Distilling Large Language Models for Network Active Queue Management

Shiva Raj Pokhrel, *Senior Member, IEEE*, Deol Satish, Jonathan Kua, *Member, IEEE*, and Anwar Walid, *Fellow, IEEE*

***Abstract*—We propose AQM-LLM, a framework that distills Large Language Models for Active Queue Management in modern networks. Unlike conventional learning-based AQMs that require extensive feature engineering and struggle with dynamic conditions, our approach leverages the contextual reasoning capability of LLMs to enhance the Low Latency, Low Loss, and Scalable Throughput (L4S) architecture with minimal manual intervention. The L4S-LLM design introduces three key components: 1) a state encoder that transforms heterogeneous network telemetry into token embeddings, 2) a specialized L4S-LLM head that produces congestion actions in a single inference step, and 3) a data-driven Low-Rank Adaptation scheme that drastically reduces trainable parameters while preserving accuracy. Our open-source FreeBSD-14 implementation demonstrates improved queue delay stability and higher bandwidth utilization for both DCTCP and UDP Prague traffic. We emphasize that this work demonstrates architectural feasibility through controlled experiments; deployment on router-class hardware will require additional model optimization (e.g., compression, pruning, or quantization) and is left for future work.**

***Index Terms*—AQM, large language model, congestion prevention, low latency, low loss, scalable throughput.**

I. Introduction

SINCE the 1990s, Active Queue Management (AQM) algorithms have employed mathematical models for optimal rule-based control systems. Early algorithms like Random Early Detection (RED) [1] utilized queue length as the primary buffer management metric. Modern AQMs including Controlled Delay (CoDel) [2], Proportional Integral Controller Enhanced (PIE) [3], and Common Applications Kept Enhanced (CAKE) have transitioned to queue delay-based metrics. This shift enables direct latency control and enhanced network performance. The **Low Latency, Low Loss, and Scalable Throughput** (L4S) framework [RFC 9330] [4] exemplifies these advances.[1] Recent Bell Labs findings validate these approaches [5].

Rule-based AQMs encounter critical limitations in contemporary heterogeneous networks. WiFi, 5G/6G, optical fiber, and satellite environments exhibit rapid link quality fluctuations. Traffic patterns and interference levels change faster than formulaic adaptation capabilities. Dynamic heterogeneous challenges necessitate complex, computationally intensive rules. Continuous parameter tuning and periodic re-engineering, perhaps by employing machine learning (ML), become mandatory[2] [6].

Deep learning enables learning-based AQMs that reduce reliance on predefined analytical models [7]. Supervised methods support tasks such as traffic classification and bandwidth estimation, while reinforcement learning has been applied to congestion control, adaptive streaming, and cloud scheduling [8]. Although learning-based approaches can adapt more effectively to dynamic network conditions than rule-based systems [5], their deployment remains challenging due to three factors: (1) *Design complexity*, as architecture design, hyperparameter tuning, and training require substantial effort and computational resources; (2) *Generalization limitations*, where models trained under static traffic conditions often perform poorly in real-world heterogeneous environments; and (3) *Retraining overhead*, since evolving network conditions demand continuous dataset collection and retraining, increasing operational cost.

To address these challenges, we adopt transformer-based policies that capture long-range temporal dependencies and cross-feature interactions in heterogeneous network telemetry. The controller is defined by adapting the self-attention mechanism which models global correlations across queue delay, burst allowance, drop probability, and packet statistics. Unlike recurrent architectures that rely on local receptive fields or sequential state compression, attention enables direct modeling of non-local dependencies in congestion dynamics. While simpler neural architectures [5] offer lower computations, our evaluation across multiple transformer foundations reveals a clear performance–complexity trade-off, with larger models achieving higher policy fidelity.

A. LLMs Over AQM: Challenges and Distillation

Leveraging Large Language Models (LLMs) like GPT, Opt, and Llama [9], [10], [11], [12] into AQM can address the high costs of developing task-specific DNNs for solving networking challenges [11], [13]. Pretrained on vast text datasets, LLMs excel in contextual understanding, reasoning,

Received 28 January 2025; revised 15 August 2025 and 9 March 2026; accepted 28 April 2026; approved by IEEE TRANSACTIONS ON NETWORKING Editor D. Pei. Date of publication 4 May 2026; date of current version 10 June 2026. This work was supported in part by APNIC Foundation Ltd. (via ISIF Asia Fund) 2023-2026 and in part by Australian Defence Science and Technology 2025. *(Corresponding author: Shiva Raj Pokhrel.)*

Shiva Raj Pokhrel, Deol Satish, and Jonathan Kua are with the School of IT, Deakin University, Geelong, VIC 3220, Australia (e-mail: shiva.pokhrel@deakin.edu.au).

Anwar Walid is with Defense Unicorns, Colorado Springs, CO, USA, and also with Columbia University, New York, NY 10027 USA.

This article has supplementary downloadable material available at https://doi.org/10.1109/TON.2026.3690076, provided by the authors.

Digital Object Identifier 10.1109/TON.2026.3690076

[1] https://www.rfc-editor.org/rfc/rfc9330.html

[2] https://www.bell-labs.com/white-papers/l4s/, accessed 15 Aug 2025.

and generalization [10] making them ideal for tasks like AQM. Their attention mechanism enables dynamic analysis of traffic parameters and dependencies, allowing proactive congestion management and adaptability to evolving network conditions. The distillation of LLM for AQM offers a universal framework to improve the Internet with minimal modifications, which can perhaps be achieved by adopting new RL ideas for the distillation of knowledge [10], [14]. However, key challenges include:

(1) *Diverse Inputs*: AQM data includes time series (e.g., queue delay) and scalar values (e.g., burst allowance), requiring specialized encoders.
(2) *Inference Latency*: Token-by-token output generation causes delays, disrupting AQM's rapid update cycles (45 ms). Hallucinated outputs further exacerbate the delays.
(3) *Distillation Costs*: Distilling LLMs for AQM requires time-consuming fine-tuning, especially in decision-making tasks using reinforcement learning, and demands significant computational resources for large models. While adopting LLMs can unify and enhance AQM, the above mentioned advances must be developed for effective AQM-LLM design and deployment.

B. Preliminaries and Design Details of L4S-LLM

For clarity, we first define the terminology and relationships between the proposed components. **AQM-LLM** denotes the general framework that adapts transformer-based large language models for Active Queue Management. The framework models congestion control as a sequential decision process in which heterogeneous network telemetry is encoded into token representations and processed by a transformer policy to generate congestion actions.

L4S-LLM is a specific instantiation of the AQM-LLM framework within the L4S architecture. In this design, the LLM operates as an external inference module that analyzes L4S telemetry and provides periodic policy guidance for congestion control decisions. In contrast, **L4S-AQM** refers to the underlying rule-based queue management mechanism defined by the L4S architecture (e.g., DualPI2), which executes packet-level operations such as enqueue, drop, and ECN marking directly within the router dataplane. The relationship between the components can therefore be expressed as

$$\text{L4S-LLM} \subset \text{AQM-LLM} \tag{1}$$

and L4S-AQM is the baseline dataplane controller. The overall system follows a hybrid control architecture in which the LLM does not directly execute packet-level operations. Instead, the native L4S-AQM dataplane remains responsible for real-time queue control, while the LLM periodically provides policy guidance based on observed network dynamics. This separation preserves compatibility with existing AQM mechanisms and guarantees that router forwarding performance is not degraded.

Building on this framework, we develop AQM-LLM and implement *L4S-LLM* as a proof-of-concept intelligent congestion prevention system within the L4S architecture. The design

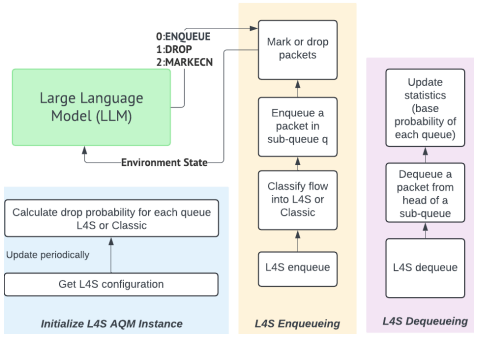


Fig. 1. L4S-LLM Coupling: Interaction between different components of L4S and LLM, illustrating how the model integrates with the existing network stack.

adopts a hybrid safety mechanism in which the LLM operates as an external guidance module while the baseline L4S AQM continues to perform packet-level decisions. This enables rule-based verification, confidence-based fallback to the baseline controller, and telemetry logging for debugging and validation.

L4S-LLM adapts a foundational LLM for queue management with minimal architectural modifications through three key components:

1) *State Encoder:* A multi-path encoder that transforms heterogeneous network telemetry (e.g., queue delay and packet drops) into token embeddings compatible with transformer processing.
2) *L4S-LLM Head:* A specialized action head that replaces the standard language modeling head and directly maps transformer outputs to congestion actions (enqueue, drop, mark) in a single inference step, eliminating token-by-token generation and reducing latency.
3) *Data-Driven Low-Rank L4S Adaptation (LoRA):* The model is fine-tuned through RL-based distillation over experience traces generated by existing AQM algorithms. Low-Rank Adaptation enables parameter-efficient training, reducing GPU memory usage by 64% and training time by 15.1%.

Our key contributions in this paper are as follows.

1) Identification of key challenges and advantages of using LLMs for network queue management.
2) Development of L4S-LLM with a multi-faceted encoder, specialized L4S-LLM head, and the proposed LoRA scheme for improved RL-based distillation and finetuning of the L4S-LLM.
3) Evaluation demonstrating improved queue management, congestion mitigation, and latency reduction across unseen data and network environments.

L4S-LLM Head Design & Development. Conventional LLM heads rely on iterative token-by-token generation, introducing substantial latency unsuitable for AQM. To address this, we design the L4S-LLM head, which predicts a probability distribution over congestion actions (enqueue, drop, mark) in a single inference step. Inspired by speculative decoding, this design eliminates sequential generation and enables parallel evaluation of candidate outcomes, significantly reducing inference latency while preserving accuracy.

Figure 1 illustrates the integration of the LLM module with the L4S network stack. The LLM analyzes real-time network telemetry and traffic patterns to provide adaptive guidance for key L4S parameters, including dual-queue management and ECN marking thresholds. Through this feedback loop, the system enables predictive congestion control while preserving the operational stability of the underlying L4S architecture, improving network efficiency and reducing latency.

Algorithm 1 L4S-LLM State Encoder

Input: State tensor `state` with shape `(batch_size, seq_len, 8, 1)`
Output: Encoded feature tensors for each state component
Initialize layers:
    fc_1,..., fc_nas Linear(1,feature_dim) for scalar data
    $conv_1$,..., $conv_n$as Conv1d(1, feature_dim, conv_size) for sequence data
    $embed\_feature$,..., $embed\_feature\_n$ as Linear(feature_dim, plm_embed_size) where plm_embed_size depends on LLM token size.
for each feature i from 1 to n **do**
    if feature i is scalar data **then**
        Extract feature_i from state[:, i-1:i]
        Apply fc_i(feature_i) to encode the feature
    else
        Extract feature_i from state[:, i-1:i]
        Apply conv_i(feature_i) to encode the feature

    end if
    Apply embed_i(feature_i) to encode the featureinto its embedded representation ▷Embedding Layer
    Reshape encoded feature to (batch_size, seq_len,-1)
end for
return Encoded features: features1, features2,..., features8

Temporal Modeling Framework. Our temporal modeling framework adopts a dual-stage architecture to capture queue dynamics across multiple timescales. First, 1D convolutional networks (Algorithm 1) with multi-scale kernels extract temporal features from sequential network metrics, providing efficient preprocessing before transformer reasoning. The transformer then applies self-attention over an empirically optimized context window that balances performance and computational efficiency; shorter windows degraded accuracy, while longer ones provided no additional benefit. To prevent temporal overfitting and improve generalization, we employ protocol-diverse training traces, stochastic temporal augmentation, and temporal dropout. Empirical results confirm that this design maintains strong accuracy on previously unseen network conditions while optimizing long-horizon congestion control objectives through discounted cumulative rewards.

It is worth noting that the LLM module operates as a loosely coupled external inference service that provides periodic policy guidance, while packet-level enqueue, drop, and ECN-marking decisions continue to be executed by the native L4S AQM dataplane, ensuring no additional runtime overhead or degradation of baseline router performance [15], [16]. By learning traffic patterns indirectly from network telemetry, L4S-LLM enables proactive congestion prevention and demonstrates strong generalization across previously unseen network scenarios. In the congestion control literature, we have examples [6], [17] where we setup an external controller (out of the critical run time path) used to affect the decision in behavior of the kernel.

Open Source FreeBSD Implementation. We design and implement L4S-LLM on FreeBSD-14, providing publicly accessible user and kernel modules to facilitate the testing and experimentation by the broader network and Internet research community.

C. Literature

Networks often suffer from bufferbloat [18], [19], where excessive packet buffering leads to high latency and jitter. To mitigate this, AQM schemes were developed, which detect congestion and either drop packets or use ECN to signal TCP congestion control algorithms to adjust the congestion window. Rule-based AQMs like CoDel [20], PIE [3], and CAKE depend on manually created rules, but these static approaches struggle to adapt to dynamic network traffic patterns.

Learning-based AQM algorithms address these limitations by leveraging deep learning techniques to automate decision-making. Early works, such as PFED [21], forecast traffic using MMSE prediction to regulate queue length and penalize misbehaving flows. Advanced approaches include ECN-based algorithms using LSTM architectures for traffic prediction [8], QP-AQM for adaptive queue management with Q-learning traffic predictors [22], and Deep Q-Network-based AQMs for intelligent packet dropping and differential QoS [23].

While these methods improve adaptability, they face challenges like high engineering effort, computational cost, and resource-intensive training. Their reliance on continuous learning limits generalization across diverse network environments, hindering practical deployment in real-world settings. LLMs [24] like ChatGPT, PaLM, Llama2, and OPT are advanced deep neural networks built on the Transformer architecture, with widespread applications across diverse domains. LLMs process input and output as token sequences, converting text into embeddings and predicting subsequent tokens using an auto-regressive mechanism. The self-attention mechanism allows LLMs to focus on relevant parts of the input, enabling them to handle complex language patterns and nuances effectively. Nevertheless, the LLM Driven networking has potential in several networking aspects including network function virtualization and large scale simulations [25], [26], [27].

There are some notable approaches for adapting LLMs to networking domains [28], [29]. Supervised Fine-Tuning (SFT) [30] enables efficient domain adaptation with minimal data while preserving instruction-following capabilities, but exhibits safety degradation that increases with dataset size and is particularly sensitive to structured formats like lists and tables [31]. Continual Pre-Training (CPT) [32] provides deeper domain knowledge integration and superior performance on complex tasks, but suffers from severe safety degradation when not followed by safety-focused instruction tuning [32], [33]. To address these safety concerns, several

TABLE I
ECN CODEPOINTS

Codepoint	Binary	Description
00	00 00	Non-ECT (Not ECN-Capable Transport)
01	00 01	ECT(0) (ECN-Capable Transport)
10	00 10	ECT(1) (ECN-Capable Transport)
11	00 11	CE (Congestion Experienced)

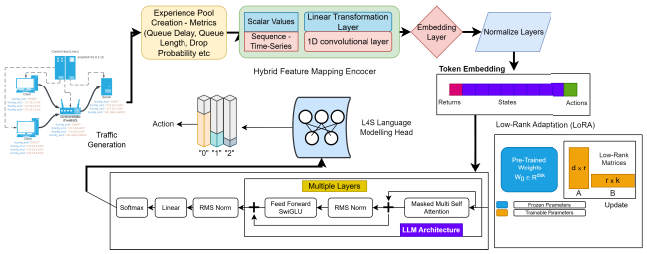


Fig. 2. L4S-LLM Architecture: from raw network metrics to AQM decisions, showing the data flow through the state encoder, transformer model, and L4S-LLM head.

methods have been proposed that maintain safety guardrails while preserving task performance. These include SafeInstruct [34], which interleaves safety-aligned QA pairs during fine-tuning; SafeLoRA [35], which selectively adapts only layers exhibiting harmful behavior; and SafeMERGE [36], which merges fine-tuned layers with those of known safe models.

II. RESEARCH DESIGN

The L4S architecture, defined in RFC 9330 [4], enhances QoE and QoS by integrating ECN with advanced TCP and AQM techniques. Central to this is the Dual-Queue Coupled AQM[3] [RFC 9332], which prioritizes latency-sensitive L4S flows over traditional Classic queues, reducing delay at the cost of minor performance trade-offs for Classic flows. Complementing this, the Prague TCP variant works synergistically with DualPI2 and ECN to minimize packet loss and latency while maintaining high throughput.

DualPI2 uses ECN (RFC 3168 [37]) to mark packets during congestion, resorting to packet drops only under severe congestion. ECN leverages two bits in the IP header's DiffServ field to signal congestion with four states. Table I shows the details. Non-ECT (00) indicates endpoints don't support ECN, so routers drop packets during congestion. ECT (1) (01) and ECT(0) (10) show endpoints are ECN-capable, allowing routers to mark packets instead of dropping them. CE (11) indicates the packet experienced congestion and is set by routers to notify the receiving endpoint.

Upon receiving CE-marked packets, the receiver informs the sender via an ECE flag, prompting the sender to adjust congestion control and respond with a CWR flag to acknowledge the notification. This mechanism mitigates delays, reduces packet loss, and minimizes retransmission, leading to lower latency, improved throughput, and decreased head-of-line blocking [38]. An abstract view of our L4S-LLM Architecture with complete pipeline from raw network metrics to AQM decisions, are shown in Figure 2 with the data flow through

[3]https://www.rfc-editor.org/rfc/rfc9332.html

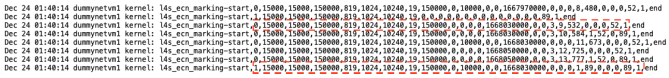


Fig. 3. Snapshot of raw kernel log.

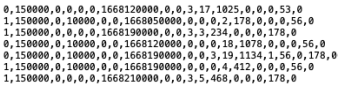


Fig. 4. Snapshot of our processed experience pool.

the state encoder, transformer model, and L4S-LLM head. We discuss the components of the L4S-LLM Architecture in details as follows.

A. Kernel Log Collection and Format

Our approach begins with collecting raw network metrics directly from the FreeBSD kernel's DualPI2 implementation. These logs capture the complete state of the AQM system at each decision point. Figure 3 shows an example of the raw kernel log in the format (observe the values in the highlighted dashed rectangles): *queue type, qdelay reference, tupdate, max burst, max ecn threshold, alpha coefficient, beta coefficient, flags, burst allowance, drop probability, current queue delay, previous queue delay, accumulated probability, measurement start time, average dequeue time, dequeue count, status flags, total packets, total bytes, queue length, length in bytes, total drops, packet length, dequeue action.*

Each log entry represents a single packet processing event with timestamps and key metrics. The final field in the dashed rectangle, `action`, indicates the decision made by the original DualPI2 algorithm: 0 for enqueue, 1 for drop, and 2 for ECN marking. See Table I for details. These logs are collected using a custom kernel module seamlessly that hooks into the DualPI2 decision path without affecting normal operation.

B. Experience Pool Generation

The raw kernel logs are processed into structured experience pools for training and evaluating our L4S-LLM model. Each entry contains a state vector of 8 network condition features, an action decision (0: enqueue, 1: drop, 2: ECN mark), a computed reward (balancing throughput and latency), and a Boolean flag refer to as *Done* indicating episode termination (0 except at transmission end). Figure 4 shows the snapshot of the processed experience pool. The experience pools are stored in a structured JSON format to facilitate efficient loading during model training.

We demonstrate the L4s-LLM pipeline overview from data collection to LLM head in Table II. Table III displays all metric data obtained from the kernel along with their descriptions. Custom logging functions were implemented within the kernel to accurately capture data for analysis, offline data cleaning, and preparation for generating the LLM experience pool.

C. Tokenization and Embedding Process

Unlike traditional NLP applications where text is tokenized into discrete units, our network metrics require specialized

TABLE II
PIPELINE STAGES AND OVERVIEW

Stage	Description	Example
Kernel Log Collection	Real-time AQM metrics collected from kernel-level code during packet processing	Queue Type = 1, Queue Delay = 50ms, Drop Prob = 0.02
Experience Pool Generation	Kernel logs transformed into training/evaluation pools with state, action, reward, and done flags	State: $[f_1, f_2, \ldots, f_n]$, Action: $\{0, 1, 2\}$, Reward: throughput, Done: $\{0, 1\}$
Tokenization	Network data converted into discrete tokens for LLM processing	Network state $\rightarrow$ [queue_delay, drop_prob, pkt_len, ...]
Embedding	Tokens mapped to high-dimensional vectors matching LLM input space	queue_delay $\rightarrow [0.2, 0.7, 0.1, \ldots]$, action $\rightarrow [0.3, 0.1, -0.1, \ldots]$
RL Adaptation	L4S-LLM fine-tuned using rewards to optimize AQM decisions	$s_t = (qdelay_t, pdrop_t, lenb_t)$, $R_t = \frac{pkt\_len_t}{qdelay_{t+1}}$
LoRA	Parameter-efficient training using low-rank matrices while freezing LLM weights	$\Delta W = A \cdot B$, where A, B are low-rank matrices
L4S Head	Maps LLM outputs to congestion actions (enqueue, drop, mark)	$Action = \arg\max(P(enqueue), P(drop), P(mark))$

TABLE III
COLLECTED KERNEL-LEVEL METRICS FROM FREEBSD DUALPI2 AQM

(a) Kernel logs utilized by L4S-LLM

Variable	Definition
queue_type	Queue type identifier (0: Classic, 1: L4S)
burst_allowance	Maximum allowed burst size before congestion control activation
drop_probability	Probability of dropping a packet during congestion
current_queue_delay	Current queue delay experienced by packet
accumulated_probability	Accumulated probability for drop decisions
length_in_bytes	Current queue length in bytes
packet_length	Length of current processed packet

(b) Non-utilized Kernel log parameters

Variable	Definition
qdelay_reference	Target queue delay reference for AQM algorithms
tupdate	Update interval for queue state variables
max_burst	Maximum allowable burst duration
max_ecn_threshold	Upper ECN marking threshold
alpha_coefficient	PI controller proportional gain
beta_coefficient	PI controller integral gain
flags	Internal AQM status flags
previous_queue_delay	Previous measurement cycle queue delay
measurement_start_time	Current measurement interval timestamp
average_dequeue_time	Mean packet dequeue time
dequeue_count	Packets dequeued in current interval
status_flags	Operational state indicators
total_packets	Total packets processed
total_bytes	Total bytes processed
queue_length	Current queue length in packets
total_drops	Total packet drops in interval
dequeue_action	Decision (0:enqueue, 1:drop, 2:ECN mark)

processing to be compatible with LLM architectures. We implement a custom tokenization process that converts numerical network metrics into a format suitable for transformer models.

We present a novel approach to network congestion control by adapting transformer-based language models for sequential decision-making in L4S environments. The architecture fundamentally reframes network state transitions as a sequence modeling problem, leveraging the autoregressive capabilities of pre-trained LLMs. Our tokenization process transforms heterogeneous network telemetry data into a unified sequential representation suitable for transformer processing. Rather than traditional linguistic tokens, the system constructs sequences from temporal network observations, where each timestep t generates a multi-modal token comprising: (1) return-to-go R_t representing cumulative future rewards, (2) decomposed state features $\mathbf{s}_t = [s_{t,1}, s_{t,2}, \ldots, s_{t,7}]$ capturing network conditions, and (3) action $a_t \in \{0, 1, 2\}$ representing packet handling decisions.

The sequence construction follows the following pattern:

$$\mathcal{S} = [R_t, s_{t,1}, s_{t,2}, \ldots, a_t, R_{t+1}, s_{t+1,1}, \ldots, a_{t+1}, \ldots] \quad (2)$$

which creates a causal dependency structure where states predict subsequent actions, aligning with the autoregressive nature of transformer architectures.

Furthermore, the proposed embedding mechanism employs separate linear projection layers for each modality to map heterogeneous input spaces into a unified d-dimensional embedding space matching the pre-trained LLM's hidden size. State features undergo domain-specific transformations through dedicated embedding matrices $\mathbf{W}_{\text{state},i} \in \mathbb{R}^{d \times d_{\text{state}}}$ for $i \in \{1, \ldots, 7\}$, while actions and returns are projected via $\mathbf{W}_{\text{action}} \in \mathbb{R}^{d \times 1}$ and $\mathbf{W}_{\text{return}} \in \mathbb{R}^{d \times 1}$ respectively.

Temporal information is incorporated through learned positional embeddings $\mathbf{W}_{\text{time}} \in \mathbb{R}^{(T+1) \times d}$, which are additively combined with all modality embeddings

$$\text{embed}(\mathbf{x}_t) = \mathbf{W}_{\text{modality}}(\mathbf{x}_t) + \mathbf{W}_{\text{time}}(t) \quad (3)$$

where $\mathbf{x}_t$ represents the input at timestep t.

The concatenated embedding sequence undergoes layer normalization before being fed into the transformer backbone, bypassing traditional tokenization by directly providing input embeddings rather than discrete tokens. This preserves the continuous nature of network telemetry while leveraging pre-trained attention patterns. The transformer processes sequences through multi-head self-attention mechanisms, capturing complex temporal dependencies and cross-modal interactions between network states, actions, and rewards. The attention mechanism computes weighted representations by calculating similarity scores between query, key, and value matrices, applying softmax normalization, and producing context-aware

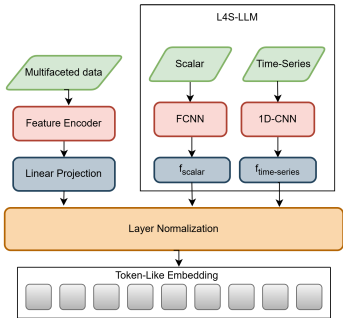


Fig. 5. L4S-LLM Encoder: Detailed architecture showing how different types of network metrics are processed through specialized encoding paths.

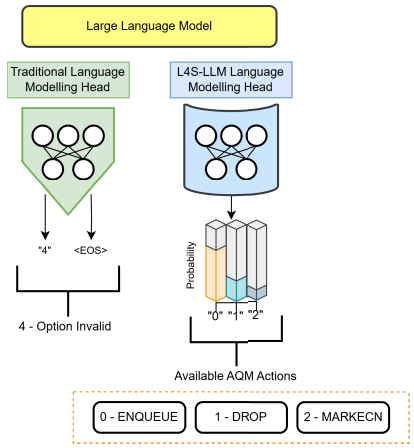


Fig. 6. L4S-LLM Head.

embeddings. The final hidden states from state embedding positions are extracted and passed through a linear classification head to generate action logits, treating congestion control as a sequence-to-sequence mapping where network history predicts optimal packet handling decisions.

D. Architecture of AQM-LLM

1) L4S-LLM State Encoder: The L4S-LLM State Encoder transforms raw network metrics into a format compatible with transformer models. As shown in Figure 5, it consists of two main components:

1) **Feature Encoder**: Processes different types of network metrics through specialized paths
2) **Linear Projector**: Maps extracted features to the token space required by the LLM

Algorithm 1 details the encoding process. For scalar metrics like queue type or drop probability, fully connected layers (line 4, Algorithm 1) extract meaningful representations. For time-series data like queue delay trends, 1D CNNs (line 5, Algorithm 1) capture temporal patterns with kernel sizes $k \in \{3, 5, 7\}$ to extract multi-scale features.

The extracted features are then projected into the LLM's token space using trainable embedding layers (line 16, Algorithm 1). For example, with Llama2, features are mapped to 4096-dimensional vectors compatible with the model's input requirements.

Algorithm 2 L4S Language Modelling Head

1: **Input:** $input\_embeddings, attention\_mask, action\_embed\_pos, residual\_flag$
2: **Output:** predicted_actions
3: **Initialize L4SLLM head:**action_head ← Linear($plm\_embed\_size, actionCount$)
4: action_head ← action_head.to(device) ▷Move to appropriate device
5: logits_used ← `Transformer`(input_embeddings, attention_mask)
6: **if** residual_flag **then**
7: logits_used ← logits_used + input_embeddings ▷Add residual connection
8: **end if**
9: selected_logits ← logits_used[:, action_embed_pos - 2] ▷Select relevant logits
10: predicted_actions ← action_head(selected_logits) ▷Predict actions using action head
11: **Return** predicted_actions ▷Probability Distribution of Candidate actions

2) L4S Language Modelling Head: Traditional language model heads generate responses token-by-token, requiring multiple inference rounds and introducing unacceptable latency for AQM applications. Our custom L4S-LLM head as shown in Figure 6 addresses this by generating complete actions in a single inference step. See details in Algorithm 2.

The L4S-LLM head (line 3, Algorithm 2) is a trainable linear layer that maps transformer outputs directly to a probability distribution over the three possible actions: enqueue (0), drop (1), or ECN mark (2).[4]

E. Data-Driven RL Pipeline

Recall that we employ an RL-based adaptation pipeline constructed from experience trajectories collected from the FreeBSD DualPI2 implementation. To guide learning, we define a reward function capturing the fundamental congestion-control objective of maximizing throughput while minimizing queueing delay:

$$r_t = \frac{\text{packet\_length}_t}{\text{qdelay}_t + 1}, \tag{4}$$

where packet_length_t denotes the transmitted packet size and $qdelay_t$ represents the measured queue delay. The denominator penalizes excessive queue buildup while maintaining numerical stability through a constant offset, encouraging high delivery rates with low latency consistent with the throughput–delay trade-off in modern AQM and L4S systems.

Each trajectory $\tau = \{s_t, a_t, r_t\}_{t=1}^{T}$ consists of states s_t represented by an 8-dimensional vector of network telemetry (queue type, burst allowance, drop probability, current queue delay, accumulated probability, queue length in bytes, packet drops, and packet length), actions $a_t \in \{0, 1, 2\}$ corresponding to

[4]This approach, inspired by speculative decoding, eliminates the need for token-by-token generation, reducing inference time from hundreds of milliseconds to under 40ms.

enqueue, drop, and ECN marking, and rewards r_t. The return-to-go is computed as

$$R_t = \sum_{i=t}^{T} \gamma^{i-t} r_i, \tag{5}$$

with discount factor $\gamma = 0.95$, producing training tuples $\{R_t, s_t^1, \ldots, s_t^8, a_t\}$.

During training, batches are sampled from the experience pool using context windows of length w, forming sequences

$$d = \{R_t, s_t^1, \ldots, s_t^8, a_t\}_{t=t-w+1}^{t}.$$

The transformer processes the encoded states to produce action predictions $\hat{a}_t$, and the policy is optimized using a categorical objective

$$L = \frac{1}{w} \sum_{t=1}^{w} F_{CE}(a_t, \hat{a}_t), \tag{6}$$

where F_{CE} denotes the cross-entropy loss. This formulation enables the transformer policy to learn congestion-control decisions from historical network trajectories while capturing temporal dependencies across the observation window.

Periodic LLM Policy Intervention The proposed architecture follows a hybrid control model where the native L4S-AQM (DualPI2) performs packet-level operations in the dataplane, while the L4S-LLM acts as an external policy advisor invoked periodically (e.g., every $K \in \{10, \ldots, 100\}$ packets). Let s_t denote the network telemetry state at decision interval t, composed of queue delay, burst allowance, drop probability, and packet statistics. At every K packets, the LLM performs a single inference step

$$a_t = \pi_\theta(s_t), \quad a_t \in \{\text{enqueue}, \text{drop}, \text{ECN mark}\}, \tag{7}$$

where π_θ denotes the transformer policy. The recommended action does not directly replace the dataplane decision; instead, it modulates the AQM control parameters (e.g., marking probability or threshold bias)

$$p_{t+1} = p_t + \Delta(a_t), \tag{8}$$

thereby influencing the marking or dropping probability applied to subsequent packets. Consequently, while the packet processed at the invocation point may experience an adjusted control probability, the primary effect propagates through the queue dynamics and congestion feedback loop, shaping the trajectory of the following packet sequence. This loose coupling ensures that real-time packet processing remains entirely within the kernel dataplane, while the LLM provides periodic policy corrections outside the critical runtime path, preserving compatibility and stability of the underlying L4S controller. In the congestion control literature, we have examples [6] where such setup where an external controller (out of the critical run time path) used to affect the behavior of the kernel.

F. Low-Rank Adaptation (LoRA)

Fine-tuning large language models with billions of parameters is computationally prohibitive for networking applications. We address this using Low-Rank Adaptation (LoRA), which dramatically reduces the number of trainable parameters while preserving model quality. Figure 7 shows the LoRA fine-tuning architecture where low-rank matrices (A and B) integrate with frozen pre-trained weights, enabling parameter-efficient L4S-LLM adaptation to network-specific patterns without full model retraining. Algorithm 3 details the LoRA training process. For each pre-trained weight matrix $W_0 \in \mathbb{R}^{d\times k}$ in the LLM, we introduce low-rank matrices $A \in \mathbb{R}^{d\times r}$ and $B \in \mathbb{R}^{r\times k}$ with rank $r \ll \min(d, k)$. The effective weight matrix

$$W = W_0 + AB \tag{9}$$

By freezing W_0 and only training A and B, we reduce the number of trainable parameters from billions to millions—a 99% reduction for Llama2-7B.

For Llama2-7B, we use rank $r = 128$, reducing trainable parameters from 7 billion to approximately 70 million while maintaining 97.56% accuracy. This approach preserves the pre-trained knowledge in the LLM while enabling efficient adaptation to the AQM domain.

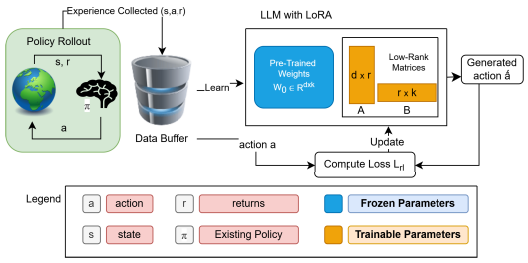


Fig. 7. Low-Rank L4S adaptation (LoRA) Design: Visualization of how LoRA matrices are integrated with frozen pre-trained weights to enable efficient fine-tuning.

Algorithm 3 Low-Rank Adaptation (LoRA) Algorithm

Input: Pretrained model W_θ, adaptation rank r, training dataset D, number of epochs E
Output: Fine-Tuned model parameters W_{fine_tuned}
Initialize low-rank matrices $A \in \mathbb{R}^{d\times r}$, $B \in \mathbb{R}^{r\times d}$ randomly
Set the learning rate η
for each epoch $e = 1$ to E **do**
  for each batch b in the dataset D **do**
    Forward pass: compute model output with current θ and fine-tuned weights (A, B)
    Compute the loss $\mathcal{L}$ based on the model output and ground truth
    Compute the gradient of the loss with respect to θ and (A, B)
    Update the low-rank matrices using gradient descent:

$$A \leftarrow A - \eta \frac{\partial \mathcal{L}}{\partial A}, \quad B \leftarrow B - \eta \frac{\partial \mathcal{L}}{\partial B}$$

  end for
end for
Set the final Fine-Tuned model parameters:

$$W_{fine\_tuned} \leftarrow W_\theta + AB$$

Return: θ_{adapt}

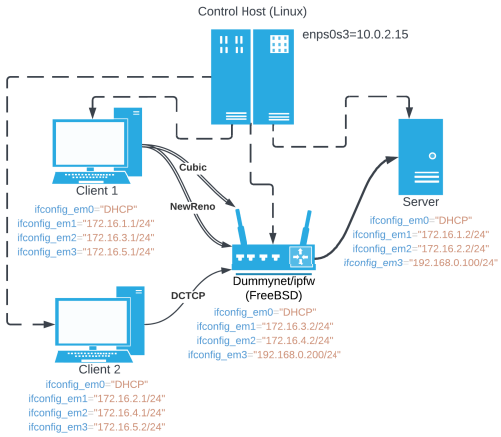


Fig. 8. Topology of the developed experimental testbed for collecting datasets and evaluating L4S-LLM framework.

III. Setup and Evaluation

A. Experimental Testbed Setup

We collect the experiences required for training our L4S-LLM by developing a testbed as shown in Figure 8 consisting of FreeBSD-based virtual machines, built and configured using VirtualBox and VMware virtualization platforms. The virtual machines run FreeBSD version 14.1 as their operating system. To create a network setup in VirtualBox, we clone four virtual machines: two clients (Client 1 and Client 2), one router, and one server. Client 1 utilizes two different congestion control algorithms - Cubic and NewReno simultaneously, while Client 2 uses ECT (1)-enabled DCTCP congestion control algorithm. We replace DCTCP with UDP Prague and repeat all the experiments in VMware.[5] The router is configured with a custom-built DualPI2 algorithm that supports the L4S architecture, as described in [5]. All clients send data to the server through the router as illustrated in Figure 8. All scripts and additional information required to set up the environment are made public in the L4S-LLM repository.

In our experiment to collect data, we configure the router with a bandwidth of 8 Mbps and a delay of 10 ms. We initiate two `iperf3` connections for each congestion control algorithm used in Client 1, with each connection utilizing two parallel flows via the `-P 2` option in `iperf3`. Client 2 only utilizes a single flow for its `iperf3` connection. Each `iperf3` test runs for one minute. We repeat the experiment ten times and collect all the data. We have enabled ECN on all devices. Table IV summarizes the baseline AQM configuration and adaptation methodology.

[5]VirtualBox provides open-source accessibility, simpler network configuration options, better FreeBSD guest support through built-in additions, and lower resource requirements suitable for basic protocol testing. In contrast, VMware delivers superior network performance with enhanced throughput and lower latency, advanced virtual networking capabilities (distributed switches, VLAN trunking, traffic shaping), better hardware virtualization, and more reliable snapshot functionality for experimental state preservation. For performance-sensitive experiments like congestion control or AQM evaluation, VMware's network stack more accurately emulates physical hardware behavior, while VirtualBox remains adequate for functional protocol testing with reduced resource overhead.

To gather the data, we use a debugging tool that is directly embedded in the DualPI2 kernel code to store data from the kernel into the debug kernel logs. The relevant data is then extracted from these logs. If the size of the collected data exceeds 1000 KB, the log files will be truncated and rotated to prevent excessive growth. To increase the file size limits, we modify the configuration files for the `newsyslog` service, found in */etc/newsyslog.conf*.

Once the data collection process is complete, we process the collected data into an experience pool to be fed into the LLM; see L4S-LLM experience pool[6] implementation for details. We experiment with multiple foundation LLM models, including Llama2[7] [9] (7b), OPT (1.3b), GPT2[8] [39] (125m) and T5-LLM[9] [40], which are fine-tuned for our L4S-LLM task. Among these, Llama2 demonstrated the best performance (see Section III-B) and was selected as the foundational LLM for extensive evaluation in simulations, where L4S-LLM is compared against the legacy AQM algorithms. To execute L4S-LLM, we utilize `Runpod`, an online cloud computing platform that provides access to powerful GPUs and scalable storage, ensuring efficient training and testing of our framework. We used two NVIDIA A40 GPUs, each equipped with 48 GB of VRAM, carried out on an Ubuntu 22.04 instance with CUDA 11.8 version.

We log key metrics during L4S-LLM training and testing, including loss, accuracy, reward, GPU/CPU utilization, computation time, RAM usage, and total training/testing time. To evaluate the model's efficiency, we analyze loss and accuracy, which are calculated using cross-entropy loss for this multi-class classification problem. Cross-entropy loss measures the discrepancy between predicted probabilities and true labels, outputs a scalar value where lower values indicate better performance. The training process employs mini-batches with gradient accumulation to enhance efficiency and stability. Gradient clipping is utilized to prevent gradient explosion by capping gradients to a predefined threshold during backpropagation. The model predicts a probability distribution over actions, represented by `action.preds`, and the predicted action is obtained using the `argmax()` function. True actions, extracted and normalized from the experience pool, are aligned with the predicted actions using a custom vectorization and conversion function to ensure compatibility for computing accuracy

$$\text{Accuracy} = \frac{1}{N}\sum_{i=1}^{N}\mathbb{I}(\text{preds\_actions}_i = \text{true\_classes}_i) \quad (10)$$

over all N samples or steps in the epoch where $\mathbb{I}$ is the indicator function returns 1 when the predicted action (preds_actions) matches the true action class (true_classes), and 0 otherwise.

B. L4S-LLM Training and Validation

1) Evaluation of Loss Dynamics in L4S-LLM: Figure 9 demonstrates the training loss evolution across four LLM

[6]https://github.com/MPTCP-FreeBSD/LLM_Gen_Exp_Pool
[7]https://www.llama.com/llama2
[8]https://openai.com/index/better-language-models/
[9]https://github.com/google-research/text-to-text-transfer-transformer

TABLE IV
BASELINE AQM CONFIGURATION AND ADAPTATION

Algorithm	Configuration Used	Guideline Source	Tuning Method	Validation Procedure
RED	$min_{th} = 0.2B$ $max_{th} = 0.6B$ $p_{max} = 0.1$ $w_q = 0.002$ (where B is queue buffer size)	RFC 2309 and RED deployment recommendations	Initial parameters derived from guideline ratios relative to buffer size and link capacity; lightly tuned to ensure stable queue occupancy under test traffic.	Multiple preliminary runs performed under Cubic and DCTCP traffic to verify stable queue behavior and consistent delay distribution.
CoDel	Target = 5 ms Interval = 100 ms	Original CoDel design specification and standard FreeBSD implementation defaults	Default parameters retained as recommended for general deployment; validated across experimental workloads.	Preliminary sensitivity checks confirmed that small parameter variations produced consistent queue delay trends.
DualPI2	FreeBSD L4S RFC 9330/9332	RFC 9330/9332 and FreeBSD DualPI2 reference implementation; **(L4S Baseline)**	Parameters inherited from the reference L4S implementation used to generate the experience traces.	Validated under Cubic, NewReno, DCTCP, and UDP Prague traffic patterns to ensure stable L4S operation.
L4S-LLM	Context window $w = 64$ LoRA rank $r = 128$ Inference interval = 10–100 packets	Empirical optimization based on training convergence and ablation experiments	Parameters selected to balance inference cost and queue control performance.	Validated via repeated experiments with Cubic, DCTCP, and UDP Prague traffic.

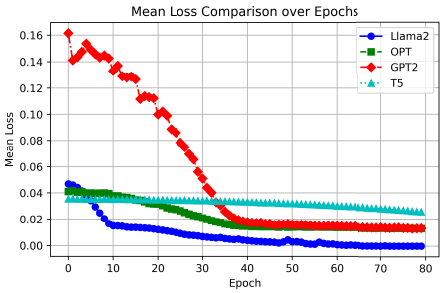


Fig. 9. L4S-LLM Training Loss Trends: Evolution of Mean Loss over 80 Epochs.

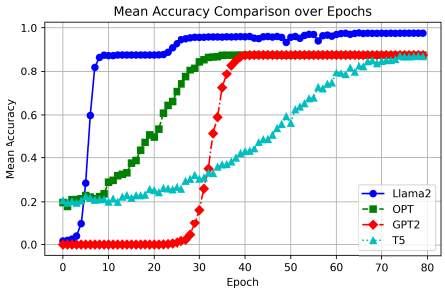


Fig. 10. L4S-LLM training accuracy trends: Mean Accuracy over 80 Epochs.

architectures over 80 epochs. Llama2 exhibits superior convergence characteristics, achieving the lowest mean loss by epoch 5 and maintaining consistent improvement throughout training. OPT displays gradual loss reduction with slower optimization dynamics, while T5-LLM follows similar trends but achieves marginally better final loss than OPT. GPT2 initially exhibits the highest loss but rapidly converges from epoch 40, ultimately matching OPT's performance level.

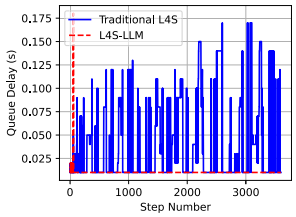
The loss trajectories validate L4S-LLM's robust adaptability to complex network queue management across diverse architectural foundations. By leveraging real-time kernel-level metrics, the framework demonstrates effective optimization and generalization under dynamic network conditions. Llama2's 7B parameter architecture achieves optimal loss minimization, followed by OPT and GPT2 with comparable performance, while T5-LLM exhibits slightly elevated loss values. These results reinforce Llama2's selection as the primary foundation model, combining superior convergence properties with effective parameter reduction through LoRA for practical deployment in network congestion control scenarios.

2) Evaluation of Accuracy Dynamics in L4S-LLM: Figure 10 illustrates the training accuracy evolution across four LLM architectures during fine-tuning. Llama2 exhibits initial near-zero accuracy with rapid improvement between epochs 5-9, followed by plateau and gradual ascent from epoch 30, ultimately achieving superior performance at 97.56%. OPT demonstrates higher initial accuracy than competitors, steadily increasing until epoch 40 where it stabilizes at 83%. GPT2 remains stagnant near zero until epoch 29, then exhibits sharp improvement through epoch 40 before stabilization. T5-LLM begins at 20% accuracy with gradual progression, plateauing around 83% after epoch 75.

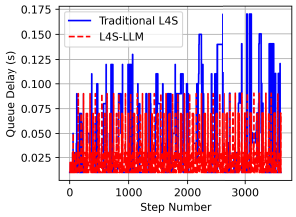
Llama2 significantly outperforms all alternatives, achieving 97.56% accuracy compared to the 82-83% plateau reached by OPT, GPT2, and T5-LLM. Despite requiring over 20 hours training time versus competitors' sub-hour requirements, Llama2's superior accuracy justifies the computational investment. Critically, LoRA reduces Llama2's 7B parameters to 70M trainable parameters while maintaining performance, and inference times remain comparable to smaller models during deployment.

C. Evaluation of L4S-LLM Over AQM

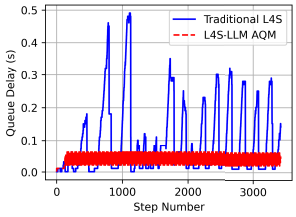
We evaluate the performance of our L4S-LLM framework, with Llama2 -7B as our base, fine-tuned using LoRA, with its performance benchmarked against real-world L4S AQM data. The benchmarking focuses primarily on optimizing queue delay and utilized bandwidth. The network topology depicted in Figure 8 involves Client 1 using Cubic and New-Reno concurrently, while Client 2 runs DCTCP, and the router is configured to implement L4S. Using real-world data LLM predicts the AQM action - Enqueue, Dequeue, or ECN marking. The LLM is used at specific intervals as it is



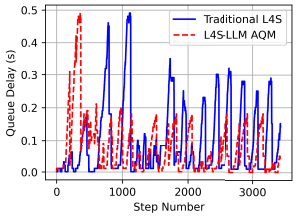
(a) Aggregate Delay with 10-Interval LLM Inputs with DCTCP



(b) Aggregate Delay with 100-Interval LLM Inputs with DCTCP

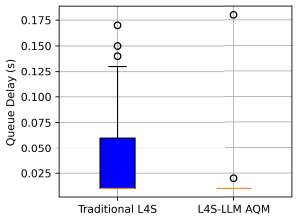


(c) Classic Queue with 10-Interval LLM Inputs with UDP Prague

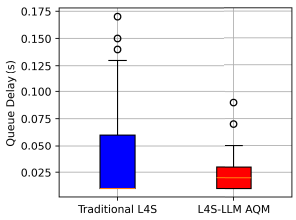


(d) Classic Queue with 100-Interval LLM Inputs with UDP Prague

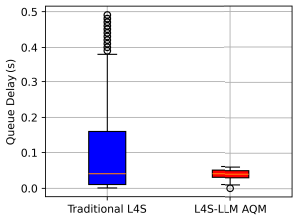
Fig. 11. Queue delay analysis with periodic LLM Inputs.



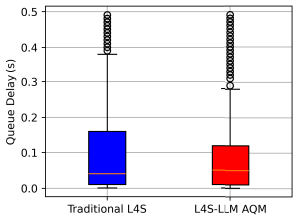
(a) DCTCP: 10-Interval Inputs



(b) DCTCP: 100-Interval Inputs



(c) UDP Prague: 10-Interval Inputs

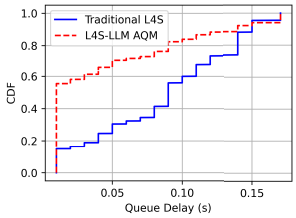


(d) UDP Prague: 100-Interval Inputs

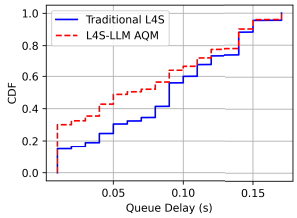
Fig. 12. Box Plot of DCTCP and UDP Prague Queue Delays of Fig 11.

resource-intensive. Our results, Figs. 11 and 12, demonstrate the LLM's ability to optimize the queue delay and bandwidth of both DCTCP and UDP Prague flows.

1) Analysis of Queue Delay: Figure 11 presents a comparison of queue delays between the original L4S AQM implementation and the enhanced L4S-LLM framework leveraging the fine-tuned LLM model to optimize queue delay. The LLM model determines the most optimal action–Enqueue, Drop and ECN marking–based on the system's state. As shown in Figure 12, our reward function guides the LLM in finding the most optimal action which keeps improving the reward. The results show a significant reduction in queue delay for the LLM-based system compared to the traditional algorithm. The traditional queue delay has a higher median queue delay with high variability. Conversely, the L4S-LLM achieves a remarkably low median queue delay value, with low variability and fewer outliers, demonstrating the LLM's

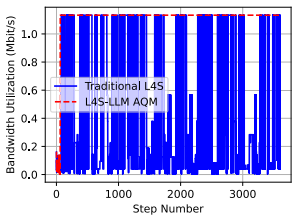


(a) CDF of Queue Delay with 10-Interval LLM Inputs

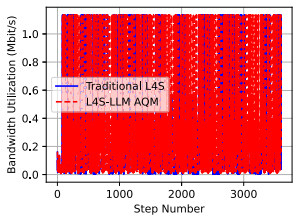


(b) CDF of Queue Delay with 100-Interval LLM Inputs

Fig. 13. CDF of DCTCP Queue Delay with Periodic LLM Inputs.

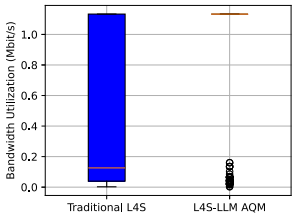


(a) Bandwidth with 10-Interval LLM Inputs

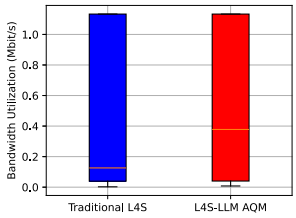


(b) Bandwidth with 100-Interval LLM Inputs

Fig. 14. Absolute bandwidths when L4S-LLM is enabled at 10 and 100-Packet Intervals.

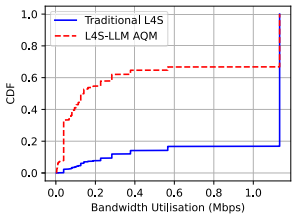


(a) Box Plot of Bandwidth with 10-Interval LLM Inputs

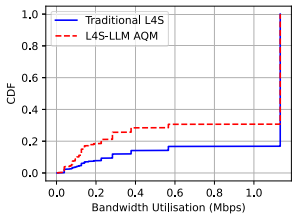


(b) Box Plot of Bandwidth with 100-Interval LLM Inputs

Fig. 15. Box Plot of used bandwidth with LLM Enabled at 10 and 100-Packet Intervals.



(a) CDF of Bandwidth with 10-Interval LLM Inputs



(b) CDF of Bandwidth with 100-Interval LLM Inputs

Fig. 16. CDF of Bandwidths with LLM Enabled at 10 and 100-Packet Intervals.

capacity to maintain low-latency behaviour. Our extensive analysis hereafter primarily focuses on DCTCP/VirtualBox, though results for UDP Prague/VMware have been produced using the code available in our GitHub repository.

Figure 13 illustrates the cumulative distribution functions (CDFs) of queue delay for traditional L4S and L4S-LLM AQM. These results provide an alternative representation of queue delay performance dynamics capturing LLM's effectiveness in optimizing queue management strategies.

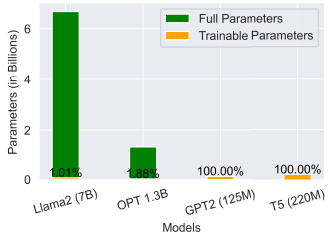


Fig. 17. Distribution of trainable parameters across four distinct LLMs.

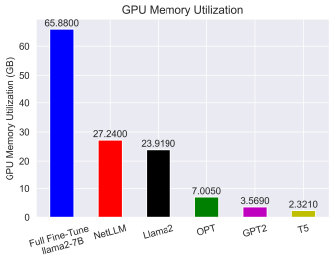


Fig. 18. Comparative Analysis of GPU VRAM Usage Across Different LLMs.

2) Analysis of Bandwidth: Figure 14 and Figure 15 showcase the comparative analysis of bandwidth perceived by the traditional L4S and L4S-LLM framework. The results show a significant increase in median obtained bandwidth, with significantly low variability, although along with a few large outliers. We found that these outliers mainly appear at the start of the data transfer and are not present after a certain duration. In contrast, the traditional L4S algorithm in Figure 16 shows a low median bandwidth value with high variability. Similarly, L4S-LLM shows its effectiveness in optimizing bandwidth harnessing while maintaining low queue delay.

The CDFs of used bandwidths for both traditional L4S and L4S-LLM AQM, as depicted in Figure 16, offer an alternative visualisation of router performance. These results underscore the LLM's capability to enhance queue management strategies effectively.

D. Computation Overhead(s)

Figure 17 demonstrates LoRA's parameter reduction efficiency: Llama2's 7B parameters reduce to 70M (1%), while OPT's 1.3B decrease to 25M (1.9%) using rank-128 LoRA. GPT-2 (125M) and T5-LLM (220M) need no reduction due to their manageable size. This rank-128 configuration optimally balances efficiency with performance, maintaining full fine-tuning accuracy while reducing computational demands. Figure 18 highlights memory constraints critical for deployment: Llama2 requires 23.91GB VRAM, OPT 7GB, GPT2 3.6GB, and T5-LLM just 2.32GB. These requirements emphasize the need for efficient architectures in resource-constrained networking environments where routers must operate within strict hardware limitations while maintaining real-time performance.

Figure 19 demonstrates L4S-LLM's significant impact on inference latency. The unadapted Llama2-7B exhibits prohibitive answer generation time of 0.200 seconds, while LoRA-adapted models achieve substantial reductions: Llama2 at 0.0391s, OPT at 0.0395s, GPT2 at 0.0414s, and T5-LLM at

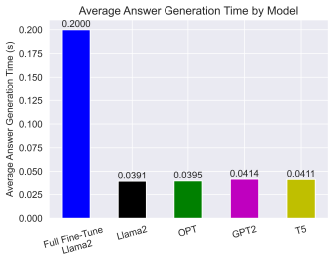


Fig. 19. Comparison of Decision Time Across Different LLMs (s).

Fig. 20. Comparison of Training Times Across Different LLMs (s).

0.0411s. Despite Llama2-7B and OPT-1.3B containing billions of parameters, LoRA reduces inference time by approximately 80%, achieving 0.039s response times crucial for real-time congestion control. This 5x speedup validates L4S-LLM's efficiency for latency-critical network applications. However, further optimization opportunities exist through small language models (SLMs), model pruning, partitioning strategies, and specialized hardware acceleration to achieve sub-millisecond inference suitable for high-frequency packet processing.

In Figure 20, we observe the total training time required to run 80 epochs for four different LLMs. Even with the application of LoRA, Llama2-7B exhibits the highest training time by a significant margin at approximately 22.3 hours. In contrast, the training time for OPT with LoRA is around 29.9 minutes, while GPT2 requires approximately 17.96 minutes, and T5-LLM requires about 32.26 minutes. With the exception of Llama2, the training times for other LLM models are notably faster. However, during evaluation or actual deployment, the inference time is of utmost importance. Despite Llama2's higher training time, its inference performance is comparable to other models, making it suitable for real-time congestion control applications.

Limitations. While our evaluation demonstrates the feasibility of LLM-assisted AQM under protocol-diverse traffic scenarios (e.g., Cubic, NewReno, DCTCP, and UDP Prague), we acknowledge that a comprehensive robustness assessment under adverse and non-stationary conditions remains an important direction for future work. In particular, stress scenarios such as flow bursts, RTT variability, loss bursts, satellite-like delays, link flaps, and potential congestion misclassification could further test the resilience of the proposed framework. Conducting such sensitivity and stress testing requires a larger controlled network experimentation environment and extended infrastructure. This work focuses primarily on establishing the architectural design and algorithmic feasibility of a loosely coupled LLM-assisted congestion control framework. Future

work, including ongoing collaboration with industry partners, will investigate deployment-scale evaluations under realistic network dynamics and perform sensitivity and ablation studies to better understand the impact of telemetry features and model components on congestion decision performance.

IV. Conclusion

This work presents AQM-LLM, a framework that adapts distilled large language models for Active Queue Management within the L4S architecture. By combining a telemetry-aware state encoder, a single-inference L4S-LLM decision head, and parameter-efficient Low-Rank Adaptation, the proposed system enables LLMs to act as adaptive policy modules for congestion control. Our open-source FreeBSD-14 implementation demonstrates improved queue delay stability and bandwidth utilization across both DCTCP and UDP Prague traffic scenarios. These results highlight the potential of transformer-based models to enhance data-driven queue management while remaining compatible with existing AQM mechanisms. While this study focuses on demonstrating architectural feasibility through controlled experiments, future work will explore model compression, pruning, quantization, and distillation toward small-language-model-based AQM to enable deployment on resource-constrained router platforms.

References

[1] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Trans. Netw.*, vol. 1, no. 4, pp. 397–413, Apr. 1993.

[2] K. Nichols, V. Jacobson, A. McGregor, and J. Iyengar, *Controlled Delay Active Queue Management*, document RFC 828, Jan. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8289

[3] R. Pan, P. Natarajan, F. Baker, and G. White, *Proportional Integral Controller Enhanced (PIE): A Lightweight Control Scheme To Address the Bufferbloat Problem*, document RFC 8033, Feb. 2017. [Online]. Available: https://www.rfc-editor.org/info/rfc8033

[4] B. Briscoe, K. D. Schepper, M. Bagnulo, and G. White, *Low Latency, Low Loss, and Scalable Throughput (L4S) Internet Service: Architecture*, document RFC 9330, Jan. 2023. [Online]. Available: https://www.rfc-editor.org/info/rfc9330

[5] D. Satish, J. Kua, and S. R. Pokhrel, "Active queue management in L4S with asynchronous advantage actor-critic: A FreeBSD networking stack perspective," *Future Internet*, vol. 16, no. 8, p. 265, Jul. 2024. [Online]. Available: https://www.mdpi.com/1999-5903/16/8/265

[6] S. R. Pokhrel and A. Walid, "Learning to harness bandwidth with multipath congestion control and scheduling," *IEEE Trans. Mobile Comput.*, vol. 22, no. 2, pp. 996–1009, Feb. 2023.

[7] S. R. Pokhrel, J. Choi, and A. Walid, "Fair and efficient distributed edge learning with hybrid multipath TCP," *IEEE/ACM Trans. Netw.*, vol. 31, no. 4, pp. 1582–1594, Aug. 2023.

[8] C. A. Gomez, X. Wang, and A. Shami, "Intelligent active queue management using explicit congestion notification," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2019, pp. 1–6.

[9] H. Touvron et al., "Llama 2: Open foundation and fine-tuned chat models," 2023, *arXiv:2307.09288*.

[10] D. Guo et al., "DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning," 2025, *arXiv:2501.12948*.

[11] D. Wu et al., "NetLLM: Adapting large language models for networking," in *Proc. ACM SIGCOMM Conf.*, Aug. 2024, pp. 661–678.

[12] G. O. Boateng et al., "A survey on large language models for communication, network, and service management: Application insights, challenges, and future directions," *IEEE Commun. Surveys Tuts.*, vol. 28, pp. 527–566, 2026.

[13] S. R. Pokhrel and A. Walid, "On large language model based joint source channel coding for semantic communication," in *Proc. 2nd Int. Conf. Found. Large Lang. Models (FLLM)*, Nov. 2024, pp. 322–329.

[14] S. R. Pokhrel et al., "Deakin RF-sensing: Experiments on correlated knowledge distillation for monitoring human postures with radios," *IEEE Sensors J.*, vol. 23, no. 22, pp. 28399–28410, Nov. 2023.

[15] X. Gao et al., "Nous: Drop-freeness and duplicate-freeness for consistent updating in SDN multicast routing," *IEEE/ACM Trans. Netw.*, vol. 32, no. 5, pp. 3685–3698, Oct. 2024, doi: 10.1109/TNET.2024.3404967.

[16] A. Majidi, X. Gao, S. Zhu, N. Jahanbakhsh, J. Zheng, and G. Chen, "MiFi: Bounded update to optimize network performance in software-defined data centers," *IEEE/ACM Trans. Netw.*, vol. 31, no. 1, pp. 322–335, Feb. 2023.

[17] S. R. Pokhrel, J. Kua, B. Fleming, S. Ozer, J. Howe, and A. Walid, "Multipath TCP implementation under FreeBSD-13 for pluggable machine learning models," *Comput. Netw.*, vol. 252, Oct. 2024, Art. no. 110671.

[18] J. Ye, K.-C. Leung, and S. H. Low, "Combating bufferbloat in multi-bottleneck networks: Theory and algorithms," *IEEE/ACM Trans. Netw.*, vol. 29, no. 4, pp. 1477–1493, Aug. 2021.

[19] J. Gettys and K. Nichols, "Bufferbloat: Dark buffers in the internet: Networks without effective AQM may again be vulnerable to congestion collapse," *Queue*, vol. 9, no. 11, pp. 40–54, Nov. 2011.

[20] K. Nichols and V. Jacobson, "Controlling queue delay," *Commun. ACM*, vol. 55, no. 7, pp. 42–50, 2012.

[21] W. Gao, J. Wang, J. Chen, and C. Song-qiao, "PFED: A prediction-based fair active queue management algorithm," in *Proc. Int. Conf. Parallel Process. (ICPP)*, vol. 24, 2005, pp. 485–491.

[22] J. Liu and D. Wei, "Active queue management based on Q-learning traffic predictor," in *Proc. Int. Conf. Cyber-Physical Social Intell. (ICCSI)*, 2022, pp. 399–404.

[23] M. Kim, M. Jaseemuddin, and A. Anpalagan, "Deep reinforcement learning based active queue management for IoT networks," *J. Netw. Syst. Manage.*, vol. 29, no. 3, p. 34, Jul. 2021.

[24] A. Vaswani et al., "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 6000–6010. [Online]. Available: https://dl.acm.org/doi/10.5555/3295222.329534

[25] G. Sun, Z. Xu, H. Yu, X. Chen, V. Chang, and A. V. Vasilakos, "Low-latency and resource-efficient service function chaining orchestration in network function virtualization," *IEEE Internet Things J.*, vol. 7, no. 7, pp. 5760–5772, Jul. 2020.

[26] G. Sun, G. Zhu, D. Liao, H. Yu, X. Du, and M. Guizani, "Cost-efficient service function chain orchestration for low-latency applications in NFV networks," *IEEE Syst. J.*, vol. 13, no. 4, pp. 3877–3888, Dec. 2019.

[27] F. Huang et al., "Large language model simulator for cold-start recommendation," in *Proc. 18th ACM Int. Conf. Web Search Data Mining*, 2025, pp. 261–270.

[28] W. Saad et al., "Artificial general intelligence (AGI)-native wireless systems: A journey beyond 6G," *Proc. IEEE*, vol. 113, no. 9, pp. 849–887, Sep. 2025.

[29] H. Zou et al., "Genainet: Enabling wireless collective intelligence via knowledge transfer and reasoning," 2024, *arXiv:2402.16631*.

[30] W.-L. Chiang et al., "Vicuna: An open-source chatbot impressing GPT-4 with 90% ChatGPT quality," Tech. Rep., 2023. [Online]. Available: https://www.lmsys.org/blog/2023-03-30-vicuna/

[31] A. Djuhera, S. R. Kadhe, F. Ahmed, S. Zawad, H. Boche, and W. Saad, "Safecomm: What about safety alignment in fine-tuned telecom large language models?," 2025, *arXiv:2506.00062*.

[32] A. Maatouk, K. C. Ampudia, R. Ying, and L. Tassiulas, "Tele-LLMs: A series of specialized large language models for telecommunications," 2024, *arXiv:2409.05314*.

[33] R. Nikbakht, M. Benzaghta, and G. Geraci, "TSpec-LLM: An open-source dataset for llm understanding of 3GPP specifications," 2024, *arXiv:2406.01768*.

[34] F. Bianchi et al., "Safety-tuned LLaMAs: Lessons from improving the safety of large language models that follow instructions," in *Proc. 12th Int. Conf. Learn. Represent.*, 2023, pp. 1–18.

[35] P.-Y. Chen, C.-Y. Hsu, C.-Y. Huang, C.-H. Lin, Y.-L. Tsai, and C.-M. Yu, "Safe LoRA: The silver lining of reducing safety risks when finetuning large language models," in *Proc. Adv. Neural Inf. Process. Syst.*, 2024, pp. 65072–65094.

[36] A. Djuhera, S. R. Kadhe, F. Ahmed, S. Zawad, and H. Boche, "Safemerge: Preserving safety alignment in fine-tuned large language models via selective layer-wise model merging," in *Proc. ICLR Workshop Building Trust LLMs*, 2025, pp. 200–213. [Online]. Available: https://research.ibm.com/publications/safemerge-preserving-safetyalignment-in-fine-tuned-large-language-models-via-selective-layer-wise-model-merging

[37] S. Floyd, D. K. K. Ramakrishnan, and D. L. Black, *The Addition of Explicit Congestion Notification (ECN) to IP*, document RFC 3168, Sep. 2001. [Online]. Available: https://www.rfc-editor.org/info/rfc3168
[38] G. Fairhurst and M. Welzl, *The Benefits of Using Explicit Congestion Notification (ECN)*, document RFC 8087, Mar. 2017. [Online]. Available: https://www.rfc-editor.org/info/rfc8087
[39] OpenAI.(2024). *ChatGPT*. [Online]. Available: https://chat.openai.com/chat
[40] A. Roberts et al., "Exploring the limits of transfer learning with a unified text-to-text transformer," Tech. Rep., 2019.

Shiva Raj Pokhrel (Senior Member, IEEE) received the Ph.D. degree in engineering from Swinburne University of Technology in 2017. He is currently with the School of IT, Deakin University, Australia, leading IoT and software Engineering Research and recognized leadership in Congestion Control, TCP, mobile computing, quantum technologies, and distributed machine learning systems. His career spans academia and industry, including a Research Fellowship with The University of Melbourne and seven years as a Systems Engineer, combining strong theoretical foundations with large-scale experimental expertise. His research focuses on federated learning, distributed quantum computing, quantum communication, and scalable ML/DL systems, including distributed training and hyperparameter optimization across cloud and HPC environments using platforms, such as IBM and Amazon Quantum Systems. He works across congestion control, multipath internet, semantic communication, federated learning, quantum machine learning, 6G networks, the IoT, and cyber-physical systems, with applications in smart cities, autonomous systems, satellite networks, and intelligent manufacturing. He is a Marie Curie Fellow from 2017 to 2022. He has led multiple projects, supervised M.Sc. and Ph.D. researchers, serves as an Editor for IEEE TRANSACTIONS ON VEHICULAR TECHNOLOGY, and actively contributes to the research community through editorial roles and leadership in major venues, including IEEE INFOCOM, IEEE GLOBECOM, IEEE ICC, and ACM MOBICOM.

Deol Satish is currently pursuing the degree (Hons.) in software engineering with Deakin University with strong technical and programming expertise. He is a Research Assistant with the IoT Research Laboratory, Geelong, VIC, Australia. His work focuses on congestion control and AI/ML-driven networking systems, with experience spanning kernel development, cloud computing, and advanced network technologies.

Jonathan Kua (Member, IEEE) received the B.Eng. degree (Hons.) in telecommunications and network engineering and the Ph.D. degree in telecommunication engineering from the Swinburne University of Technology, Australia, in 2014 and 2019, respectively. He is currently a Senior Lecturer in Internet of Things with the School of Information Technology, Deakin University, Australia. His research spans computer systems and data networking, with a focus on low-latency transport protocols, adaptive multimedia streaming, content delivery, network measurement, and bottleneck queue management. He also works on the IoT, industry 4.0, distributed and networked systems, and industrial cyber-physical systems, aiming to enhance network performance, user experience, and quality of service. He is a member of IEEE TCCC, ACM, and ACM SIGMM. He has served as a Guest Editor for IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS (JSAC) Special Issue on Co-Design of Communication, Computing, and Control in Industrial Cyber-Physical Systems.

Anwar Walid (Fellow, IEEE) received the B.S. degree in electrical engineering from New York University and the Ph.D. degree in electrical engineering from Columbia University, USA. He is an Adjunct Professor with the Department of Electrical Engineering, Columbia University. He previously led the Mathematics of Systems Research Department, Nokia Bell Labs, and has held senior roles in industry, including at Amazon Science. He has been teaching courses on machine learning for networks and large-scale content systems with Columbia University, since 2009. His research spans networking systems, machine learning for networks, and large-scale content delivery. He is an ACM Fellow. He is also an Elected Member of Tau Beta Pi and IFIP Working Group 7.3. He has received several prestigious awards, including the IEEE Communications Society William R. Bennett Prize, the ACM SIGCOMM Networking Systems Award, the ACM SIGMETRICS Best Paper Award, and the IEEE INFOCOM Test of Time Paper Award. He has served as an Associate Editor for leading journals, such as IEEE/ACM TRANSACTIONS ON NETWORKING and IEEE TRANSACTIONS ON CLOUD COMPUTING. He is a Senior Editor of IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS.