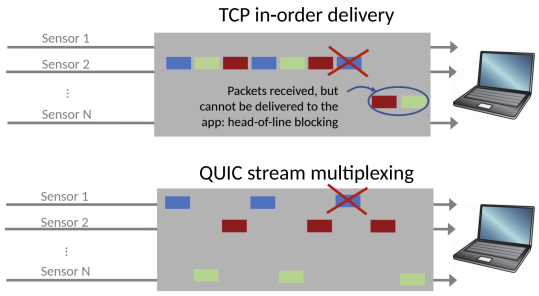

Figure 1. Stream multiplexing in TCP (which suffers from head-of-line blocking) vs QUIC. Credits to (Chiariotti et al., 2021).

QUIC resolves this through flow multiplexing and retransmission semantics at a stream-level granularity. As such, packets in a stream do not block or wait on packets of another stream in the context of loss recovery.

Priority for stream multiplexing was first introduced to HTTP/2 in RFC 7540 (Belshe et al., 2015). However, it was subsequently removed in HTTP/3 (Bishop, 2022) and

the revised HTTP/2 (Thomson & Benfield, 2022) due to high complexity and low utilization of the feature. As an artifact of the old HTTP/2 priority scheme, the QUIC RFC recommends that implementations offer ways to indicate the relative priority of streams (Iyengar & Thomson, 2021); however, a wide number of open-source implementations do not provide this feature.

2.4. Stream Scheduling

Despite a stream multiplexed design, the QUIC RFC does not define or suggest stream scheduling behaviors. Consequently, the vast majority of open-source QUIC implementations use either a default First-Come-First-Serve (FCFS) or Round Robin (RR) scheduler (Kutter & Jaeger, 2022). These implementations do not provide rich scheduling abstractions beyond the default. Literature exploring stream scheduling in QUIC is also quite limited. One work proposed an abstraction to allow applications to effectively map information flows to QUIC streams, focusing on the ability to define correlated data flows (Chiariotti et al., 2021). Another work implements the MPEG-DASH protocol over QUIC, involving scheduling only within the context of the DASH protocol (Cui et al., 2022). Fernandez et al. explore modifications of the QUIC stream scheduler to provide QoS and latency guarantees to UAV video and control flows (Fernández et al., 2023). They focus on priority scheduling, which some open-source implementations already provide upon RFC recommendation. To the best of our knowledge, there are no existing implementations of non-priority-based scheduling algorithms, nor intricate QoS/multi-level scheduling paradigms in open-source QUIC.

3. Design and Architecture

Interactive applications should aim to query as large of an LLM as possible while still meeting an acceptable deadline requirement for their user. However, just using one model type (e.g. OPT-175B) will lead to unacceptably high latencies and impossible-to-meet deadlines during periods of high request load to the inference system. In order to cope with the increasing demands, LLM serving systems need a methodology to schedule all requests by deadline, priority, and various other application-specific constraints, as well as an option to provide smaller models at a small cost of output fidelity.

We use smaller model sizes (e.g. OPT-125M and OPT-6.7B) along with effective request stream scheduling during periods of high request load so users can achieve their desired application latency requirements while receiving acceptable quality from the LLM service. While the QUIC network protocol does not inherently support stream scheduling, the protocol packet framer can be modified to fit various scheduling paradigms. First, we choose an appropriate network stream

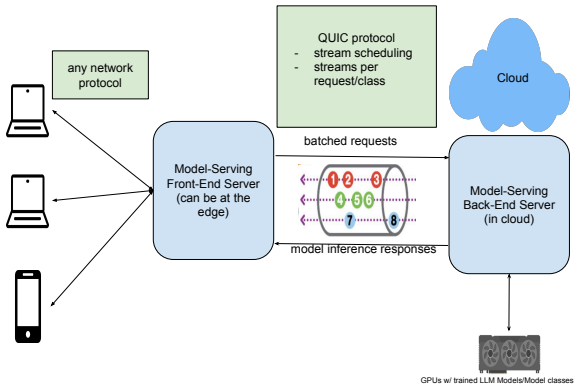

Figure 2. Architecture Diagram of the system. Front-end servers receive client requests, initiate QUIC streams with the model-serving back-end, and schedule request-response flows. The back-end model serving environment is detailed in Figure 3.

scheduling algorithm, given user application requirements. Then, we leverage DQN, a deep Q-learning algorithm, to train a policy that determines what model to route a query to, given the task and system load. During periods of low arrival rates, we expect that a computationally efficient scheduling algorithm, such as RR or FCFS, will be able to adequately serve our requirements and we predict that our policy will route to the largest LLM with no quality degradation. As this is the expected environment most of the time, our system should rarely degrade quality. We optimize for tail request loads by scheduling streams to meet application requirements (using Earliest Deadline First (EDF), Absolute Priority (ABS), and novel application-specific MultiLevel (ML) scheduling procedures) and routing queries to small models.

Overall, the system architecture can be modeled in Figure 2. Client LLM queries reach a group of front-end servers over any network protocol or medium. The front-end servers, which can either be at the edge or in a cloud-like environment, batch requests and initiate request-response streams to the model-serving back-end server. A detailed view of the model-serving environment is shown in Figure 3. In LLM model inference, increased batch sizes result in increased model throughput, providing benefits to batch several independent requests together (Kwon et al., 2023). The batching process smooths out requests and provides consistent behavior on top of which scheduling can be introduced. The front-end servers utilize an active QUIC connection to the back-end, initiating a bi-directional QUIC stream for each request/class of requests. The request and response traffic are assigned to a stream, each which is scheduled according to its application traffic class. Each request is consumed

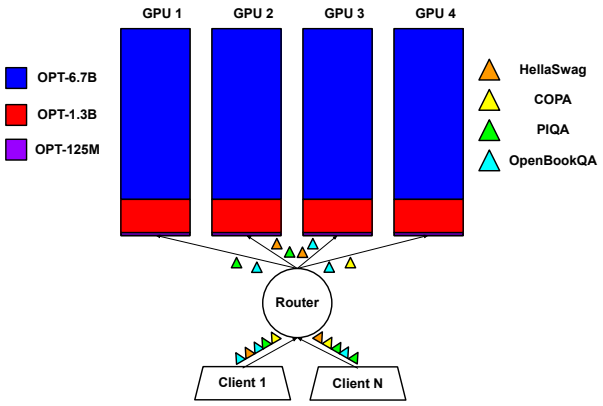

Figure 3. The model serving environment consists of OPT-6.7B, OPT-1.3B, and OPT-125M replicated across 4 GPUs. The system serves HellaSwag, COPA, PIQA, and OpenBookQA.

by the model serving environment, which uses the DQN agent to select a model along with a GPU to generate an output. The RL agent analyzes the query and the load at each of the models and GPUs and determines the best model to route the query in order to maximize quality and meet latency deadlines. The model response is sent back to the corresponding front-end via the same QUIC stream, scheduled with the same semantics. Finally, these responses are delivered back to the initiating clients, completing the flow.

We do not evaluate the setting where there are multiple model-serving environments, since our work focuses on optimizing network utility within a single endpoint-to-endpoint connection and GPU utility within a single machine; however, the proposed architecture will scale accordingly. In the above scenario, the front-end servers can each select among a group of back-end servers, employing a load-balancing framework to evenly spread the request load among the group. Since QUIC streams are bi-directional, the model-serving environment simply utilizes the same incoming stream to send a response back to the appropriate front-end server.

4. Implementation

4.1. QUIC Schedulers

The open-source QUIC implementation we decided to use is `quic-go`. `quic-go` (Clemente & Seemann, 2023) is one of the original canonical implementations that is up-to-date with all current QUIC and HTTP/3 RFCs. It is actively maintained, has shown to be one of the best-performing open source implementations (Crochet et al., 2021), and is the most well-implemented (Seemann, 2023).

The default scheduling procedure in `quic-go` is Round Robin (RR). It cycles through each active stream ("active" is defined as a stream with pending data to be sent), filling up a fixed-size packet with the current stream's data before moving to the next stream and doing the same.

We extend the default RR scheduler with the following four scheduling algorithms:

1. First-Come-First-Serve (FCFS): Streams are processed in order of their arrival, with all of the data of the first stream sent before processing the data of the next stream. We use this as a benchmark, similar to RR.

2. Earliest Deadline First (EDF): Streams are each assigned a deadline by the application and are processed in chronological order of their deadline. If a stream's deadline has passed before it has finished sending all its data, it is discarded and removed from the active stream queue.

3. Absolute Priority (ABS): Streams are each assigned a priority level by the application and are processed in order of priority (highest to lowest), with streams of the same priority served FCFS.

4. MultiLevel (ML): Streams are bifurcated into a lower and higher priority level. The higher priority streams are always served before the lower priority ones. The high priority streams are each assigned a deadline by the application and are served on an EDF basis whereas the lower level streams are served in a fair round robin manner.

In addition, we have built a scheduling abstraction layer that provides applications the functionality to select which of the stream schedulers to utilize, along with an easy way to pass in necessary information for each stream (priority for ABS, deadline for EDF and ML, etc) at stream-creation time. In our experimentation, we have enforced the same scheduler at both the client and server ends (front-end server and back-end server, respectively) but this is not necessary - connection endpoints may inter-operate between different scheduling schemes without any problem.

Stream management in `quic-go` is implemented by the `framer` interface. The interface contains a *streamQueue*, which keeps track of a list of active streams. The framer interacts with scheduling in two main ways: first when an active stream is created and added to the *streamQueue*, and second when the active stream queue is processed to send packets. In default RR, a stream gets added to the *streamQueue* at the end of the queue. While there are active streams in the *streamQueue*, the `framer` pops a stream from the front of the queue and fills up-to a packet of data from the popped stream. If the stream still has data to send,

the stream is appended to the back of the queue; if not, it is removed.

First, we modified the behavior when a stream is initially added to the *streamQueue*. In ABS, the stream is added to the *streamQueue* in descending priority order (i.e. streams with the highest priority values are added to the front of the queue). Similarly, in EDF, the stream is added to the *streamQueue* in ascending deadline order (i.e. streams with the earliest deadlines are added to the front of the queue). In EDF, an additional check is performed to verify that the stream has not already missed its deadline (this check can be omitted for soft-deadline use cases). In MultiLevel (ML), we created an additional *levelTwoStreamQueue*, creating a lower priority level. Higher priority streams are added to the main *streamQueue* in ascending deadline order (as in EDF), and lower priority streams are appended to the *levelTwoStreamQueue* (as in RR). For FCFS, we append to the end of the *streamQueue*.

Next, we modify the behavior when the active *streamQueue* is being processed by the `framer` to create packets. In FCFS, EDF, and ABS, the stream we are currently processing is either the one that "came first", has the "earliest deadline", or has the "highest priority", respectively. Consequently, it remains at the front of the queue if it still has data to be sent. If the stream has no more data to send, it is removed from the queue. EDF does an additional check that discards the stream if its deadline has passed (as mentioned before, this can be omitted). The MultiLevel (ML) processes the primary *streamQueue* while it has active streams, utilizing the EDF approach above. If the primary *streamQueue* has no more active streams, the secondary *levelTwoStreamQueue* gets processed, utilizing the same behavior as RR.

4.2. RL Router Agent

We train our router's policy, represented by a 2-layer MLP with hidden size 256, using the DQN algorithm. To prevent over-estimation of Q-values we employ Double Q-learning and use a target network that gets updated every 500 iterations. We use a discount rate of 0.99 and a learning rate of 0.0001. For exploration, we use an epsilon-greedy strategy. We performed minimal hyper-parameter tuning and use this for all trained policies. All models are served using vLLM (Kwon et al., 2023), which is a state-of-the-art inference serving system. The state that the agent sees is the batch size at each model in the system, the request's task, and an estimate of the request rate. The action is the selected model, and the reward is the model's accuracy if the latency requirement is met and zero otherwise.

4.3. System Implementation

For the frontend and backend servers, we provisioned two GCP VM instances running Ubuntu 22.04 with 8 vCPUs. In order to simulate 3G network conditions (mimicking edge placement of the frontend server), we utilized the *tc* Linux kernel command, which allows a user to adjust packet drop rate, packet latency, and connection bandwidth on any specified network interface. We utilize socket IPC to forward requests to-and-from the QUIC and the model-serving runner. Apart from modifying network interface characteristics through *tc*, the network topology between the front-end and back-end servers was hidden, as is commonly the case in cloud environments.

5. Evaluation

5.1. RL Microbenchmarks

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 request periods at a fixed arrival rate before the arrival rate changes. On the other hand, the MAF2 trace (Zhang et al., 2021) has much more unpredictable client behavior and the arrival rates rapidly change. Based on these observations, we evaluate our system on three types of synthetic workloads that capture a wide range of client behavior. 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 workload represents one in which the arrival rate of requests rapidly switches due to an underlying stochastic process that controls the arrival rate and its duration.

5.1.1. Environment

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, as depicted in Figure 3. 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. 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 pick tasks uniformly at random. We train the policy for 1.2 million iterations using hard deadlines, a uniform task distribution, and randomly chosen arrival rates.

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

5.1.2. STABLE WORKLOAD

For the stable workload, we vary the arrival rate of the Poisson Process from 0.25 to 48 requests per second and server for 40 seconds at each arrival rate before resetting and going to the next arrival rate. We show the results with hard deadlines in Figure 4. In the hard deadline setting, a client request's reward is zero if the policy does not pick an action that returns a response to the client within the latency requirements. As baselines, we show the performance when only serving to OPT-6.7B, only serving to OPT-1.3B, only serving to OPT-125M.

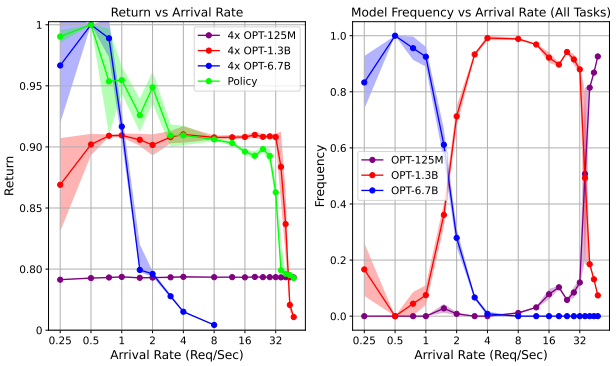

Figure 4. The left figure shows the performance on the environment with hard deadlines. The right figure shows the distribution of model selection from the policy.

As Figure 4 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 client utility sharply declines. While OPT-1.3B can serve requests at much higher arrival rates, it's 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 returns to route 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 4 as it is able to multiplex between models at a given arrival rate.

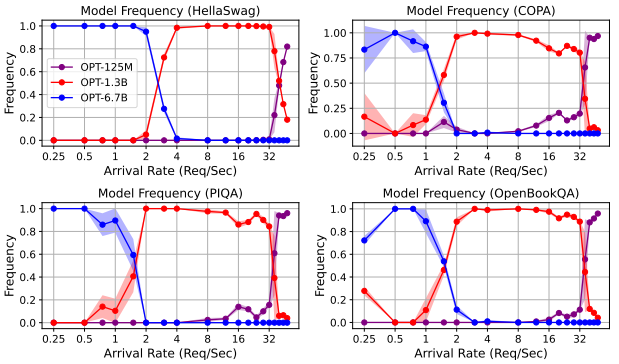

Figure 5. Model selection frequency for each individual task with hard deadlines.

We now examine how the routing varies for different tasks, as shown in Figure 5. We see that the policy sends HellaSwag tasks 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. Furthermore, when the arrival rate is very high, HellaSwag is sent to OPT-1.3B more often than the other three tasks, for the same reason as above. 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.

5.1.3. UNPREDICTABLE WORKLOAD

We evaluate on an unpredictable workload with large bursts, as mentioned in subsection 5.1. 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.

Figure 6 shows the performance of the routing policy as well as the baselines, in addition to the changing arrival rate. We show both the running average of performance across all

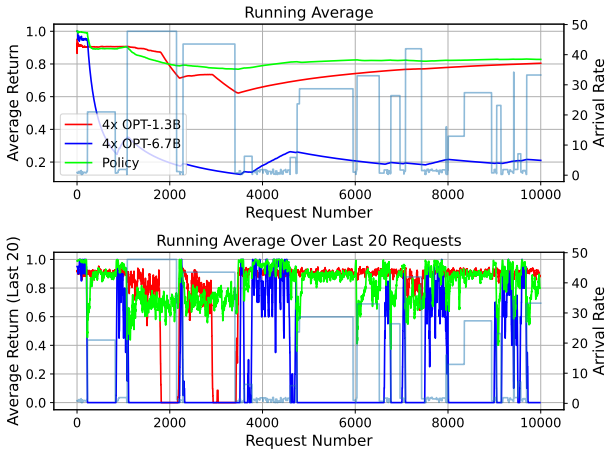

Figure 6. Running total and windowed average over the last 20 requests of performance on the unpredictable workload. The arrival rate at each step is also shown.

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 utility to the user is highly variable. Even though OPT-6.7B's windowed average has many averages near 1, the policy is in fact able to achieve more of these peaks. We quantify this in Table 2.

Table 2. Number of request windows of size 20 that meet average quality thresholds on the first 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

As shown in Table 2, OPT-6.7B is able to achieve more windowed averages with the top average of 1 compared to the policy. However, when analyzing the number of windows which meet high utility 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 at 99% quality, $2.32\times$ more windows at 98% quality, and $4.11\times$ more windows at 96% quality compared to OPT-6.7B. Additionally, it achieves 94% of peak quality $2.75\times$ 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 in the same window, even while faced with an unpredictable workload.

5.2. QUIC Scheduler Microbenchmarks

5.2.1. EXPERIMENTAL SETUP

To evaluate our implemented schedulers, we set up a network test between the frontend and backend VM instances. As mentioned previously, we utilized the *tc* Linux kernel command to emulate various 3G network characteristics. We ran QUIC and TLS/TCP clients on the frontend VM, which initiated streams/connections to the QUIC and TLS/TCP servers on the backend instance. The server echoes all the bytes it receives back to the client leveraging the bidirectional streams of QUIC or the bidirectional connections in TLS/TCP. We timed the round-trip total, starting when the client first establishes a connection/stream and ending when the client receives the final echoed byte from the server. We also evaluate the completion rates of each scheduler, based on the final round trip latency measurements.

We benchmark the newly implemented EDF, ML, and ABS schedulers against the baseline RR QUIC scheduling implementation; we also utilize the new FCFS scheduler as a baseline representing other implementations' default scheduler. Moreover, we compare our modified schedulers against TLS/TCP, since the HTTP-2/TLS/TCP stack is the industry standard protocol used in open-source model-serving applications. For TLS/TCP, we imitated stream multiplexing behavior by opening the same number of TCP connections between the client and server as we did streams in the QUIC tests. However, since we also wanted to take into account the overhead of opening many TCP connections and provide a standard of comparison against QUIC's singular connection, we also tested a 1-connection TLS/TCP setup. We test against TLS in conjunction with TCP since QUIC embeds the TLS protocol by default to provide secure transport (Iyengar & Thomson, 2021), and we wanted to provide an equal standard of comparison considering the increased computational overhead.

For our tests, we established 3 classes of traffic, representing specific LLM usage patterns. The first class of traffic is what we label "Real-Time Short" traffic, consisting of 8 KB long streams with a 1-second deadline. This traffic represents common LLM usage patterns, to be served with strict latency requirements optimizing interactive user experiences. Despite each LLM utilizing its own tokenizing methodology, a commonly quoted amount for the number of characters in a token is somewhere between 4-5 (Raf, 2023) (Kadous, 2023). 2000 tokens is well within the context window sizes of most small-sized models (OpenAI, 2023) and represents the common usage pattern of supplying a model with background contextual information followed by a question based on the context. Since most system character encodings represent each character with a byte of data, 2000 tokens is approximately 8 KB; as such, we

have selected it as the size for Short Traffic. In EDF this traffic class is assigned a 1-second deadline, in ABS it is assigned the highest priority level, and in ML it is assigned the higher priority queue. We made it our primary objective to schedule streams to achieve as low of a latency and as high of a completion rate as possible for this traffic class.

The second class of traffic is what we label "Free-Tier Short" traffic, consisting of the same 8 KB request streams as the above Real-Time Short, but representing traffic with more lenient deadlines and QoS requirements - perhaps "free-tier" traffic for LLM providers. In EDF this traffic class is assigned a 2-second deadline, in ABS it is assigned the lower priority level, and in ML it is assigned the lower priority queue.

Finally, we label the third class of traffic as "Long Output" traffic. This traffic class consists of 102.4 KB long streams with a 7-second deadline, representing long-input, long-output corpora synthesizing tasks. 25,000 tokens is on the higher end of context window sizes but is attainable by today's state-of-the-art advanced LLMs (OpenAI, 2023) (Anthropic, 2023). In EDF this traffic class is assigned a 7-second deadline, in ABS it is assigned the lower priority level, and in ML it is assigned the lower priority queue.

We simulate the 3G network via *tc*, dialing in 5 Mbps of bandwidth, 0.5% packet loss, and 50 ms of added packet latency (Chan & Ramjee, 2002). The client sends 90 requests to the server, randomly selecting a traffic class (with uniform probability), and recording the time from when the stream/connection is first established, to when last byte of the request is echo-ed back from the server. The latency of each request is recorded, and a completion rate is calculated based on the request class deadline.

5.2.2. Discussion

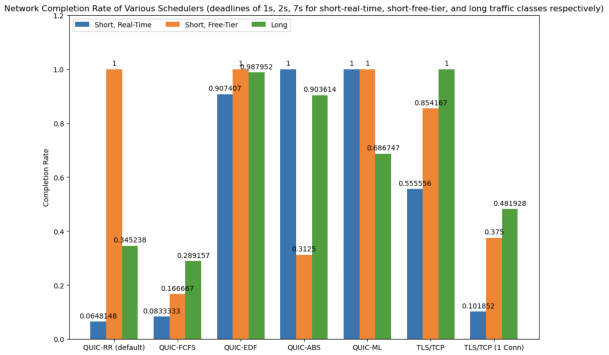

Figure 7. Network completion rates of the 3 traffic classes using various scheduling algorithms and network protocols. Short, Real-Time tasks with 1-second deadline; Short, Lower-Priority tasks with 2-second deadline; Long Context Window, Lower-Priority tasks with 7-second deadline.

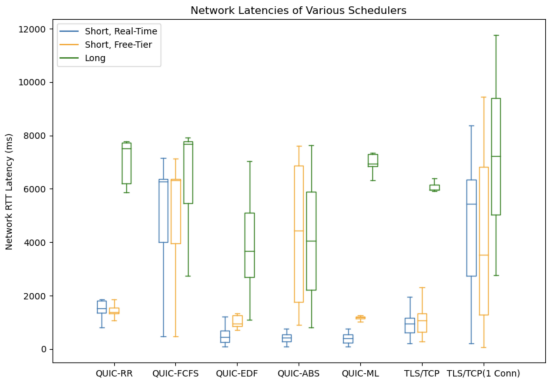

Figure 8. Network request latencies of the 3 traffic classes using various scheduling algorithms and network protocols.

From the box plot of latency distributions, we see that QUIC-RR and TLS/TCP (n-connections) performed similarly. Intriguingly, the performances of the two protocols are quite evenly matched despite the on-paper advantages that QUIC has. We hypothesize several possible reasons - increased overhead of a large number of streams multiplexing a single QUIC connection, TCP integration and optimization for Golang (TCP is part of the standard language library whereas quic-go is independent), UDP kernel buffer size restrictions, and QUIC's strict maximum packet size limits - however, we leave investigation out of scope for this work.

When analyzing completion rates, we see that both QUIC-RR and TLS/TCP struggle to achieve high completion rates for high-priority traffic, which is our most important metric. TLS/TCP 1-connection latency results underscore the head-of-line blocking issues present in the protocol with exceptionally high tail latencies for all 3 traffic classes. We see a similar issue in QUIC-FCFS, where long-output streams block progress for both high-priority and low-priority short streams.

Given the primary metric used in LLM model-serving tasks is per-token latency and deadline SLOs, we decided to first build an EDF scheduler. From the boxplots, we can see that it outperforms QUIC-RR in latency for all traffic classes. Furthermore, QUIC-EDF achieves high completion rates across all 3 traffic classes, reaching 100% for short/low-priority. However, since short/low-priority stream deadlines can occur before short/high-priority stream deadlines (due to random selection and order of stream creation), high-priority traffic can be preempted and thus miss its deadline. As a result, we see sub-100% completion rates for high-priority traffic prompting us to build two other schedulers aimed to prevent the preemption of high-priority traffic.

Next, we implemented an absolute priority scheduler and evaluated its performance. Since streams would be served on a strict priority basis, preemption of high-priority traf-

fic would be avoided. As a result, we observe a 100% completion rate and near-optimal latency of high-priority traffic, accomplishing our primary scheduling metric. However, we note a huge drop-off in the performance of the low-priority/short traffic class, with especially degraded tail latencies. Since the priority scheduler serves each priority class on a first-come-first-serve basis, we see that low-priority/short traffic gets blocked behind low-priority/long traffic. This could be solved by assigning long-output traffic a lower priority; however, we reasoned that this would not match the intended semantics of the traffic classes.

In response, we created a novel multi-level scheduling scheme, to address both preemption and head-of-line blocking issues seen in previous schedulers. The multi-level scheduler addressed the high-priority preemption issues by placing high-priority traffic in its own EDF higher-level scheduling queue while also addressing low-priority/short traffic blocking issues by utilizing a round-robin approach for the lower-priority queue. Overall, we deem the ML scheduler to perform best since it meets the primary criteria achieving 100% completion rate and optimal latencies for high-priority traffic while also serving both classes of low-priority traffic with strong guarantees. We also observe the most consistent behavior with the ML scheduler, evidenced by the significantly lower latency variance in the boxplot.

5.3. Macrobenchmark

We now evaluate the performance of our network scheduling in combination with the learned router agent. Network scheduling and dynamic model selection can be applied in a variety of settings. However, for our evaluation, we focus on LLM applications that serve three classes of requests:

1. High priority traffic with small input sizes
2. Low priority traffic with small input sizes
3. Low priority traffic with large input sizes

As described in the above microbenchmark sections, an application may need to serve small input sizes if it is answering questions such as "How to seal wood?" The application may need to serve queries with large input sizes to answer questions such as "Summarize the following document about sealing wood: ...". We evaluate various network schedulers and serving mechanisms on this workload, using both stable and unpredictable arrival patterns.

The following are our metrics of success to test during evaluation:

- Meet user-defined latency deadlines and QoS requirements, achieving high request completion rates.
- Serve highest possible model quality given resource contention
- Handle both stable and unpredictable client requests patterns across a wide range of client request rates.

We show the average performance on both the stable and first unpredictable workload from subsection 5.1 in Table 3 and Table 4. Our performance score combines both model quality and latency. If request's latency deadline is met, the performance score for the request is the selected model's accuracy. Otherwise, if the latency deadline is not met, the performance is zero. Similarly, we serve HellaSwag, COPA, PIQA, and OpenBookQA. We denote OpenBookQA to be high priority traffic. The policy recognizes this by up-weighting the reward for serving OpenBookQA at high quality. We train the policy for 1.1 million iterations using OPT-2.7B as the larger model and OPT-1.3B as the smaller model. We see that the policy outperforms static serving baselines on the stable workload, but only outperforms OPT-2.7B on the unpredictable workload. We see that OPT-1.3B outperforms the policy in the unpredictable workload. We believe this is because the policy does not take into account the network latency when making its decision, although it takes into account the priority of the task. We leave this as future work.

Table 3. Performance on the stable workload.

SCHEDULER	OPT-2.7B	OPT-1.3B	POLICY
QUIC-RR	0.20	0.25	**0.42**
QUIC-FCFS	0.19	0.24	**0.36**
QUIC-EDF	0.21	0.37	**0.52**
QUIC-ABS	0.20	0.35	**0.49**
QUIC-ML	0.21	0.29	**0.45**
TLS/TCP	0.20	0.24	**0.41**
TLS/TCP (1 CONN)	0.18	0.22	**0.31**

Table 4. Performance on the unpredictable workload.

SCHEDULER	OPT-2.7B	OPT-1.3B	POLICY
QUIC-RR	0.19	**0.54**	0.42
QUIC-FCFS	0.17	**0.53**	0.36
QUIC-EDF	0.19	**0.64**	0.48
QUIC-ABS	0.18	**0.62**	0.46
QUIC-ML	0.19	**0.57**	0.44
TLS/TCP	0.18	**0.54**	0.41
TLS/TCP (1 CONN)	0.16	**0.51**	0.35

We also investigate the quality of OpenBookQA, which is the prioritized task in our system. As shown in Figure 9, OpenBookQA and HellaSwag are sent to OPT-6.7B significantly more often than COPA and PIQA. This is because HellaSwag is a hard task whose quality benefits significantly

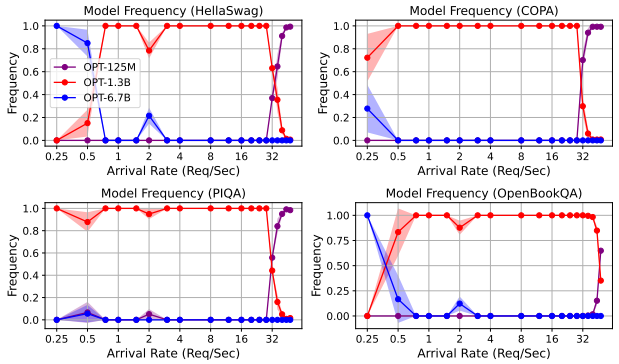

Figure 9. Model selection frequency for each individual task with OpenBookQA prioritized.

from OPT-6.7B, and OpenBookQA is a prioritized task in the system. Additionally, at high arrival rates, the system favors sending OpenBookQA to OPT-1.3B while it sends the other tasks to the smaller OPT-125M.

6. Future Work

There are a number of future directions to be explored to improve the router agent. When running multiple models on a GPU, there are scheduling decisions on the systems side that need to be made to determine how models share compute resources to further help meet latency deadlines. Additionally, it will be interesting to see extensions to the work that use embeddings or other ways to expand the state in order to capture further information about a client's request and the state of the system. On the network scheduling end, further exploration into the throughput tradeoffs between QUIC and TLS/TCP is necessary, as our initial benchmarks showed similar results between the two baselines. We also believe that the overall system performance can be improved by having the router agent take into account the network latency for a request, rather than just its priority. In addition, we can also assess the performance of multiple model-serving environments. We envision an environment where front-end servers can each select among a group of back-end servers, employing a load-balancing framework to evenly spread the request load among the group.

7. Conclusion

In this work, we have presented an end-to-end quality-of-service aware model serving system. We utilized a Q-learning-based routing agent to dynamically choose between models of different latency and quality during periods of high request load, maintaining availability and serving at the highest possible quality. We have also designed and implemented a series of stream scheduling algorithms for the QUIC network protocol. We identified the SLO requirements of various classes of LLM requests and utilized both network scheduling as well as dynamic model selection to meet application requirements. We have trained the Q-learning agent to optimize latency and model serving quality of higher priority classes while achieving fairness and best-effort response quality for lower priority requests. We have benchmarked the QUIC network schedulers and have shown that the EDF and Multilevel schedulers perform best, achieving the highest completion rates and lowest network latencies for high-priority traffic while maintaining high completion rates for lower-priority traffic as well. Overall, the network schedulers have shown increased performance over the baseline QUIC protocol, as well as TLS/TCP. We benchmark our end-to-end system against today's model-serving standard and demonstrate the ability to not only cater to QoS differentiation between requests but also outperform current standards in per-token latency and model-output quality.

References

Agarwal, M., Qureshi, A., Sardana, N., and Li, L. Llm inference performance engineering: Best practices. https://www.databricks.com/blog/llm-inference-performance-engineering-best-practices, 2023. Accessed: 12/14/2023.

Anthropic. Introducing 100k context windows, 2023. URL https://www.anthropic.com/index/100k-context-windows. Accessed: 12/14/2023.

Belshe, M., Peon, R., and Thomson, M. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, May 2015.

Bishop, M. HTTP/3. RFC 9114, June 2022.

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.

Bornstein, M. and Radovanovic, R. Emerging architectures for llm applications. https://a16z.com/emerging-architectures-for-llm-applications/, 2023. Accessed: 12/14/2023.

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

Cettolo, M., Federico, M., Bentivogli, L., Jan, N., Sebastian, S., Katsuitho, S., Koichiro, Y., and Christian, F. Overview of the iwslt 2017 evaluation campaign. In *Proceedings of the 14th International Workshop on Spoken Language Translation*, pp. 2–14, 2017.

Chan, M. and Ramjee, R. Tcp/ip performance over 3g wireless links with rate and delay variation. volume 11, pp. 71–82, 09 2002. doi: 10.1007/s11276-004-4748-7.

Chiariotti, F., Deshpande, A. A., Giordani, M., Antonakoglou, K., Mahmoodi, T., and Zanella, A. Quicest: A quic-enabled scheduling and transmission scheme to maximize voi with correlated data flows. *IEEE Communications Magazine*, 59(4):30–36, 2021. doi: 10.1109/MCOM.001.2000876.

Clemente, L. and Seemann, M. quic-go: A quic implementation in pure go, 2023. URL https://github.com/quic-go/quic-go. Accessed: 12/14/2023.

Crochet, C., Rousseaux, T., Piraux, M., Sambon, J.-F., and Legay, A. Verifying quic implementations using ivy. In *Proceedings of the 2021 Workshop on Evolution, Performance and Interoperability of QUIC*, EPIQ '21, pp. 35–41, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450391351. doi: 10.1145/3488660.3493803.

Cui, C., Lu, Y., Li, S., Li, J., and Ruan, Z. Dash+: Download multiple video segments with stream multiplexing of quic. In *2022 Tenth International Conference on Advanced Cloud and Big Data (CBD)*, pp. 66–72, 2022. doi: 10.1109/CBD58033.2022.00021.

Fernández, F., Zverev, M., Diez, L., Juárez, J. R., Brunstrom, A., and Agüero, R. Flexible priority-based stream schedulers in quic. In *Proceedings of the Int'l ACM Symposium on Performance Evaluation of Wireless Ad Hoc, Sensor, & Ubiquitous Networks*, PE-WASUN '23, pp. 91–98, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703706. doi: 10.1145/3616394.3618267.

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.

Hermann, K. M., Kocisky, T., Grefenstette, E., Espeholt, L., Kay, W., Suleyman, M., and Blunsom, P. Teaching machines to read and comprehend. *Advances in neural information processing systems*, 28, 2015.

Iyengar, J. and Swett, I. QUIC Loss Detection and Congestion Control. RFC 9002, May 2021.

Iyengar, J. and Thomson, M. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021.

Kadous, W. Numbers every llm developer should know, 2023. URL https://www.anyscale.com/blog/num-every-llm-developer-should-know. Accessed: 12/14/2023.

Kutter, M. and Jaeger, B. Comparison of different quic implementations. In *Proceedings of the Seminar Innovative Internet Technologies and Mobile Communications (IITM)*, 2022.

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.

Langley, A., Riddoch, A., Wilk, A., Vicente, A., Krasic, C. B., Shi, C., Zhang, D., Yang, F., Kouranov, F., Swett, I., Iyengar, J., Bailey, J., Dorfman, J. C., Roskind, J., Kulik, J., Westin, P. G., Tenneti, R., Shade, R., Hamilton, R., Vasiliev, V., and Chang, W.-T. The quic transport protocol: Design and internet-scale deployment. 2017.

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.

Narayan, S., Cohen, S. B., and Lapata, M. Don't give me the details, just the summary! topic-aware convolutional neural networks for extreme summarization. *arXiv preprint arXiv:1808.08745*, 2018.

OpenAI. Openai models, 2023. URL https://platform.openai.com/docs/models. Accessed: 12/14/2023.

Raf. What are tokens and how to count them? https://help.openai.com/en/articles/4936856-what-are-tokens-and-how-to-count-them, 2023. Accessed: 12/14/2023.

Rajpurkar, P., Zhang, J., Lopyrev, K., and Liang, P. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

RayTeam. Ray serve: Scalable and programmable serving, 2023. URL https://docs.ray.io/en/latest/serve/index.html. Accessed: 12/14/2023.

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.

Seemann, M. Quic interop runner, 2023. URL https://interop.seemann.io/. Accessed: 12/14/2023.

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.

Thomson, M. and Benfield, C. HTTP/2. RFC 9113, June 2022.

Thomson, M. and Turner, S. Using TLS to Secure QUIC. RFC 9001, May 2021.

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.

vLLM Team. vllm: Quickstart, 2023. URL https://docs.vllm.ai/en/latest/getting_started/quickstart.html. Accessed: 12/14/2023.

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, 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.