SwiftQueue: Optimizing Low-Latency Applications with Swift Packet Queuing

Siddhant Ray[1]
University of Chicago, IL, USA

Xi Jiang
University of Chicago, IL, USA

Jack Luo
University of Chicago, IL, USA

Nick Feamster
University of Chicago, IL, USA

Junchen Jiang
University of Chicago, IL, USA

Abstract

Low Latency, Low Loss, and Scalable Throughput (L4S), as an emerging router-queue management technique, has seen steady deployment in the industry. An L4S-enabled router assigns each packet to the queue based on the packet header marking. Currently, L4S employs *per-flow* queue selection, *i.e.,* all packets of a flow are marked the same way and thus use the same queues, even though each packet is marked separately. However, this may hurt tail latency and latency-sensitive applications because transient congestion and queue buildups may only affect a fraction of packets in a flow.

We present *SwiftQueue*, a new L4S queue-selection strategy in which a sender uses a novel *per-packet* latency predictor to pinpoint which packets likely have latency spikes or drops. The insight is that many packet-level latency variations result from complex interactions among recent packets at shared router queues. Yet, these intricate packet-level latency patterns are hard to learn efficiently by traditional models.

Instead, SwiftQueue uses a custom *Transformer*, which is well-studied for its expressiveness on sequential patterns, to predict the next packet's latency based on the latencies of recently received ACKs. Based on the predicted latency of each outgoing packet, SwiftQueue's sender dynamically marks the L4S packet header to assign packets to potentially different queues, even within the same flow. Using real network traces, we show that SwiftQueue is 45-65% more accurate in predicting latency and its variations than state-of-art methods. Based on its latency prediction, SwiftQueue reduces the tail latency for L4S-enabled flows by 36-45%, compared with the existing L4S queue-selection method.

2012 ACM Subject Classification Networks → Network architectures; Computing methodologies → Machine learning approaches

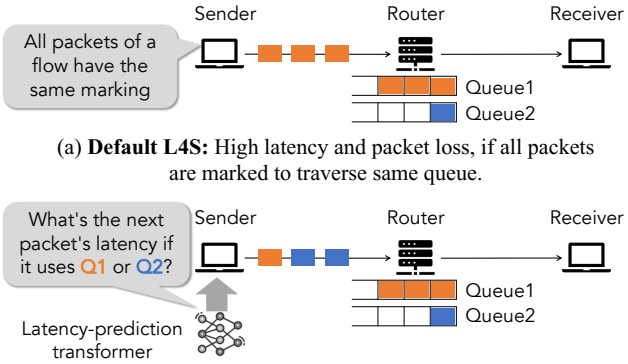
Keywords and phrases Latency prediction, L4S Queue Management

Digital Object Identifier 10.4230/OASIcs.NINeS.2026.24

1 Introduction

Internet Service Provider (ISP)-supported QoS enhancement in wide area networks has been long-awaited. Fortunately, the gradual yet steady deployment of L4S in ISPs over the last few years enables sender-driven queue selection on the network routers [16, 14, 58, 50, 6]. The sender marks the outgoing packet headers of L4S-enabled flows, while the L4S-enabled routers

[1] Corresponding author



(a) **Default L4S:** High latency and packet loss, if all packets are marked to traverse same queue.

(b) **SwiftQueue-based L4S:** Lower delay, if all packets are marked differently so that traverse different queues.

Figure 1 To reduce tail latency, SwiftQueue predicts latency per packet and choose the appropriate queue for each packet.

along the path will use different queues for these packets depending on the marked headers. With this new capacity, the sender could mark the packets in a way such that packets that would otherwise experience high latency can now go through alternative lower-latency queues, potentially reducing the tail end-to-end latency for low-latency applications.

As with many previous sender-driven queue selection techniques (*e.g.,* [29, 5, 92]), L4S marks *all packets* of a flow in the same way, *i.e.,* all packets in a flow are assigned to the same queue. Yet, even within a flow, it is beneficial to select queues differently for different packets, as the latency of a router queue can temporarily increase or decrease significantly. Drastic changes in latency may occur due to inter-packet interactions on a *shared network* in a short time window [49], which can cause sudden changes in the network state. Realizing the full potential of L4S's sender-driven queue selection thus requires predicting precisely when latency spikes and drops happen at the per-packet level.

Predicting sudden, per-packet latency changes is a longstanding challenging problem [61, 60, 90, 48]; and most congestion control and queuing schemes only *react* to packet-level latency changes. Inspired by the Transformer's success in other time-series domains (*e.g.,* natural languages) [89], we view the sequence of packets and their latencies as a token sequence, and ask a simple question: *can Transformers learn to predict packet latency accurately enough to allow L4S to proactively reduce tail latencies for low-latency applications?*

This paper shows that Transformers when trained appropriately, can predict sharp latency changes much more accurately than prior methods, even when tested on network traces unseen in training. This stems from the observation that while some latency variations are indeed due to exogenous factors (*e.g.,* user clicks), not all changes are. It is possible to have some learnable events in the network indicated by signals embedded in the latencies of the past packets. For instance, video applications (*e.g.,* Netflix or Zoom) can have periodic packet bursts of video chunks or frames within a session, and these flows can have some learnable patterns for their latencies [4, 70]. For another example, the interaction of packets from running flows using certain congestion control logic can result in learnable latency patterns (*e.g.,* queue buildup and back-off in TCP Cubic [32]). Of course, the real latency patterns can be more complicated than these simple examples.

Learning to extract signals from past packets' latencies has been challenging for traditional time-series models. Specifically, simple models (*e.g.,* EWMA [65], ARIMA [87]) can learn seasonality and smooth past values, but they fall short of capturing more complex and

finer-grained inter-packet dependencies [98]. Deep learning models offer better learning and expressiveness, but training and updating these models can be slow and data-intensive [45, 11]. The Transformer's unique advantage comes from its parallel *attention* mechanism, which simultaneously captures complex sequential patterns within a long context window and allows faster training and convergence than pre-Transformer neural networks, *e.g.,* Recurrent Neural Networks (RNNs), Long-Short Term Memory Networks (LSTMs) [35] For these reasons, Transformers have shown great promise in general time-series predictions [68, 104, 106, 94].

Inspired by this, we present *SwiftQueue*, a new L4S queue-selection system driven by a *custom Transformer* for per-packet latency prediction. SwiftQueue has two components: (1) a Transformer that predicts next outgoing packet's latency based on information (latencies and queue selections) of recent packets from the same sender, and (2) a queue-selection logic that uses the Transformer-based latency prediction to select the queue for outgoing packets in an L4S flow and marks the packet headers accordingly so that L4S-enabled routers will put the packet in the corresponding queues. As illustrated in Figure 1(b), since SwiftQueue assigns queues per packet rather than per flow, it can reduce tail latency.

SwiftQueue presents the first effort to adapt Transformers for *packet-level* latency prediction. Our custom Transformer focuses on neighborhoods of sharp changes and learns to predict the latency of future packets based on the timeseries of past latencies and packet information. Unlike prior uses of Transformers in networking [18, 31, 52, 27], which focuses on *flow-level* estimates or classification, our goal requires learning latency patterns due to *cross-flow* interactions at the packet level and making fast prediction per packet.

Contributions

In short, we make the two contributions:

1. First, we show that our Transformer predictor, which is continuously fine-tuned using past packet latency values, can accurately predict latency spikes or drops 45-65% more accurately than the state-of-the-art methods. Moreover it can make per-packet latency prediction fast enough for a 200-Mbps link by making batched predictions.
2. Second, we show that, using SwiftQueue, a sender can significantly reduce the tail latency by 36-45%, by first predicting the latency of each outgoing packet in the L4S flow if it goes through the L4S queues or classic queues and then marking each packet for queue selection such that its predicted latency is minimized.

2 Background

2.1 Queue selection and management

L4S dual queue

L4S is already being rolled out for deployment at scale [14, 58, 50, 6] and is expected to be the dominant solution for reducing queuing latency for low-latency applications (*e.g.,* cloud gaming) and networks (low-latency 5G/ISP). The L4S Dual Queue [78, 7, 79] is developed to mitigate bufferbloats and excessive queuing latency in networks. With the L4S Dual Queue, the sender tags a packet header's ECN bit [21] to indicate if a packet should go through the "Classic" queues or the "L4S" queues in the L4S-enabled routers (Figure 2). To sort the traffic into each of the two queues correctly, the L4S dual-queue relies on packets being tagged appropriately. This enables different rules for ECN marking and/or dropping in each of the queues. That said, currently, L4S senders only select queues per flow/application based on the application's nature (video or web) or user preference.

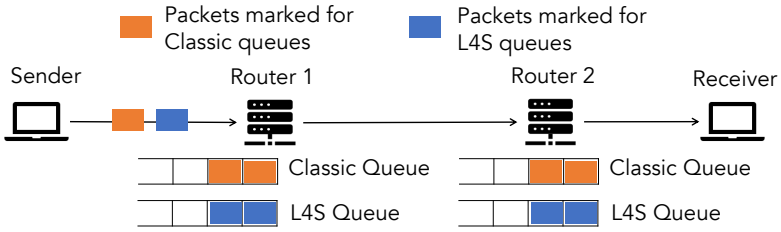


Figure 2 L4S by default marks all packets from the flow into the same queues, and once marked, packets traverse the chosen queues at all hops.

Active queue management (AQM)

Before L4S, AQM has already been an active area of study for decades to ensure the efficient co-existence of TCP flows. Several AQM methods have been proposed for optimising different aspects, including queuing latency reduction, such as Random Early Detection [23], CoDel [67] and FQ-CoDel [36]. Traditional AQM algorithms have been built to run in conjunction with TCP congestion control algorithms, which rely only on packet loss as a signal (as opposed to L4S, which uses ECN markings). To handle the bursty nature of TCP, these AQM techniques are equipped with large data buffers to prevent excessive packet drops due to these bursts. However, bufferbloat arises when queue length grows [26], and latency can often build up as a result of individual bufferbloats at multiple routers on the network path. As traditional AQM techniques do not allow per-packet queue selection, they are unsuitable for low-latency applications where latency spikes happen at the packet level.

2.2 Related work

Latency prediction

Latency prediction in networks has been studied for decades. Most recent work focuses on estimating latency values at the flow level (*e.g.,* flow completion times or average packet latency in a flow) using heuristics [103, 25] or deep learning [52]. There has also been work on measuring latency for network performance estimation at scale [44, 73, 30]. Other works have focused on predicting latency from noisy and incomplete network measurements [86]. Finally, latency prediction has been actively explored at the application level for video streaming [53, 96], HTTP web traffic [37, 69], cloud datastores [84], and CDN serving [101].

Machine learning for network traffic analysis

Recent advancements in machine learning and deep learning have significantly impacted network traffic analysis for various tasks, from conventional anomaly detection and traffic classification [42, 59] to more complex behavior characterization [8] and traffic content forecasting [41, 40]. Various approaches have been proposed, including conventional models like simple regression and tree-structured classifiers [43, 51, 1], as well as advanced deep learning techniques such as recurrent neural networks (RNNs) [46, 74], convolutional neural networks (CNNs) [57] and diffusion models [82, 40]. These methods have shown promise in understanding patterns in network traffic data but often fall short in handling long-range dependencies and sharp changes in traffic patterns [3, 39].

Transformers have been proposed as a new class of models to reason about more complex sequential dependencies in a variety of fields. In the networking domain, the Transformer architecture has already shown promise in predicting flow-level behaviors [18, 31, 52], telemetry

signals [27], and other time-series [93]. They have also been used recently to generate network traffic [63, 105] and analyze traffic behaviour at the flow level [100]. These architectures use decoder-based Transformers to generate packet headers with semantically correct header structure but often lack temporal inter-packet patterns. Predicting actual packet latency values is different from generating packet headers. We need to learn packet dependencies within flows and across flows from an endpoint, which the models above do not address yet.

In short, we summarize the background as follows:

1. Prior sender-driven queue selectors, including L4S's header marking strategy, only selects queues per flow/application as it does not predict packet-level events.
2. Transformers have shown promise in some networking problems but not for per-packet latency prediction.

3 Motivation

Inspired by the Transformer's promise, this work applies the Transformer to packet-level queue-selection in L4S. Importantly, our work is focused on queue selection at the *sender* and *without* changing the L4S routers or the sender's congestion control logic.

3.1 L4S queue selection: per-*packet* vs. per-*flow*

Mechanically, L4S AQM architecture allows the sender to mark each packet in a flow differently into different queues, but by default, all packets of a flow are either sent through the classic queues or the L4S queues. This can be suboptimal. For example, if the network has more low-latency flows than classic traffic flows, L4S will, by default, put all packets from low-latency flows on the L4S queue on the bottleneck router, causing the L4S queue to build up and create additional queuing latency for low-latency flows.

Current L4S vendors pre-partition queues for classic and L4S traffic for a given link subscription. The L4S queue is designed to be much shorter as L4S latency-sensitive traffic has been measured around 10-20% of the overall network traffic.[2] However due to these fixed size queue allocations, if latency-sensitive traffic becomes capacity seeking on the link even for a short window, packets which would experience high-latency can be switched to the classic queue. This prevents L4S flows from experiencing latency spikes which would adversely have affected application-level Quality-of-Experience (QoE).

This way, one could potentially reduce tail latency by focusing the use of L4S queues on only the packets with high latency due to transient queue buildup and congestion. Such per-packet queue selection could particularly benefit the flows that have stringent tail-latency requirements and only share a part of the entire network path with the other flows. For example, if we have some L4S flows at a given router, which has packets that would experience further downstream congestion (due to cross traffic), we can select a non-congested queue for those packets at the current router, minimizing the tail latency as a result of those packets. We evaluate this in detail in §6.3.

With a per-packet latency predictor, we can predict which packets in the low-latency flows would experience high latency and then mark them to let the router put just these packets into a different, lower-latency queue (classic queue) than other packets in the same flow, which reduces the tail latency. Meanwhile the current L4S queue would have been serviced and we can continue to put the next incoming packets back on the L4S queue after its queue length becomes low.

[2] Data collected after conversation with a leading US telecom vendor deploying L4S on their network.

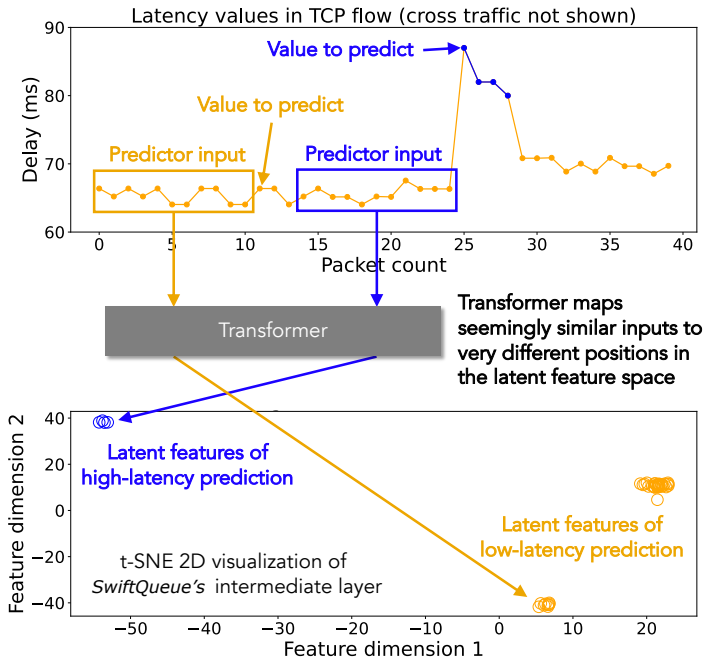


Figure 3 A Transformer maps the inputs of low-latency packets and those of high-latency packets to separate clusters.

3.2 Latency prediction needs expressive models

To enable packet-level queue selection, per-packet latency prediction is essential, especially when latency spikes and drops happen. Classical ML models have been shown to perform poorly at capturing sudden variations in time-series [107]. Neural network models are superior at learning complex feature relationships in time-series data as they have much greater memorization and reasoning.

The core innovation of Transformers is the *self-attention* mechanism [89], which allows the model to weigh the importance of different elements (tokens) in a sequence when making predictions. The self-attention mechanism works by computing a weighted sum of input elements, where the weights are determined by the relevance of each element to the current prediction. This allows focusing on different parts of the input sequence in an adaptive manner, enhancing its ability to capture dependencies and interactions.

Transformers have shown promise as a superior tool for general time-series prediction tasks [54, 56, 68, 104, 106, 94], leveraging the ability of the attention mechanism to capture complex temporal patterns much better than other deep learning models. Transformer's *attention* mechanism offers greater memorization of past interactions [95]. While other deep learning models like LSTMs and RNNs have been used for time-series prediction, they suffer from well-studied issues of *memory limitations* [38, 83] and *vanishing gradients* [55, 71] which reduces their prediction accuracy and training speed.

To showcase that a Transformer, even at a small size, could capture the complex patterns before latency changes, Figure 3 visualizes the intermediate feature space of different Transformer inputs. This Transformer, which has 2 layers (will be described in more detail soon), predicts a given packet's latency based on recent packets' latencies as input. The intermediate feature, a 64×64 tensor, is visualized by first mapping the tensor to a two-dimensional space using t-SNE [88].

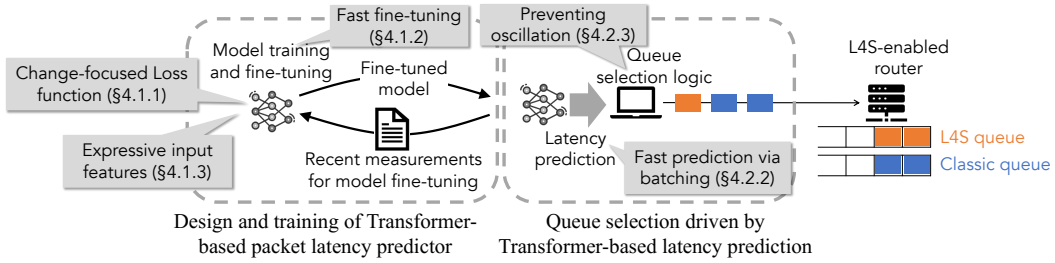


Figure 4 SwiftQueue consists of a custom Transformer latency predictor and a prediction-driven per-packet queue selector.

As an example, we consider two particular different packet latencies under prediction. While their inputs may seem similar (the blue box and the yellow box), their features produced by the Transformer fall in well-separated clusters. Specifically, for the *very first* packet after a latency spike (the blue point), the Transformer already maps them to the correct feature region, even if significantly increased or decreased delay is not visible in their inputs. In other words, even though their inputs have little noticeable difference to packets prior to the spike/drop, the Transformer can tell the difference from the distribution embedded in the values!

4 SwiftQueue Design

Now, we present SwiftQueue, a Transformer-based L4S queue selection system. As depicted in Figure 4, SwiftQueue has two components: a Transformer as a *per-packet latency predictor* and a prediction-driven *proactive L4S controller*.

While L4S can benefit from per-packet queue selection, for which Transformers could be a good fit, making SwiftQueue practical creates challenges in both of its components.

Per-packet latency prediction (§4.1)

1. Sharp latency changes, due to their rarity, are more difficult to predict, but they are also crucial as they are when proactive queue switching is most useful. How to make SwiftQueue predict sharp latency changes accurately?
2. Transformers are workload-dependent, so it must be able to update itself with the new latency measurements frequently. How to fine-tune SwiftQueue's Transformer with the latest measurements with minimum overhead?

Queue selection controller (§4.2)

1. To inform per-packet marking, the latency prediction must be fast. How can SwiftQueue make predictions fast enough without sacrificing its accuracy?
2. Straightforward prediction-driven control can cause oscillation as it may assign too many packets to a seemingly low-latency queue before their ACKs are returned. How can SwiftQueue prevent such oscillation?

4.1 SwiftQueue's latency predictor

Transformer basics

Transformers can learn sequentially dependent information at scale using the parallelized *attention* operation. The attention module removes the need to explicitly model the sequence for learning relationships. Instead, it utilizes a weighted dot product matrix, which assigns importance to a sequence value based on its relative importance to all other values. This matrix is updated during training to understand sequential dependencies between different values. While they work well for general sequence prediction tasks, to effectively perform latency prediction, we make several custom changes to adapt to the challenges of network packet sequences and traffic patterns.

Predicting sharp changes, not values

In network packet traces, we may have a series of packets with similar delays, interspersed with occasional sharp changes in latency. A sharp change often results from increased queuing latency as the queue might be occupied by packets from a competing flow. It is useful to select a lesser-filled queue for packets that would have otherwise experienced such an increase in queuing latency. Hence, we want the model to focus on learning to predict these sharp changes instead of predicting every latency value when latency changes smoothly.

More specifically, in this paper, we define a sharp latency change as at least 20ms difference and a relative 20% difference between two successive packet latencies. We choose these values as in practice, a change of 20 ms and higher starts being observable for low-latency sensitive applications like VoIP [91] and cloud gaming [15] on ISPs.

4.1.1 Focusing on sharp changes with new loss function

Why not standard loss functions?

Standard deep learning models often use loss functions such as the Mean Squared Error (MSE) to train the models, to make the predicted value closer to the ground truth. This helps the model in learning patterns in the input data over average trends in the input. However, these traditional loss functions, such as the MSE, try to reduce prediction errors *everywhere* [97], rather than focusing on *sharp changes*. As we just discussed, SwiftQueue needs the Transformer architecture to specifically target the neighbourhoods of sharp changes and accurately predict them for better packet-level queue selection. Thus, the key question is, *how to make SwiftQueue focus on sharp latency changes?*

Focusing loss function on change points

For this, we design a custom loss function based on the original MSE loss. We employ a scaling factor to the loss, in the case the absolute difference successive latency of two values is greater than a given threshold (*i.e.,* a sharp latency). This helps the model give greater attention to this neighbourhood, and the increased difference in latency signifies the sudden drop or spike in latency. For a given prediction $p(x)$ and a target value $t(x)$, we define the loss function LF as

$$LF(p(x), t(x)) = \begin{cases} \alpha \cdot (p(x) - t(x))^2 \text{ , if } |t(x) - t(x-1)| \geq \delta \\ (p(x) - t(x))^2 \text{ , otherwise} \end{cases}$$

Our loss function LF has two hyper-parameters: α as the penalizing factor, in case there is a sudden change in latency greater than δ. The optimal choice of these values can vary with applications. SwiftQueue's current design chooses $\alpha = 10$ and, following our definition of sharp changes (§4.1), δ is at least 20 ms difference in addition to at least a relative 20% difference between two successive packet latencies. We observed that the SwiftQueue can learn well under this design.

This loss function does come with a trade-off: it has a slightly higher mean prediction error *averaged over all packets*, but is able to accurately predict many more of the sharp changes than if the model is trained with traditional MSE-based loss function. We show this in §6.3. For our L4S queue-selection, we need SwiftQueue's predictor to accurately predict latency spikes and drops for packet-level queue selection.

4.1.2 Fast online model fine-tuning

We train the SwiftQueue to learn the interaction between packets from their implicitly embeddings in the packet sequence. These interactions can significantly change with changing traffic patterns. For example, if new flows start, flow interactions can significantly change. Adapting to such *concept drift* is necessary for a model like SwiftQueue.

The Transformer architecture is known for its fine-tuning efficiency compared to other neural network models. It can perform comparatively fast model updates, but the fine-tuning speed of traditional Transformers (*e.g.,* used for language modeling) is still too slow for packet-switched networks. Commonly used Transformers take tens to hundreds of minutes to fine-tune [75], which is too slow to keep up with changing network traffic. We observed that to use SwiftQueue's predictions on long traces with new flows, fine-tuning the model is needed for continued correct sharp change predictions.

To further speed up fine-tuning, we design the Transformer architecture to have *fewer* trainable parameters with two Transformer Encoder-based *attention block* layers followed by two *linear layers* derived from the PyTorch [72] implementation. In total, it has $\sim 100K$ trainable parameters. This is much smaller than most Transformer models over the years, which have parameters in the order of $100M - 400B$ [17, 20, 19]. To further speed up the fine-tuning, we update *only* the linear layers (includes activation) of the model as SwiftQueue has already learnt the representation of latency changes and just needs to focus on new interactions in the recent network state.

To minimize the compute overhead of fine-tuning, we do not want to fine-tune excessively with marginal gains as we want to prevent resource wastage. To find a reasonable fine-tuning interval, we independently measured the performance at different fine-tuning intervals, starting from the same pre-trained model. We found a common regular interval which performs well across our workloads. We evaluate this in §6.3.

4.1.3 Input features of the latency predictor

In addition to our customization of the loss function and the model architecture, we also take care in designing the *input* of SwiftQueue's latency predictor. As part of the Transformer's input, the end-to-end latency of each acknowledge packet includes the queuing delay of the packet and the propagation delay on the link. Thus, the latency value is influenced by the interaction between packets on the network, and it reflects the effect of events queue buildup, congestion, and packet loss.

That said, the past latency values *alone* are insufficient to make SwiftQueue learn effectively.

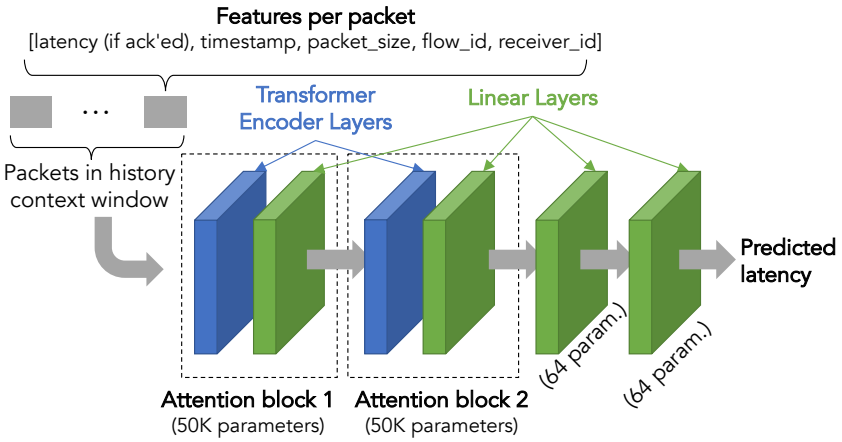


Figure 5 SwiftQueue's architecture with showing its context window, multiple *attention blocks* and *linear layers*.

Input features beyond just latency values

The parallelized attention mechanism lose the explicit *position* of each item in the sequence and must be supplied with this information externally in the form of *positional embeddings*. Positional embeddings can take many forms [80, 89, 24]. Fortuitously, network traces *naturally* provide the positional information with *packet timestamps*. We use the *relative timestamp* of a packet with respect to the first packet of the flow as our absolute positional information.

Moreover, we discovered that only using past latency values and timestamps does not provide enough context about the network for learning latency prediction. We select a few more features in order to indicate the state of the network. Inspired by the authors in [18] where they show the need to jointly look at both packet-level and network-level features, for SwiftQueue we choose the following features as input to supplement packet latency values and timestamps.

- *packet_size:* The packet size along with the timestamp implicitly helps SwiftQueue understand the real-time sending rate. The size of the packet affects the number of packets which can be accommodated in the queue buffer, *e.g.,* larger packets means fewer packets are present in the queue. Using the packet size helps SwiftQueue understand packet interactions inside the queue better.
- *flow_id:* Assigned to all packets of the same flow with the same value, the flow_id lets SwiftQueue distinguish between the packets from different concurrent flows from the same sender.[3] This helps SwiftQueue understand interactions of packets between flows better.
- *receiver_id:* Assigned to all packets being sent to the same receiver endpoint, the receiver_id helps SwiftQueue understand which packets likely traverse the same path and the effect of different path latencies.

[3] SwiftQueue's current design requires the visibility of the latency and L4S decision on all packets being sent out from the same end-point in order to make accurate predictions. Note that this does not require visibility into each flow's internal TCP state machine. Implementing this in the Linux kernel requires a new module that can access multiple TCP streams. While we do not focus on the exact implementation of such a module in this paper, several recent works [66, 99, 28] have developed a centralized TCP CC plane, which views multiple data paths and integrates a central feedback loop. Several of these already have Linux kernel integration. Compared to these efforts, our design has a much simpler requirement – it only needs to look at flows from the same endpoint. Thus, we believe a portion of an existing centralized TCP CC module can be reused by us for SwiftQueue.

It is true that adding more features may further improve learning, but these features are already sufficient for SwiftQueue to learn to predict latency at the sender.

History context window size

General Transformers scale quadratically $O(n^2)$ with input sequence size n (in tokens). However, sequences in network traces are often over $1000s$ of packets in length for a trace measured over a few seconds. Since learning from extremely long sequences can scale poorly, we need to choose *how many packets into the past do we have to look into to learn sharp latency changes?*

Since sharp latency changes in packet sequences are often constrained to localised neighbourhoods of effect (*e.g.,* a queue fills up and drops packets), we observed in practice we can use shorter histories of packet sequences to train SwiftQueue's predictor. In particular, we use a sliding window approach, where we train the Transformer on smaller sliding windows of history size rather than the entire trace, reducing the quadratic scale factor.

Rather than choosing a fixed window size, SwiftQueue matches the sliding window size to the maximum *bandwidth-delay product* (BDP) of the link. This roughly covers the number of in-flight packets in the network at all times and we observed that packets which are queued together have greater interaction over the current network state. For example, a recent packet history of $1-2\times$BDP can signal queue buildup. We can dynamically adapt the sliding window during training to match changing BDP values for various network conditions. We allow SwiftQueue to learn these interactions over multiple sliding windows. Of course, our choice of history window size may not be optimal, and more research will be needed to pick the optimal window size.

4.2 SwiftQueue's L4S queue selection

We design SwiftQueue to predict latency and act as a packet-level controller for queue selection at the sender. After SwiftQueue predicts whether a set of subsequent packets is likely to experience high latency, we leverage the L4S's packet header marking to influence the queue selection at L4S dual-queue routers. We explain this in more detail next.

L4S queue selection with SwiftQueue

Current L4S queue selection involves putting all packets from a given L4S flow into the same queue. Yet, recall that this approach can be problematic when dealing with localised increasing cross-traffic congestion in a short window (§3.1). Such events result in higher packet latency, pushing up the tail latency. Predicting packet-level latency at the sender enables marking packets within a flow to be put into different queues to reduce latency.

In practice, L4S routers are designed to respect ECN markings. Thy are configured to handle both classic flows (i.e., non-L4S flows with packets are marked *non-ECT*) and L4S flows (packets marked as *ECT*). Once packets of a flow are marked, every L4S-enabled router downstream respects the packet marking. L4S traffic is sent using DCTCP based TCP Prague, with all packets are marked as ECT. L4S enabled routers at every hop, ensure that ECT marked traffic is placed on the L4S queue and all other traffic is placed in the classic queue. If a router does not have L4S support all traffic is placed in the classic queue.

For a set of L4S flows from a sender, SwiftQueue predicts sharp latency spikes for future packets based on the learnt effects of packet interaction. Based on the prediction, SwiftQueue's controller makes a simple decision in order to reduce the queuing latency on the L4S queue on the bottleneck router. It changes the individual packets within the flow which

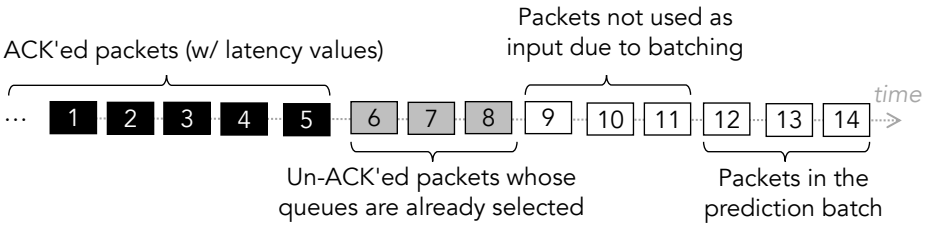


Figure 6 An illustration of two main ideas. (1) SwiftQueue speeds up inference by batching, *e.g.,* Prediction of Packet 12-14 is done simultaneously as a batch, though the prediction delay precludes the use of the information between Packet 9-11. (2) SwiftQueue prevents prediction-driven oscillation by incorporating un-ACK'ed packets in the prediction input, *e.g.,* although Packet 6-8 do not have latency values yet, but by incorporating their queue selection in the input, the predictor knows how many packets have been assigned to each queue.

have high predicted latency to have non-ECN markings in the IP header. These packets will then be put on the less-congested classic queue on the bottleneck router. This reduces the queuing latency experienced by these L4s flows.

We acknowledge that if *both* queues are at capacity, changing the queue will not reduce latency. For this paper, we focus on reducing the tail latency for L4S flows, when the classic queue is relatively less congested. We also do not change the router behavior or modify the sending congestion control (CC) logic. Instead, we only perform queue selection. As a result, SwiftQueue's reduction of tail latency should have little negative effect on throughput.

4.2.1 Fast queue selection for in-time predictions

In order to use the SwiftQueue's predictions for L4S queue selection, the predictions need to be *fast enough*. This means we need to make the decision before it is time to send out the packet. High link speeds and data rates require an extremely fast inference (for both prediction and control logic). Waiting for the last packet to make a prediction is not always scalable at the packet level with increasing link speeds.

To solve this, we make *batched* predictions. Like in standard Transformers, batching increases the throughput with a negligible increase in inference delay because running the model on one input does not fully utilize the compute resource. We make latency predictions for a future batch of packets, given the latency values of the preceding packets. By increasing the batch size for predictions, the inference latency at the per-packet level is reduced. Since the prediction for a future packet is made a few packets earlier, SwiftQueue's controller has time to account for the inference delay and the control logic, in order to make a decision for the packet before it is sent out. Figure 6 shows an illustrative example. We need to make the prediction for *packet 12-14*, as one prediction batch, by seeing up-to *packet 8* in order to ensure we send out *packet 12* in time.

However increasing the batch size beyond a point does come at the cost of reduced prediction quality. SwiftQueue's current implementation finds a balance point where it supports a large enough prediction batch for in-time predictions on fast ISP links, without significant drop in quality. Currently SwiftQueue's predictor is fast enough to make predictions for real networks like ISPs etc. running its typical applications (e.g. YouTube video streaming at 1080p). We can predict upto a batch size of packets (upto 8 future packets) within $200-400$ μs of inference latency without noticeable reduction prediction quality. This is fast enough for predictions and decisions in time for link speeds of upto 200 Mbps.

4.2.2 Preventing oscillation in selected queues

Finally, pure prediction-driven control often leads to unstable outcomes. It is possible that SwiftQueue's queue selection controller can incorrectly select a particular queue for a packet if it only uses information from packets which have been ACKed. If SwiftQueue has not received enough recent acknowledgments (ACKs) for the previously sent out packets, it may not select the queue correctly as it does not have the most recent network state information. This can cause SwiftQueue to overfill a particular queue and increase the latency.

To prevent this, SwiftQueue's queue selection controller additionally uses a combination of (1) latency values of packets that have been ACK'ed and (2) packets that have been marked with queue-selection decisions but have not been ACK'ed. A combined view of these factors helps SwiftQueue decide if the packet should be finally put on a particular queue in case the previous latency values are missing. We illustrate this in Figure 6. For instance, SwiftQueue's queue selection for *packet 12* looks at both past packets with latency values (*packet 1* - *packet 5*) *and* at the queue selection on un-ACK'ed packets (*packet 6* - *packet 8*) in order to select the queue for the given packet. Even though we do not know the latency of the un-ACK'ed packets, they still provide vital information about at least how many packets have been assigned to each queue. This information is helpful for SwiftQueue to prevent oscillation in its queue selection decisions.

5 Implementation

We implemented SwiftQueue's predictor and L4S queue selector in 1500 lines of Python code using PyTorch [72] and Lightning [22]. We build on an existing NS-3 official implementation of L4S [34]. The L4S applications within NS-3 were implemented in 500 lines C++ code.

For training and inference we use a single TITAN RTX 3080 with 10 GB memory with a peak memory utilization of 2GB. This is *much cheaper and less powerful* compared to GPUs used to train Transformers today. For comparison, LLMs today use multiple H100 GPUs for training, and even a single H100 GPU has $10-12\times$ higher memory and has $> 100\times$ higher price than what we use to run SwiftQueue.

As we train for prediction tasks, we only use the Transformer Encoder layers to learn representation similar to the literature on Transformers for time-series as introduced in Section 2.2. We use the ADAM [47] optimizer with its default parameters of $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10^{-9}$.

We use dropout rate of 0.2 for both the Transformer and the linear layers and ReLU [33] activation for non-linearity. We also employ a weight delay of $1\epsilon^{-5}$ to help with regularization and reduce over-fitting. We vary the learning rate based on the original Transformer paper [89] as $l\_rate = d_{model}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$ with setting $warmup\_steps = 2000$, based on empirical testing.

We have an initial pre-training time of 5 - 10 minutes on the real traces collected in Table 1. We do not optimize the pre-training time as it is a one-time cost. We instead focus on fine-tuning and inference speeds, as they are of greater, practical significance since the models need to be frequently updated (more details in §6).

Finally, we use ***separate*** data for training, fine-tuning, and testing, *i.e.,* no data pollution, and they are ***aligned*** across SwiftQueue and its baselines (§6). We first pre-train SwiftQueue and the baselines on the first 5 minutes of the traces. We then test them on the next 5 minutes of the trace. At every 5 minute intervals, we fine-tune the models on the preceding 1 minute of the trace, and then again test on the next 5 minutes till the end of the trace.

Table 1 Summary of the collected traces on the WiFi and Ethernet endpoints.

Trace Name	Description	Median Latency (ms)	Trace Duration (min)	Flow Duration (s)	# flows in trace
video_wifi	Netflix traffic captured on a WiFi endpoint	70	30	29 - 386	28
web_wifi	Facebook traffic captured on a WiFi endpoint	66	30	27 - 172	42
video_eth	Netflix traffic captured on an Ethernet endpoint	74	30	48 - 362	31
web_eth	Facebook traffic captured on an Ethernet endpoint	57	30	35 - 212	47

6 Evaluation

The key takeaways from our evaluation are

- SwiftQueue predicts *sharp changes* in latency more accurately than other machine-learning baselines by 45-65% across a variety of real network and simulated traces.
- SwiftQueue achieves lower P99 tail latency on L4S flows from a given sender by 36-45% compared to the default L4S queue selection logic as well as queue selected by other latency predictors.
- SwiftQueue allows frequent fine-tuning to keep the model up to date over changing network conditions and makes latency prediction fast enough at a line speed of 200Mbps.

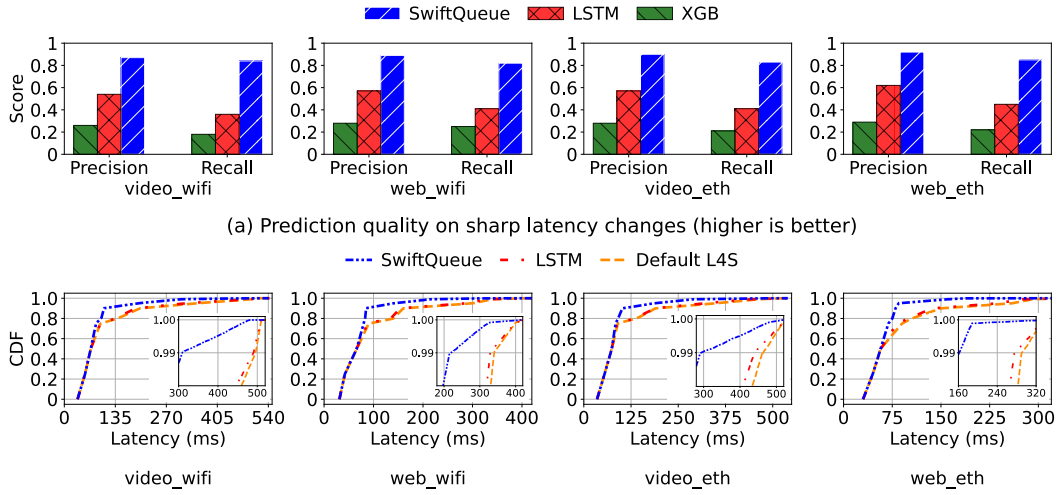
Baselines

We choose the following baselines as they were *state-of-the-art* for time-series prediction tasks prior to the introduction of Transformers.

- **LSTM:** We use a state-of-the-art BiLSTM [81] model known used in the past for its sequence prediction capabilities. For fairness, we align the LSTM model to have comparable size (parameter count) as SwiftQueue (albeit having much higher inference and fine-tuning times).
- **XGBoost:** We use a state-of-the-art classical ML regression model XGBoost(XGB) [12], which has been been used in the past for certain sequence prediction tasks using sliding windows of input sequences [13].

Evaluation Metrics

We evaluate the prediction of SwiftQueue and the baselines on the packets where there is a sharp change in latency. We define a *sharp change* as a change in latency between two consecutive packets of *at least 20 milliseconds* and a *relative change of at least* 20% as introduced in §4.1. We use *precision*, *recall* and *F1-score* to measure prediction quality at these latency changes. We define success if both the true latency and the model's predicted latency have a *sharp change* in the same direction. We report the prediction quality only over the packets that have these sharp changes in latency. For the tail-latency improvement, we measure the % reduction in P99 queuing latency. We refer to this metric from now on as *P99 Reduction*.



(a) Prediction quality on sharp latency changes (higher is better)

(b) L4S Tail-Latency with different queue selection methods (lower is better)

Figure 7 SwiftQueue achieves better prediction on sharp changes by 45-65% and has higher tail-latency (P99) reduction for L4S flows by 36-45%, compared to other predictor-driven queue selection or L4S's default strategy.

6.1 Evaluating SwiftQueue's predictions and L4S queue selection on real network traces

Real-world traces collection

We capture Netflix traffic (video streaming application) on a Home-WiFi network and on a wired Ethernet network. We also collect Facebook browsing traffic (social media web traffic application) on the same Home-WiFi network and Ethernet network. All of the flows captured are from within the same browsing session and we define a flow as the standard 5 tuple. We use self-collected ISP traces as popular public traces like CAIDA [9] or M-LAB [62] may not have all the features required for us. All traffic is collected on the same end-point device and the available bandwidths on the WiFi and Ethernet links were measured to vary between 50-100 Mbps. Table 1 shows the details of the collected traces.

L4S queue selection

We use trace-driven simulation in NS-3 to evaluate the performance of SwiftQueue's L4S queue selection. We use statistics such as trace duration, inter-packet gaps, packet sizes etc. from the real traces in Table 1.

We use an L4S implementation in NS3 with a dual-queue setup as specified by the authors in [78]. We setup a L4S enabled topology as shown in Figure 8. We have a *6 L4S flows* along with *4 Classic TCP Cubic flows* bound from Sender 1(S1) to destination Receiver 1(R1) . We have *10* L4S flows from different senders creating cross traffic (unseen by SwiftQueue at S1). All L4S flows share the same bottleneck L4S queue on the router.

By default, all packets from all L4S flows will be put on the L4S queue on Router (Default L4S). We use SwiftQueue's predictor to predict the latency of next packets at *S1* and change the ECN marking for the packets with high latency. These L4S flow packets from sender *S1* are put on the classic queue. As there is a large number of L4S flows, they experience increased queuing latency on the L4S queue. We *focus* on reducing the tail latency for the L4S flows sent out by Sender *S1*. We reduce the tail latency of packets for the L4S flows bound for R1 by proactively changing the queue selection to reduce the queuing latency.

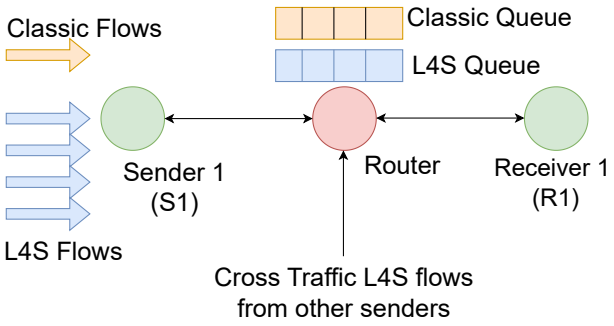


Figure 8 L4S queue selection topology in NS-3 with both L4S and Classic flows.

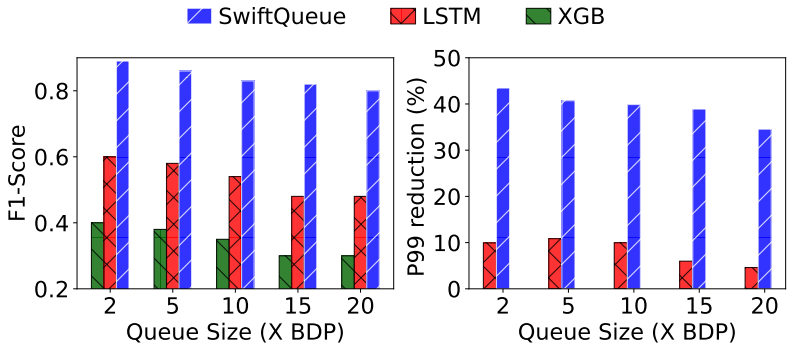


Figure 9 Impact of the bottleneck queue size on prediction accuracy and tail latency reduction.

For measuring the improvement of packet-level queue selection, we report the *P90 and P99 latency* for the L4S flows. We compare with the default L4S selection (and not other AQM techniques) as we don't change the L4S logic within the router. We only change the selection logic at the sender and do not adapt the congestion control end-point to modify its sending rate.

Result #1 – SwiftQueue can predict sharp latency changes better compared to the baselines. Figure 7 (a) shows the performance of the SwiftQueue relative to the LSTM and XGB baselines in predicting the sharp end-to-end-latency changes for the next packet. SwiftQueue predicts the sharp changes in latency much better than the LSTM and XGB baselines, consistently being *45-65%* higher values times better for all the real-world traces collected, in terms of both precision and recall.

Result #2 – SwiftQueue's prediction driven L4S queue selection achieves lower tail latency than the baselines. Figure 7 (b) shows the improved tail-latency reduction of the L4S flows by SwiftQueue's prediction based queue selection. SwiftQueue's queue selection achieves *36-45%* P99 Reduction over the LSTM baseline and the default L4S selection.

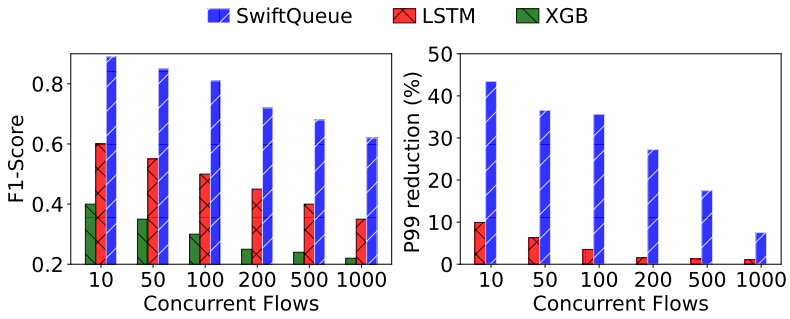


Figure 10 Impact of the number of flows on prediction accuracy and tail latency reduction.

6.2 Evaluating SwiftQueue's predictions on NS-3 simulated data

Evaluation Setup

To further evaluate the performance of SwiftQueue's predictions on packet-level interactions, we use NS-3 [76] to simulate workloads for training and evaluating on varying network conditions. We reuse the three-node topology (one sender, router and receiver) as shown in Figure 8 for our experiments. We only use TCP flows for our evaluation as we need a reliable, connection-oriented protocol. While the topology is simple, we create complex packet and flow interactions.

The TCP flows are created using messages sampled from real world distributions [64]. We use the web search workload for DCTCP [2] and the Hadoop cluster workload at Facebook [77]. Our default environment runs 10 concurrent TCP flows using DCTCP+L4S and 4 additional classic TCPCubic flows (for cross traffic). This is a realistic setup, as in practice L4S flows do not usually exist without other Classic flows on the network. We use a default propagation delay of 20ms and a link bandwidth of *100Mbps*, reflecting a typical ISP connection. Each flow is started with a base sending rate *5Mbps*. We use a default queue length of 2×BDP. To ensure we have enough packet interactions, flows are started in a flow start window, according to a uniform random distribution. This ensures the flows start at different times leading to greater interaction between flow start times.

We perform a sensitivity analysis to measure the performance of SwiftQueue in terms of prediction quality and P99 latency reduction for the L4S flows. We evaluate along the following dimensions. We change one dimension at a time, keeping all others the same as our default environment.

- **Queue size:** We increase the queue size on the bottleneck router as 2, 5, 10, 15, and 20 × the *BDP* respectively.
- **# of concurrent flows:** We increase the number of TCP flows using DCTCP+L4S as 10, 50, 100, 200, 500 and 1000.
- **Propagation delay:** We increase the propagation delay of the link as 20, 40, 100, 150 and 200.
- **CC algorithm:** We change our 10 concurrent TCP flows to run DCTPC+L4S, TCPCubic and TCPBBR. We choose them as they represent loss-based [32] and delay-based [10] CC algorithms that are deployed today.

  We compare SwiftQueue's performance against the baselines on 5 minutes of simulated traffic. We don't use the XGB model to perform packet level queue selection as the prediction quality on the *sharp changes* is extremely low.

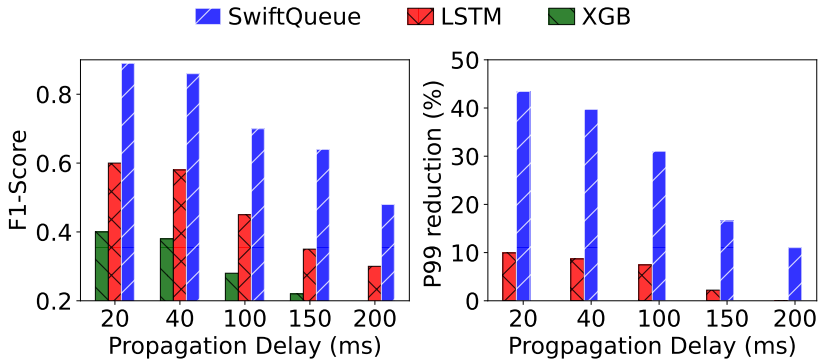


Figure 11 Impact of the end-to-end propagation delay on prediction accuracy and tail latency reduction.

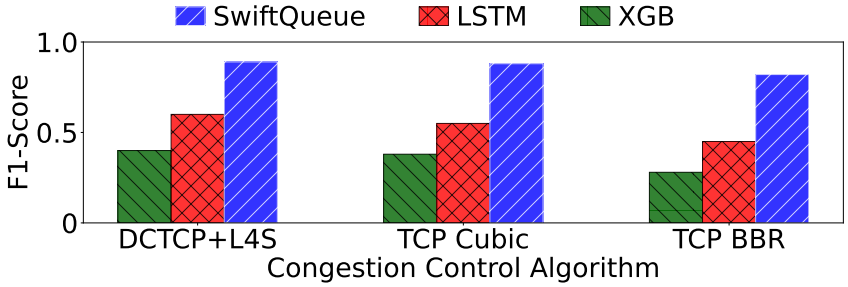


Figure 12 SwiftQueue's latency prediction accuracy under different congestion control logics.

Result – SwiftQueue achieves better predictions changing new network conditions and obtains lower P99 latency on the L4S flows

We compare the prediction quality of SwiftQueue with that of the baselines over all the individual environments. We use F1-score on the *sharp changes* as a measure of the prediction quality. We use the % decrease in P99 latency on the L4S flows, relative to the default setting of queue selection at the flow level in order to compare the performance of SwiftQueue's queue selection controller.

First, in Figure 9, we measure the effect of changing the max queue. SwiftQueue performs better than both the baselines in terms of higher F1-score by *29-62%* across all the environments and also achieves a greater P99 Reduction by *33-35%* across all environments. SwiftQueue is not sensitive to increasing queue size for a given number of concurrent flows and can continue to perform well for both prediction quality and latency reduction. With a given number of flows, changing queue size doesn't increase the complexity of packet interactions over time due to fewer, new flow starts.

Second, in Figure 10, we measure the effect of changing the number of concurrent flows on the performance of SwiftQueue. SwiftQueue performs better than both the baselines in terms of higher F1-score of *29-60%* across all the environments. SwiftQueue also achieves a greater P99 Reduction by *10-35%* across all environments. We expect the performance of SwiftQueue to be slightly sensitive, with increasing number of flows as interactions can get extremely complex. However, even upto 100s of concurrent flows, SwiftQueue provides useful latency prediction and P99 latency reduction.

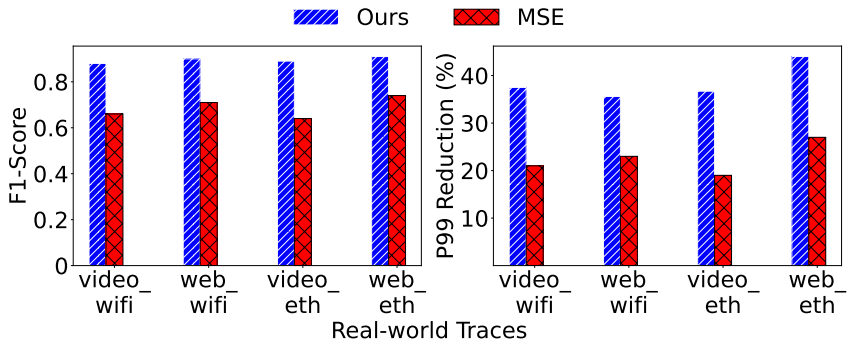


Figure 13 Compared with the standard loss function of MSE, training the Transformer with our custom loss function (§4.1.1) yields much higher prediction accuracy of sharp latency changes.

Third, in Figure 11, we measure the effect of changing the link's propagation delay on the performance of SwiftQueue. SwiftQueue performs better than both the baselines in terms of higher F1-score by *29-65%* across all the environments and achieves a greater P99 Reduction by across all environments. SwiftQueue is sensitive to increasing propagation delay on the link beyond a point for a given number of concurrent flows. With increasing propagation delays, the ACKs received at the sender are often delayed and in such cases, SwiftQueue may not have enough past context to predict accurately or perform queue selection. However, even at high propagation delays of 200 ms, SwiftQueue still has observable gains and does not drop quality or increase latency.

Finally, in Figure 12, we measure the effect of only using a single TCP CC algorithm. We see that across several popular TCP CC algorithms, SwiftQueue has better performance in terms of prediction F1-score by *25-49%*. We observed that TCP BBR flows have slightly more random interactions and the prediction F1-score is slightly lower (about 4-5%). We feel this happens due to the probing nature of TCPBBR, which sends out special probing packets. SwiftQueue currently cannot distinguish between such control packets explicitly and we understand this might be a potential reason for the reduced quality. Finally, we don't evaluate the P99 latency reduction for TCPCubic and TCPBBR as they cannot be configured to enable packet-level queue selection using ECN markings

SwiftQueue makes many correct predictions for sharp latency changes when flows compete for the available bandwidth after they starts. We acknowledge that some latency changes are fundamentally unpredictable (*e.g.,* start of a random flow) and SwiftQueue can not predict those. Instead, SwiftQueue focuses on learning interactions between competing flows on a shared link.

6.3 Component-wise analysis

Benefit of changing the loss function

We measure the effect of changing the loss function from the standard MSE to our change-point based loss function to focus on *sharp changes* (introduced in 4.2). While the MSE is more optimal for learning values over the entire trend, it achieves a lower F1-score at actually predicting the sharp changes in latency. In Figure 13 we see that SwiftQueue achieves a higher F1-score at predicting these values across all of the real trace datasets by *17-25%*.

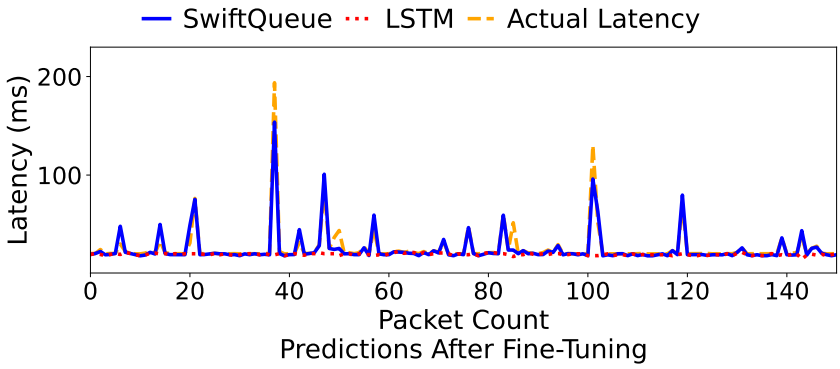


Figure 14 Visualizing the prediction of SwiftQueue (with fine-tuning) on an example of 150 consecutive packets from the video_wifi trace, compared with their actual latency values and the prediction of the most competitive baseline (LSTM).

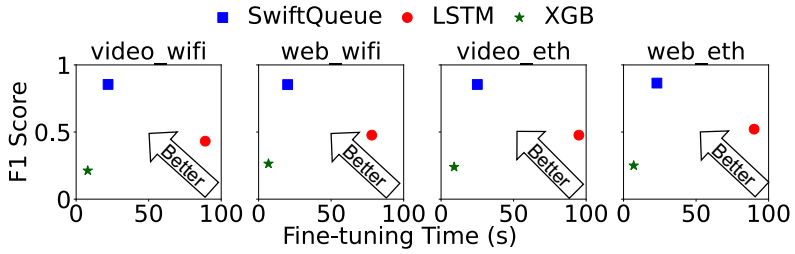


Figure 15 SwiftQueue's prediction is more accurate (F1 score) than the baselines with a fine-tuning delay below 30 seconds. (LSTM and SwiftQueue use the same hardware.)

Performance gains obtained from fine-tuning

Fine-tuning significantly helps improve the performance on SwiftQueue. On real network traces, SwiftQueue doesn't perform well without continuous fine-tuning. We observe this in Figure 14 where after fine-tuning, SwiftQueue can capture both the *sharp changes* in latency and also predict latency values which are closer to the *true_value*. We fine-tune the *linear layers* of SwiftQueue (and the LSTM). To measure the prediction quality, we measure the F1-score (combining precision and recall) on the predicted values of the sharp latency changes. SwiftQueue is consistently better by *45-65%* times as compared to both baselines as seen in Figure 15 in terms of prediction quality. SwiftQueue is also significantly faster to fine-tune, as compared to the LSTM on the same single TITAN RTX 3080 GPU by *3.8-4.4*×. While fine-tuning the XGB model on the CPU is faster, the prediction quality is too low to be useful packet-level queue selection model.

Effect of choosing the fine-tuning interval

Choosing the right fine-tuning interval is important for SwiftQueue to obtain the best prediction quality. While our fine-tuning speed for SwiftQueue is fast, updating the model very frequently (e.g. every 10s of seconds) is computationally expensive (as we still use one GPU) and maybe unnecessary. In Figure 16, we measure the effect of the fine-tuning interval on the prediction accuracy. We see that fine-tuning at intervals of upto 5 minutes has negligible drop in prediction quality for our traces. Hence we don't need to use the GPU resource to fine-tune at a very short intervals if the gains are marginal.

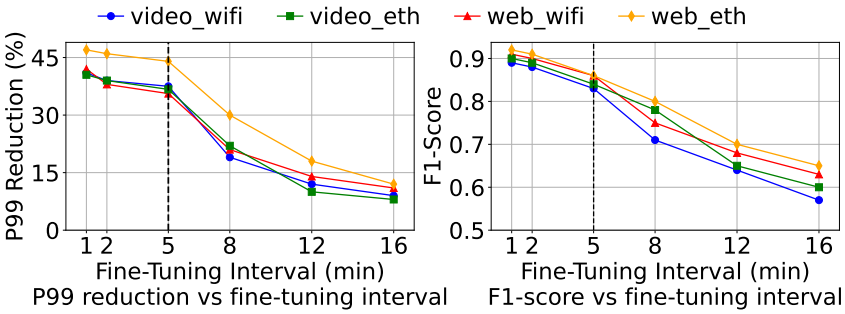


Figure 16 Latency reduction and prediction F1-score (precision + recall) generally drops with increasing fine-tuning interval, with the choice of fine-tuning interval of 5 minutes still achieving close-to-optimal latency reduction.

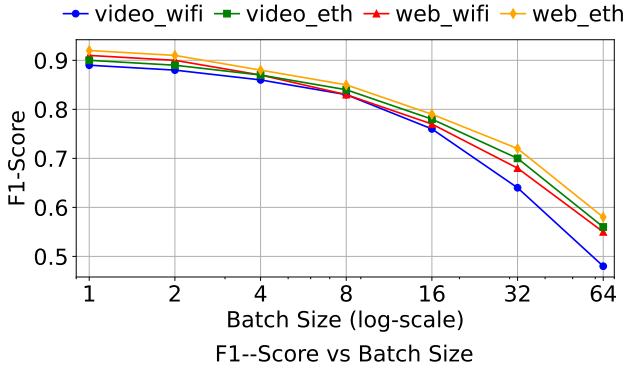


Figure 17 Prediction F1-score (precision + recall) drops with increasing batch size (more parallel prediction), with batch of 8 achieving a favorable tradeoff between parallel prediction and prediction accuracy.

Batch size vs prediction quality

We make SwiftQueue's inference fast enough by predicting the next batch of latency values in order to make the decision in time before the next packet is sent out. We do not want to wait till we receive the previous latency as it might be too late. Increasing the prediction batch size makes it possible to predict the latency value a few packets earlier. However increasing the batch size comes at the cost of potentially reduced prediction quality. In Figure 17 we measure the prediction F1-score across increasing batch sizes and see that increasing the batch size excessively leads to a drop in prediction quality. Across our workloads a batch size of 8, allows for fast inference for ISP link speeds with a marginal drop in prediction quality.

7 Discussion and Limitations

SwiftQueue for applications beyond L4S queue selection. Currently L4S is chosen as an application as its *static flow-level queue assignment* architecture can significantly benefit from a latency prediction model like SwiftQueue, in order to meet its low-latency targets. In addition, SwiftQueue currently performs queue selection but does not modify the sender's TCP CC algorithm (L4S currently is limited to work with DCTCP/TCP Prague only).

We hope to inspire more research to optimize latency prediction and CC jointly. We see the potential to use the predictor to develop new proactive CC algorithms, which adapt to real-time network conditions beyond rule-based algorithms. We also envision future research with SwiftQueue to extend to more applications like video-streaming, which can benefit from real-time network latency predictions for smoother use Quality-of-Experience.

Excessively delayed ACKs, high traffic volume. If preceding ACKs are excessively delayed (*e.g.,* due to a high network propagation delay), SwiftQueue cannot predict the next latency values accurately as it will not have enough past latency values to do so. If such a situation is detected (*e.g.,* using timeout windows), one solution for SwiftQueue might be to fall back to default L4S queue selection. Similarly, if the number of concurrent flows significantly scales up (*e.g.,* beyond 10000s), the current SwiftQueue Transformer architecture may also need to be scaled up to handle more complex inter-packet interactions.

GPU hardware requirement and costs. SwiftQueue currently needs a GPU, albeit a cheap one (§5), for low inference speed and may need a higher-end GPU to make in-time prediction for ultra-high bandwidth (*e.g.,* 10s of Gbps links). Running inference on GPUs does have additional costs and more research is needed to make SwiftQueue go beyond L4S applications at consumer-degree ISPs, which has been the focus of this paper.

One method we envision for reducing GPU costs is integration with recent outcomes on network FPGA and hardware research which allows running ML models and making predictions at line-rate on low cost hardware [85, 102]. We leave the integration of SwiftQueue with hardware beyond GPUs for future work.

Overall, while SwiftQueue, as a first step, may not provide packet-level prediction and queue selection capabilities *universally across all networks*, it opens up new avenues of research for better Transformer-based packet-level analysis and optimizing more applications to benefit from it.

8 Conclusion

We present SwiftQueue, which uses a Transformer for packet-level latency prediction and performs L4S-based packet-level queue selection to mitigate high tail latency. Using real traces and simulated ones, we show that SwiftQueue outperforms all baselines at predicting sharp latency changes, and driven by the accurate latency prediction, SwiftQueue achieves greater tail latency reduction for L4S flows.

References

1 Aristide Tanyi-Jong Akem, Michele Gucciardo, Marco Fiore, et al. Flowrest: Practical flow-level inference in programmable switches with random forests. In *IEEE International Conference on Computer Communications*, 2023.

2 Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1851182.1851192.

3 Ons Aouedi, Kandaraj Piamrat, and Benoît Parrein. Performance evaluation of feature selection and tree-based algorithms for traffic classification. In *2021 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 1–6. IEEE, 2021. doi:10.1109/ICCWORKSHOPS50388.2021.9473580.

4 S. Bauer, J. Kneip, T. Mlasko, B. Schmale, J. Vollmer, A. Hutter, and M. Berekovic. The mpeg-4 multimedia coding standard: Algorithms, architectures and applications. *J. VLSI Signal Process. Syst.*, 23(1):7–26, 1999. doi:10.1023/A:1008136602091.

5 Jean-Yves Le Boudec, William Courtney, Jon Bennett, Shahram Davari, Dimitrios Stiliadis, Kent Benson, Victor Firoiu, Dr. Bruce S. Davie, and Anna Charny. An Expedited Forwarding PHB (Per-Hop Behavior). RFC 3246. doi:10.17487/RFC3246.

6 Yanitsa Boyadzhieva. Verizon targets next wave of time-critical 5g apps with l4s. Technical report, TelecomTV, 2024. URL: https://www.telecomtv.com/content/5g/verizon-targets-next-wave-of-time-critical-5g-apps-with-l4s-49601/.

7 Bob Briscoe, Koen De Schepper, Marcelo Bagnulo, and Greg White. Low Latency, Low Loss, and Scalable Throughput (L4S) Internet Service: Architecture. RFC 9330. doi:10.17487/RFC9330.

8 Jacob Brown, Xi Jiang, Van Tran, Arjun Nitin Bhagoji, Nguyen Phong Hoang, Nick Feamster, Prateek Mittal, and Vinod Yegneswaran. Augmenting rule-based dns censorship detection at scale with machine learning. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 3750–3761, 2023.

9 caida. Anonymized Internet Traces 2016. https://catalog.caida.org/dataset/passive_2016_pcap. Dates used: <date(s) used>. Accessed: <date accessed>.

10 Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: congestion-based congestion control. *Commun. ACM*, 60(2):58–66, January 2017. doi:10.1145/3009824.

11 Ching Chang, Wei-Yao Wang, Wen-Chih Peng, and Tien-Fu Chen. Llm4ts: Aligning pre-trained llms as data-efficient time-series forecasters, 2024. arXiv:2308.08469.

12 Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2939672.2939785.

13 Yige Chen, Tianning Zang, Yongzheng Zhang, Yuan Zhou, and Yipeng Wang. Rethinking encrypted traffic classification: A multi-attribute associated fingerprint approach. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*, pages 1–11, 2019. doi:10.1109/ICNP.2019.8888043.

14 Mitchell Clark. The quiet plan to make the internet feel faster. Technical report, The Verge, 2023. URL: https://www.theverge.com/23655762/l4s-internet-apple-comcast-latency-speed-bandwidth.

15 Cloudbase. Cloud gaming latency: What it is and how to reduce it. Technical report, Cloudbase, 2024.

16 Comcast. Comcast introduces nation's first ultra-low lag xfinity internet experience with meta, nvidia, and valve, 2025. URL: https://corporate.comcast.com/press/releases/comcast-introduces-nations-first-ultra-low-lag-xfinity-internet-experience-with-meta-nvidia-and-valve.

17 Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019. arXiv:1810.04805.

18 Alexander Dietmüller, Siddhant Ray, Romain Jacob, and Laurent Vanbever. A new hope for network model generalization. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, HotNets '22, pages 152–159, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3563766.3564104.

19 Abhimanyu Dubey et al. The llama 3 herd of models, 2024. doi:10.48550/arXiv.2407.21783.

20 Hugo Touvron et al. Llama 2: Open foundation and fine-tuned chat models, 2023. doi:10.48550/arXiv.2307.09288.

21 Gorry Fairhurst and Michael Welzl. The Benefits of Using Explicit Congestion Notification (ECN). RFC 8087. doi:10.17487/RFC8087.

22 William Falcon and The PyTorch Lightning team. PyTorch Lightning, March 2019. doi:10.5281/zenodo.3828935.

23 S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993. doi:10.1109/90.251892.

24 Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann Dauphin. Convolutional sequence to sequence learning. *ArXiv*, abs/1705.03122, 2017. arXiv:1705.03122.

25 Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. SIMON: A simple and scalable method for sensing, inference and measurement in data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 549–564, Boston, MA, February 2019. USENIX Association. URL: https://www.usenix.org/conference/nsdi19/presentation/geng.

26 Jim Gettys and Kathleen Nichols. Bufferbloat: Dark buffers in the internet: Networks without effective aqm may again be vulnerable to congestion collapse. *Queue*, 9(11):40–54, November 2011. doi:10.1145/2063166.2071893.

27 Fengchen Gong, Divya Raghunathan, Aarti Gupta, and Maria Apostolaki. Zoom2net: Constrained network telemetry imputation. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, pages 764–777, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3651890.3672225.

28 Prateesh Goyal, Akshay Narayan, Frank Cangialosi, Srinivas Narayana, Mohammad Alizadeh, and Hari Balakrishnan. Elasticity detection: a building block for internet congestion control. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, pages 158–176, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3544216.3544221.

29 Daniel B. Grossman. New Terminology and Clarifications for Diffserv. RFC 3260. doi:10.17487/RFC3260.

30 Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. *SIGCOMM Comput. Commun. Rev.*, 45(4):139–152, August 2015. doi:10.1145/2829988.2787496.

31 Satyandra Guthula, Navya Battula, Roman Beltiukov, Wenbo Guo, and Arpit Gupta. netfound: Foundation model for network security, 2023. doi:10.48550/arXiv.2310.17025.

32 Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, July 2008. doi:10.1145/1400097.1400105.

33 Richard Hahnloser and H. Sebastian Seung. Permitted and forbidden sets in symmetric threshold-linear networks. In T. Leen, T. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems*, volume 13. MIT Press, 2000. URL: https://proceedings.neurips.cc/paper_files/paper/2000/file/c8cbd669cfb2f016574e9d147092b5bb-Paper.pdf.

34 Tom Henderson. NS-3 L4S Dual Queue, 2019. URL: https://gitlab.com/tomhenderson/ns-3-dev/-/tree/dual-queue-coupled-aqm?ref_type=heads.

35 Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997. doi:10.1162/neco.1997.9.8.1735.

36 Toke Høiland-Jørgensen, Paul McKenney, dave.taht@gmail.com, Jim Gettys, and Eric Dumazet. The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm. RFC 8290. doi:10.17487/RFC8290.

37 Sunghwan Ihm and Vivek S. Pai. Towards understanding modern web traffic. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, pages 295–312, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2068816.2068845.

38 Steffen Illium, Thore Schillman, Robert Müller, Thomas Gabor, and Claudia Linnhoff-Popien. Empirical analysis of limits for memory distance in recurrent neural networks. In *Proceedings of the 14th International Conference on Agents and Artificial Intelligence*, pages 308–315. SCITEPRESS - Science and Technology Publications, 2022. doi:10.5220/0010818500003116.

39 Kleidi Ismailaj, Miguel Camelo, and Steven Latré. When deep learning may not be the right tool for traffic classification. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 884–889. IEEE, 2021. URL: `https://ieeexplore.ieee.org/document/9463927`.

40 Xi Jiang, Shinan Liu, Aaron Gember-Jacobson, Arjun Nitin Bhagoji, Paul Schmitt, Francesco Bronzino, and Nick Feamster. Netdiffusion: Network data augmentation through protocol-constrained traffic generation. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 8(1):1–32, 2024. `doi:10.1145/3639037`.

41 Xi Jiang, Shinan Liu, Aaron Gember-Jacobson, Paul Schmitt, Francesco Bronzino, and Nick Feamster. Generative, high-fidelity network traces. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, pages 131–138, 2023. `doi:10.1145/3626111.3628196`.

42 Xi Jiang, Shinan Liu, Saloua Naama, Francesco Bronzino, Paul Schmitt, and Nick Feamster. Ac-dc: Adaptive ensemble classification for network traffic identification. *arXiv preprint arXiv:2302.11718*, 2023. `doi:10.48550/arXiv.2302.11718`.

43 Dongzi Jin, Yiqin Lu, Jiancheng Qin, Zhe Cheng, and Zhongshu Mao. Swiftids: Real-time intrusion detection system based on lightgbm and parallel intrusion detection mechanism. *Computers & Security*, 97:101984, 2020. `doi:10.1016/J.COSE.2020.101984`.

44 Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery. `doi:10.1145/2391229.2391238`.

45 Subina Khanal, Seshu Tirupathi, Giulio Zizzo, Ambrish Rawat, and Torben Bach Pedersen. Domain adaptation for time series transformers using one-step fine-tuning, 2024. `doi:10.48550/arXiv.2401.06524`.

46 Tae-Young Kim and Sung-Bae Cho. Web traffic anomaly detection using c-lstm neural networks. *Expert Systems with Applications*, 106:66–76, 2018. `doi:10.1016/J.ESWA.2018.04.004`.

47 Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. `arXiv:1412.6980`.

48 Cyrill Krähenbühl, Seyedali Tabaeiaghdaei, Simon Scherrer, Matthias Frei, and Adrian Perrig. Toward global latency transparency. In *2024 IFIP Networking Conference (IFIP Networking)*, pages 536–542, 2024. `doi:10.23919/IFIPNetworking62109.2024.10619858`.

49 Jim Kurose. On computing per-session performance bounds in high-speed multi-hop computer networks. *SIGMETRICS Perform. Eval. Rev.*, 20(1):128–139, 1992. `doi:10.1145/149439.133097`.

50 Nokia Bell Labs. L4s. Technical report, Nokia Bell Labs, 2023. URL: `https://www.bell-labs.com/research-innovation/projects-and-initiatives/l4s/#gref`.

51 Cing-Han Lai, Chao-Tung Yang, Endah Kristiani, Jung-Chun Liu, and Yu-Wei Chan. Using xgboost for cyberattack detection and analysis in a network log system with elk stack. In *Frontier Computing: Theory, Technologies and Applications (FC 2019) 8*, pages 302–311. Springer, 2020.

52 Chenning Li, Arash Nasr-Esfahany, Kevin Zhao, Kimia Noorbakhsh, Prateesh Goyal, Mohammad Alizadeh, and Thomas E. Anderson. m3: Accurate flow-level performance estimation using machine learning. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, pages 813–827, New York, NY, USA, 2024. Association for Computing Machinery. `doi:10.1145/3651890.3672243`.

53 Jinyang Li, Zhenyu Li, Ri Lu, Kai Xiao, Songlin Li, Jufeng Chen, Jingyu Yang, Chunli Zong, Aiyun Chen, Qinghua Wu, Chen Sun, Gareth Tyson, and Hongqiang Harry Liu. Livenet: a low-latency video transport network for large-scale live streaming. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, pages 812–825, New York, NY, USA, 2022. Association for Computing Machinery. `doi:10.1145/3544216.3544236`.

54 Shiyang Li, Xiaoyong Jin, Yao Xuan, Xiyou Zhou, Wenhu Chen, Yu-Xiang Wang, and Xifeng Yan. Enhancing the locality and breaking the memory bottleneck of transformer on time series forecasting. *Advances in neural information processing systems*, 32, 2019.

55 Shuai Li, Wanqing Li, Chris Cook, Ce Zhu, and Yanbo Gao. Independently recurrent neural network (indrnn): Building a longer and deeper rnn, 2018. arXiv:1803.04831.

56 Bryan Lim, Sercan Ö Arık, Nicolas Loeff, and Tomas Pfister. Temporal fusion transformers for interpretable multi-horizon time series forecasting. *International Journal of Forecasting*, 37(4):1748–1764, 2021.

57 Haitao Liu and Haifeng Wang. Real-time anomaly detection of network traffic based on cnn. *Symmetry*, 15(6):1205, 2023. doi:10.3390/SYM15061205.

58 Jason Livingood. ISP Dual Queue Networking Deployment Recommendations. Internet-Draft draft-livingood-low-latency-deployment-05, Internet Engineering Task Force, Work in Progress. URL: https://datatracker.ietf.org/doc/draft-livingood-low-latency-deployment/05/.

59 Mohammad Lotfollahi, Mahdi Jafari Siavoshani, Ramin Shirali Hossein Zade, and Mohammdsadegh Saberian. Deep packet: A novel approach for encrypted traffic classification using deep learning. *Soft Computing*, 24(3):1999–2012, 2020. doi:10.1007/S00500-019-04030-2.

60 Matthew Luckie, Young Hyun, and Bradley Huffaker. Traceroute probe method and forward ip path inference. In *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement*, IMC '08, pages 311–324, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1452520.1452557.

61 Harsha V. Madhyastha, Thomas Anderson, Arvind Krishnamurthy, Neil Spring, and Arun Venkataramani. A structural approach to latency prediction. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, IMC '06, pages 99–104, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1177080.1177092.

62 Measurement Lab. The M-Lab NDT data set. https://measurementlab.net/tests/ndt, (2009-02-11 – 2015-12-21). Bigquery table measurement-lab.ndt.download.

63 Xuying Meng, Chungang Lin, Yequan Wang, and Yujun Zhang. Netgpt: Generative pretrained transformer for network traffic, 2023. doi:10.48550/arXiv.2304.09513.

64 Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: a receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 221–235, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3230543.3230564.

65 Douglas C. Montgomery. *Introduction to statistical quality control*. Wiley, New York, NY [u.a.], 3. ed edition, 1997. URL: http://gso.gbv.de/DB=2.1/CMD?ACT=SRCHA&SRT=YOP&IKT=1016&TRM=ppn+192972065&sourceid=fbw_bibsonomy.

66 Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring endpoint congestion control. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 30–43, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3230543.3230553.

67 Kathleen Nichols and Van Jacobson. Controlling queue delay. *Commun. ACM*, 55(7):42–50, July 2012. doi:10.1145/2209249.2209264.

68 Yuqi Nie, Nam H. Nguyen, Phanwadee Sinthong, and Jayant Kalagnanam. A time series is worth 64 words: Long-term forecasting with transformers, 2023. arXiv:2211.14730.

69 Venkata N. Padmanabhan and Jeffrey C. Mogul. Using predictive prefetching to improve world wide web latency. *SIGCOMM Comput. Commun. Rev.*, 26(3):22–36, July 1996. doi:10.1145/235160.235164.

70 Craig Partridge, David Cousins, Alden W. Jackson, Rajesh Krishnan, Tushar Saxena, and W. Timothy Strayer. Using signal processing to analyze wireless data traffic. In *Proceedings of the 1st ACM Workshop on Wireless Security*, WiSE '02, pages 67–76, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/570681.570689.

71 Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks, 2013. arXiv:1211.5063.

72 Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019. arXiv:1912.01703.

73 Himabindu Pucha, Ying Zhang, Z. Morley Mao, and Y. Charlie Hu. Understanding network delay changes caused by routing events. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '07, pages 73–84, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1254882.1254891.

74 Benjamin J Radford, Leonardo M Apolonio, Antonio J Trias, and Jim A Simpson. Network traffic anomaly detection using recurrent neural networks. *arXiv preprint arXiv:1803.10769*, 2018. arXiv:1803.10769.

75 Sebastian Raschka. Using and finetuning pretrained transformers, 2024. URL: https://sebastianraschka.com/blog/2024/using-finetuning-transformers.html.

76 George F. Riley, Thomas R. Henderson, Klaus Wehrle, Mesut Güneş, and James Gross. *The ns-3 Network Simulator*, pages 15–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. doi:10.1007/978-3-642-12331-3_2.

77 Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. *SIGCOMM Comput. Commun. Rev.*, 45(4):123–137, August 2015. doi:10.1145/2829988.2787472.

78 Koen De Schepper, Olga Albisser, Olivier Tilmans, and Bob Briscoe. Dual queue coupled aqm: Deployable very low queuing delay for all, 2022. doi:10.48550/arXiv.2209.01078.

79 Koen De Schepper and Bob Briscoe. The Explicit Congestion Notification (ECN) Protocol for Low Latency, Low Loss, and Scalable Throughput (L4S). RFC 9331. doi:10.17487/RFC9331.

80 Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In Marilyn Walker, Heng Ji, and Amanda Stent, editors, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 464–468, New Orleans, Louisiana, 2018. Association for Computational Linguistics. doi:10.18653/v1/N18-2074.

81 Sima Siami-Namini, Neda Tavakoli, and Akbar Siami Namin. The performance of lstm and bilstm in forecasting time series. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 3285–3292, 2019. doi:10.1109/BigData47090.2019.9005997.

82 Nirhoshan Sivaroopan, Dumindu Bandara, Chamara Madarasingha, Guilluame Jourjon, Anura Jayasumana, and Kanchana Thilakarathna. Netdiffus: Network traffic generation by diffusion models through time-series imaging. *arXiv preprint arXiv:2310.04429*, 2023. doi:10.48550/arXiv.2310.04429.

83 Ralf C. Staudemeyer and Eric Rothstein Morris. Understanding lstm – a tutorial into long short-term memory recurrent neural networks, 2019. arXiv:1909.09586.

84 Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 513–527, Oakland, CA, May 2015. USENIX Association. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/suresh.

85 Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. Taurus: a data plane architecture for per-packet ml. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, pages 1099–1114, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3503222.3507726.

86 Ruchi Tripathi and Ketan Rajawat. Adaptive network latency prediction from noisy measurements. *IEEE Transactions on Network and Service Management*, 18(1):807–821, 2021. doi:10.1109/TNSM.2021.3051736.

87 Mohammad Valipour, Mohammad Ebrahim Banihabib, and Seyyed Mahmood Reza Behbahani. Parameters estimate of autoregressive moving average and autoregressive integrated moving average models and compare their ability for inflow forecasting. *Journal of Mathematics and Statistics*, 8:330–338, August 2012. doi:10.3844/jmssp.2012.330.338.

88 Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(86):2579–2605, 2008. URL: http://jmlr.org/papers/v9/vandermaaten08a.html.

89 Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

90 Kevin Vermeulen, Ege Gurmericliler, Italo Cunha, David Choffnes, and Ethan Katz-Bassett. Internet scale reverse traceroute. In *Proceedings of the 22nd ACM Internet Measurement Conference*, IMC '22, pages 694–715, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3517745.3561422.

91 Tyler Webb. Voip jitter and latency: Causes and how to troubleshoot. Technical report, GetVoIP, 2023.

92 Walter Weiss, Dr. Juha Heinanen, Fred Baker, and John T. Wroclawski. Assured Forwarding PHB Group. RFC 2597. doi:10.17487/RFC2597.

93 Duo Wu, Xianda Wang, Yaqi Qiao, Zhi Wang, Junchen Jiang, Shuguang Cui, and Fangxin Wang. Netllm: Adapting large language models for networking, 2024. arXiv:2402.02338.

94 Haixu Wu, Jiehui Xu, Jianmin Wang, and Mingsheng Long. Autoformer: Decomposition transformers with auto-correlation for long-term series forecasting, 2022. arXiv:2106.13008.

95 Yuhuai Wu, Markus N. Rabe, DeLesley Hutchins, and Christian Szegedy. Memorizing transformers, 2022. doi:10.48550/arXiv.2203.08913.

96 Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 495–511, Santa Clara, CA, February 2020. USENIX Association. URL: https://www.usenix.org/conference/nsdi20/presentation/yan.

97 Ailing Zeng, Muxi Chen, Lei Zhang, and Qiang Xu. Are transformers effective for time series forecasting? In *Proceedings of the AAAI conference on artificial intelligence*, volume 37, pages 11121–11128, 2023. doi:10.1609/AAAI.V37I9.26317.

98 Zhen Zeng, Rachneet Kaur, Suchetha Siddagangappa, Saba Rahimi, Tucker Balch, and Manuela Veloso. Financial time series forecasting using cnn and transformer, 2023. doi:10.48550/arXiv.2304.04912.

99 Junxue Zhang, Chaoliang Zeng, Hong Zhang, Shuihai Hu, and Kai Chen. Liteflow: towards high-performance adaptive neural networks for kernel datapath. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, pages 414–427, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3544216.3544229.

100 Siyao Zhang, Daocheng Fu, Zhao Zhang, Bin Yu, and Pinlong Cai. Trafficgpt: Viewing, processing and interacting with traffic foundation models, 2023. doi:10.48550/arXiv.2309.06719.

101 Xiao Zhang, Tanmoy Sen, Zheyuan Zhang, Tim April, Balakrishnan Chandrasekaran, David Choffnes, Bruce M. Maggs, Haiying Shen, Ramesh K. Sitaraman, and Xiaowei Yang. Anyopt: predicting and optimizing ip anycast performance. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, pages 447–462, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3452296.3472935.

102 Yinchao Zhang, Su Yao, Yong Feng, Kang Chen, Tong Li, Zhuotao Liu, Yi Zhao, Lexuan Zhang, Xiangyu Gao, Feng Xiong, Qi Li, and Ke Xu. Pegasus: A universal framework for scalable deep learning inference on the dataplane. In *Proceedings of the ACM SIGCOMM 2025 Conference*, SIGCOMM '25, pages 692–706, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3718958.3750529.

103 Kevin Zhao, Prateesh Goyal, Mohammad Alizadeh, and Thomas E. Anderson. Scalable tail latency estimation for data center networks. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 685–702, Boston, MA, April 2023. USENIX Association. URL: https://www.usenix.org/conference/nsdi23/presentation/zhao-kevin.

104 Haoyi Zhou, Shanghang Zhang, Jieqi Peng, Shuai Zhang, Jianxin Li, Hui Xiong, and Wancai Zhang. Informer: Beyond efficient transformer for long sequence time-series forecasting, 2021. arXiv:2012.07436.

105 Jiawei Zhou, Woojeong Kim, Zhiying Xu, Alexander M. Rush, and Minlan Yu. Netflowgen: Leveraging generative pre-training for network traffic dynamics, 2024. doi:10.48550/arXiv.2412.20635.

106 Tian Zhou, Ziqing Ma, Qingsong Wen, Xue Wang, Liang Sun, and Rong Jin. Fedformer: Frequency enhanced decomposed transformer for long-term series forecasting, 2022. arXiv:2201.12740.

107 Tian Zhou, Peisong Niu, xue wang, Liang Sun, and Rong Jin. One fits all: Power general time series analysis by pretrained lm. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 43322–43355. Curran Associates, Inc., 2023. URL: https://proceedings.neurips.cc/paper_files/paper/2023/file/86c17de05579cde52025f9984e6e2ebb-Paper-Conference.pdf.