Adaptive Request Scheduling and Load Balancing for Edge Deployed Large Language Models

Fangyi Mou, Zhiqing Tang, *Member, IEEE*, Weijia Jia, *Fellow, IEEE*, and Wei Zhao, *Fellow, IEEE*

***Abstract*—The inference services of Large Language Models (LLMs) play a crucial role in many fields. Deploying LLMs at the network edge can effectively reduce response latency and enhance domain-specific knowledge. However, due to the dynamic and unpredictable nature of edge user requests and the limited resources of edge servers, improper request scheduling can lead to server load imbalance and increased inference latency. To address this challenge, we model edge LLM inference under dynamic workloads and constrained edge resources by fully considering mutual influences and trade-offs between inference performance, inference latency, and edge resource utilization. We propose a Workload Prediction and Dynamic Request Scheduling (WPDS) algorithm for edge computing environments. The WPDS algorithm first evaluates and prioritizes heterogeneous requests by extracting features and importance scores from user requests. A Transformer-based encoder is used to extract load characteristics of edge servers over different time periods to predict future GPU resource demands. Then, a soft actor-critic reinforcement learning model dynamically schedules requests to appropriate LLM instances, capturing the dynamic relationship between request processing and edge resource utilization from complex state spaces. We evaluate our algorithm in a real Kubernetes-based edge inference prototype system. Experimental results indicate that our approach reduces average inference time by approximately 16.6% compared to the best-performing heuristic baseline algorithm, while also achieving more balanced GPU utilization across edge servers.**

***Index Terms*—Edge computing, load balancing, task scheduling, deep reinforcement learning, LLM inference.**

Received 8 April 2025; revised 19 December 2025; accepted 6 February 2026. Date of publication 16 February 2026; date of current version 10 April 2026. This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant 62272050, and Grant 62302048, in part by the Guangdong Key Lab of AI and Multi-modal Data Processing, United International College (UIC), Zhuhai under Grant 2023-2024, in part by the Guangdong Provincial Department of Education, in part by Institute of Artificial Intelligence and Future Networks (BNU-Zhuhai) and Engineering Center of AI and Future Education, Guangdong Provincial Department of Science and Technology, China;Zhuhai Science-Tech Innovation Bureau under Grant 2320004002772, and in part by the Interdisciplinary Intelligence SuperComputer Center of Beijing Normal University, Zhuhai, China. *(Corresponding authors: Zhiqing Tang; Weijia Jia.)*

Fangyi Mou is with Hong Kong Baptist University and Beijing Normal-Hong Kong Baptist University, Zhuhai 519087, China (e-mail: moufangyi@bnbu.edu.cn).

Zhiqing Tang is with the Institute of Artificial Intelligence and Future Networks, Beijing Normal University, Zhuhai 519087, China (e-mail: zhiqingtang@bnu.edu.cn).

Weijia Jia is with the Institute of Artificial Intelligence and Future Networks, Beijing Normal University, Zhuhai 519087, China, and also with the Guangdong Key Lab of AI and Multi-Modal Data Processing, Beijing Normal-Hong Kong Baptist University, Zhuhai 519087, China (e-mail: jiawj@bnu.edu.cn).

Wei Zhao is with the Shenzhen University of Advanced Technology, Shenzhen 518055, China (e-mail: weizhao86@outlook.com).

Digital Object Identifier 10.1109/TSC.2026.3665250

I. Introduction

In RECENT years, Large Language Models (LLMs) have been growing exponentially [1], jumping from millions to billions of parameters. This surge has significantly boosted the ability of LLMs to process and generate natural language, making them useful in various fields like online question answering systems and smart assistants [2]. With a massive increase in edge users on networks, there is a rising demand for quicker response times and personalized experiences from LLMs [2], [3]. As a result, a lot of research has focused on compressing or optimizing these models [4], [5], [6] using techniques like model pruning, quantization, and knowledge distillation, successfully deploying them on edge servers. By providing LLM inference services at the edge, we can effectively tackle complex edge intelligence tasks, such as inspection robots, traffic flow prediction and control, etc.

However, edge user requests are highly diverse and often latency-sensitive, making effective scheduling challenging [2], [7], [8]. Furthermore, edge servers operate under limited resources and handle fewer concurrent tasks compared to cloud data centers [9], [10], [11]. Edge-based LLM deployment faces three key challenges: (1) *High-volume and lengthy requests*, where resource-constrained edge devices struggle with complex inputs, leading to memory pressure or performance degradation [12], [13]; (2) *Load imbalance from heterogeneous workloads*, as simplistic scheduling strategies (e.g., proximity-based) often overload some servers while leaving others underutilized, resulting in poor response times [14]; and (3) *Service interruptions and fault tolerance*, since Rapid shifts in dynamic demand and limited resources increase vulnerability to workload bursts and node failures [7], [8], creating a need for robust, adaptive scheduling mechanisms that can effectively handle dynamic prioritization, share constrained resources, and facilitate multi-server collaboration for edge-based LLM inference. Consequently, efficient request scheduling and load balancing are essential for edge LLM inference.

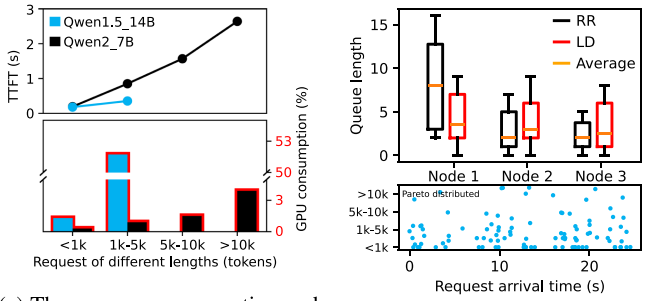
To optimize request scheduling and load balancing, the following challenges need to be addressed. *The first challenge is how to assess the concurrency of edge requests and their processing complexities.* Efficient edge LLM inference services require consideration of user request-response latency and recognition of characteristics differentiating them from typical edge services [5], [15]. Edge users submit requests of varying complexities, such as question answering, reading comprehension, and translation tasks, resulting in significant differences in input lengths, latencies, and resource demands [16]. For example, common-sense requests may involve brief questions, while reading comprehension requires handling much longer passages. Different request types also have varying latency requirements. Therefore, a dynamic request priority adjustment mechanism is

essential for scheduling diverse requests to suitable LLMs and ensuring timely processing, thereby improving both completion rates and accuracy.

Moreover, *the second challenge is how to address edge workload imbalance and unpredictable resource consumption*. Users generate highly time-varying requests, leading to significant workload disparities across distributed edge nodes [17]. Traditional scheduling algorithms, such as weighted round-robin and least response time [11], [18], are not well-suited for edge-based LLM inference. Recent resource-aware edge inference systems have explored distributed LLM serving at the edge [19], [20]. However, these approaches mainly rely on reactive resource-state awareness, causing scheduling decisions to lag behind rapidly changing request queues under heterogeneous workloads [21]. In practice, LLM inference tasks exhibit highly diverse input lengths and resource demands, while GPU memory and storage consumption at the edge vary significantly with request characteristics. Furthermore, robust fault-tolerance mechanisms are required to maintain inference availability in the presence of node or model failures [14].

In this paper, we propose an adaptive request scheduling and load balancing approach to enhance LLM inference on edge devices under large-scale user requests. We first model the edge-based LLM inference system aiming to improve user satisfaction and reduce response latency. By fully analyzing LLM request characteristics, we extract features related to high concurrency and varying request types and dynamically assign priority scores. Next, we use four Transformer-based encoders to capture temporal dependencies in historical edge LLM resource utilization data [22]. These encoders enable the model to understand both short-term failures, sudden load fluctuations, and long-term load balancing characteristics. We then employ an actor-critic method [23] to learn a load-balancing strategy, capturing the dynamic relationship between request scheduling, processing, and resource utilization in complex state space. Additionally, we introduce an edge inference recovery mechanism that reschedules pending requests by considering priority scores and queuing delays during LLM pod recovery. Finally, we evaluate the performance of our algorithm using various large-scale requests in Kubernetes-based edge systems. The results show that our algorithm reduces average inference time by approximately 16.6% compared to baselines. Our main contributions are as follows:

- We address the significant workload imbalance and inference latency resulting from processing large-scale diverse requests in an edge-based LLM inference system. Our comprehensive model fully incorporates request scheduling, inference recovery, and workload balancing, accounting for both request completion and load balancing costs.
- We propose WPDS, an actor critic-based algorithm for **w**orkload **p**rediction and **d**ynamic request **s**cheduling. By extracting and scoring request features, we evaluate and prioritize different types of requests. Using Transformer-based encoder, we extract workload features of edge servers over various time periods to represent future resource demands.
- We conduct experiments on real Kubernetes-based edge systems, using various language datasets to evaluate request processing performance under load balancing. The results show that our algorithm outperforms traditional baseline algorithms by approximately 16.6% on average.



(a) The resource consumption and computation time for different request lengths on two different models.

(b) The queue length for different nodes of two different scheduling algorithms.

Fig. 1. Motivation studies.

The remainder of the paper is organized as follows. In Section II, the related work and points of motivation are illustrated. In Sections III-A and V, the system model and the proposed algorithm are introduced. In Section IV, the system implementation and the WPDS algorithm are described. The results of performance are evaluated in Section VI. In Section VII, the conclusion and future works are given.

II. Related Works and Motivation

A. Handling High-Volume and Lengthy Requests

The increasing complexity and length of user requests processed by LLMs pose significant challenges in resource-constrained environments. As shown in Fig. 1(a), we analyze two different LLMs deployed on identical hardware with the same request workloads, and record performance under varying input lengths. Due to GPU memory limitations on edge devices, the two models demonstrate different capacities in handling long-sequence inputs. Specifically, we measure the Time to First Token (TTFT) for each model and visualize GPU memory consumption during inference, collected via Prometheus [24] integrated with our Kubernetes-based deployment. The figure illustrates how GPU utilization and inference latency increase significantly with longer input sequences, highlighting the need for length-aware request scheduling. For shorter requests, both TTFT and GPU usage remain relatively low, whereas longer inputs cause sharp growth in both metrics, indicating increased strain on edge hardware.

To tackle this challenge, numerous studies have introduced optimization strategies at both the model and system levels. Model-level optimization techniques, such as model compression and quantization [4], [6], aim to reduce the computational and memory demands of LLMs without sacrificing their ability to handle lengthy inputs. System-level strategies focus on resource allocation, model parallelization, and infrastructure optimization for LLM deployment [5], [16], [19], [25]. Liu et al. [6] present an efficient context-loading module for LLMs to reduce key-value (KV) cache size and loading delays. Xiao et al. [4] introduce a training-free post-training quantization method that achieves 8-bit quantization in LLMs. Sheng et al. [5] employ advanced weight and attention cache compression techniques for LLM inference. Miao et al. [19] present a cost-effective LLM cloud serving approach and dynamically adjust LLM parallelization configurations. Lin et al. [25] integrate semantic

load balancing and shared KV caching to reduce re-computation during LLM inference. Crankshaw et al. [26] proposed a system named InferLine, which provisions and manages multi-stage ML prediction pipelines to meet end-to-end tail latency goals while minimizing cost. By optimizing model architecture and parameterization, these approaches extend the capability of resource-constrained systems, enabling them to process longer requests efficiently.

However, these methods still struggle to meet the challenges of edge clusters running multiple LLM services, particularly under dynamic and high-volume workloads. Existing approaches often rely on static parallelization strategies, limiting efficient resource use across nodes when managing large volumes of variable-length requests. Such variations in request lengths, resource consumption, inference times, and the processing capabilities of models on different servers frequently result in load imbalances within edge clusters. Consequently, some servers become overloaded, leading to increased inference latency, while others remain underutilized, reducing overall resource efficiency.

This motivates the need for an adaptive scheduling mechanism that not only accommodates diverse request characteristics but also balances workloads across edge clusters, maximizing resource utilization and minimizing inference latency.

B. Towards Imbalanced Request Distribution

The dynamic nature of user requests, including varying arrival rates and processing demands, poses significant challenges for efficient edge-based LLM request scheduling. Differences in node processing capabilities also lead to performance variation among deployed LLMs. Traditional Kubernetes platforms rely on scheduling algorithms such as weighted round-robin or least response time, forwarding requests without considering the real-time conditions of edge-based LLM deployments. Fig. 1(b) compares queue lengths under two scheduling algorithms with identical workloads. The lower part of the figure shows requests sampled from a Pareto-distributed workload generated from the combined dataset. Queue lengths at each node reflect request processing efficiency under the same input workload. These results demonstrate that conventional schedulers fail to address the heterogeneous nature of edge-based LLM inference requests.

Inefficiency is particularly evident in LLM deployments, where requests of varying lengths entail differing GPU memory and computational demands. Traditional algorithms generally assume uniform resource requirements, neglecting the substantial variations caused by text length and complexity. As a result, these methods lead to unbalanced node utilization, wasting resources and increasing inference latency.

To enhance LLM efficiency on edge devices, researchers have explored adaptive compression and fine-tuning [15], distributed inference systems [7], [14], and request rescheduling to improve load balancing and reduce latency [8]. Yu et al. [15] achieve notable speedups and memory reduction through hierarchical unified compression. Ma et al. [7] introduce a pipeline inference framework for efficient deployment across heterogeneous devices. Borzunov et al. [14] present a decentralized system using fault-tolerant inference and load-balancing protocols for efficient LLM execution. Sun et al. [8] propose real-time migration mechanisms that reschedule requests among LLM instances to reduce latency.

These approaches improve model efficiency and dynamic scheduling but mainly optimize individual nodes or static inference setups. They do not fully address challenges in large-scale, multi-node edge clusters handling dynamic, high-volume requests with varying characteristics. This highlights the need for a scalable scheduling mechanism that adaptively allocates resources based on request features while balancing workloads across the cluster.

C. Service Interruptions in Edge-Based LLM Deployments

High-load conditions in edge-based LLM deployments often cause service interruptions due to network congestion, cache failures, or node outages. Such interruptions severely affect the system's ability to provide efficient inference, especially in dynamic, large-scale edge environments. Robust fault-tolerance mechanisms are essential to ensure service continuity during node failures.

Traditional methods often struggle in these situations. For example, reactive fault recovery can cause significant delays, reducing system responsiveness and efficiency. Miao et al. [19] propose state inference recovery for incremental submission of inference progress, enabling low-cost task resumption in SpotServe. While effective in preemptive environments, it relies on static resource configurations and specific deployment conditions. Fang et al. [27] introduce an LLM-based edge fault-tolerant paradigm combining fault perception, diagnosis, and recovery with pre-optimized knowledge. However, this approach mainly addresses network-level issues, not fully handling the complexities of dynamic, large-scale LLM requests.

These limitations highlight the need for an edge-cluster fault-tolerance mechanism that ensures system reliability under large-scale, dynamic requests. Such a mechanism should adapt to varying request characteristics and node conditions, enabling uninterrupted service even when multiple LLM nodes fail. Robust fault-tolerant strategies are essential to maintain inference efficiency and optimize resource utilization in edge-based LLM deployments.

In summary, these results indicate that the complexity and diversity of user requests affect resource usage and inference time in edge-based LLM deployments. By monitoring and predicting resource usage at each node in real time, the system can optimize scheduling based on request characteristics, reducing inference latency, preventing node overloads or idling, and improving overall stability and performance. Therefore, we design WPDS as follows: a) Request Scoring: Prioritize requests based on their length and completion time; b) Resource Utilization Extraction: Monitor GPU usage at intervals to predict node workload; c) Service Recovery: Reschedule pending requests during LLM deployment recovery.

III. System Model and Problem Formulation

A. System Description

Fig. 2 depicts the workflow of the edge-based LLM inference system, comprising user requests, LLM deployments, and edge devices. User requests are batched and prioritized based on completion time constraints and request features. Resource utilization is monitored and encoded using a temporal fusion self-attention mechanism to capture usage patterns across edge-deployed LLMs. To optimize resource usage and prevent overloading, the system dynamically balances workloads among

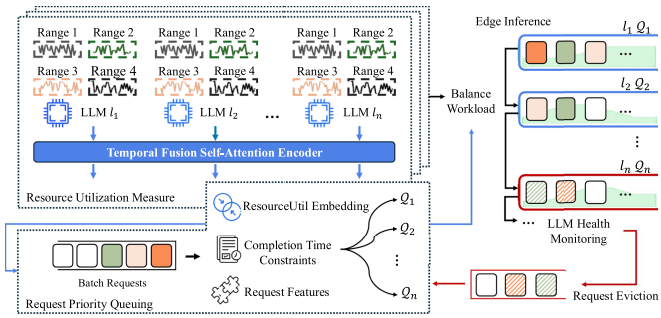


Fig. 2. An illustration of edge-based LLM inference system.

TABLE I
NOTATIONS

$\mathbf{K}$	User request set
k_i	$k^{i_{\text{th}}}$ user request ($k \in \mathbf{K}, 1 \leq i \leq \lvert\mathbf{K}\rvert$)
ℓ_k	Token length of user request k_i
$n_t^{k_i}$	Timestamp of user request k_i
z_{k_i}	Completion constraint of user request k_i
u_i	Expected answer of user request k_i
$\mathbf{L}$	LLM deployment set
l	An LLM deployment ($l \in \mathbf{L}$)
q_l^r	Replicas of LLM deployment
q_l^s	Selector label of LLM deployment
ℓ_l	Maximum model length of LLM deployment
m_l	GPU memory of LLM deployment
c_l	GPU cores of LLM deployment
$\mathcal{Q}_i$	Processing queue of LLM deployment ($1 \leq i \leq \lvert\mathbf{L}\rvert$)
$\mathbf{E}$	Edge device set
e_j	$e^{j_{\text{th}}}$ edge device ($e \in \mathbf{E}, 1 \leq j \leq \lvert\mathbf{E}\rvert$)
m_{e_j}	Total GPU memory of edge device e_j
c_{e_j}	Total GPU cores of edge device e_j
b_{e_j}	Storage space of edge device e_j
v_{e_j}	GPU computational capacity of edge device e_j
$\mathcal{C}_{k_i}^{inf}(t)$	Processing cost of request
$\mathcal{C}_{k_i}^{que}(t)$	Queuing cost of request
$\mathcal{C}_{k_i}^{tsk}(t)$	Completion cost of request
$\mathcal{C}_{k_i}^{iac}(t)$	Prediction cost of request
$\mathcal{C}^{lod}(t)$	Load-balancing cost at time t
$\mathcal{C}(t)$	Total cost at timestep t
s_t	Global system state at time t
s_t^l	State of LLM deployment l at time t
s_t^k	State associated with request k at time t
s_t^n	Resource predictive encoder representations at time t
$H_{l_j}(t)$	Health status of LLM l_j at time t
$B_{l_j}(t)$	Cumulative restart count of LLM l_j up to time t
tol_i	Tolerance value of request k_i
a_t	Scheduler decision action at time t
ν	Sequence of trajectory
$\mathcal{D}$	Replay buffer storing trajectories

edge devices. Deployment health is monitored, and unresponsive or overloaded deployments trigger a request eviction mechanism to maintain stability. Scheduling decisions focus on minimizing delays while ensuring high-quality LLM inference. Key notations are summarized in Table I.

User requests: A set of user requests, $\mathbf{K} = \{k_1, k_2, \ldots, k_{|\mathbf{K}|}\}$, is generated from a large user base and assigned to edge devices for processing, where $|\mathbf{K}|$ is the total number of requests. The characteristics of each request k_i are defined as $k_i \triangleq \{\ell_{k_i}, n_t^{k_i}, z_{k_i}, u_i\}$, for $1 \leq i \leq |\mathbf{K}|$, $t \in \mathbf{T}$. Specifically, ℓ_{k_i} is the token length of the request, $n_t^{k_i}$ is the timestamp t at which the request is generated, z_{k_i} is the completion time constraint, specifying the desired maximum time by which the request should be completed. u_i is the expected response to the request from the dataset.

Edge devices: The set of edge devices is denoted as $\mathbf{E} = \{e_1, e_2, \ldots, e_{|\mathbf{E}|}\}$, where $|\mathbf{E}|$ is the total number of edge devices. The attributes of e_j are $e_j \triangleq \{m_{e_j}, c_{e_j}, b_{e_j}, v_{e_j}\}$, for $1 \leq j \leq |\mathbf{E}|$, where m_{e_j} is the total GPU memory and c_{e_j} is the number of virtual GPU (vGPU) cores available on the edge device. Based on a Kubernetes-based inference platform, GPU virtualization supports concurrent execution of multiple models on a single physical GPU [28], [29]. b_{e_j} is the storage space that provides persistent storage for LLM model weight files, configuration files, and logs. Due to the limitations of edge computing resources, the number of pods running simultaneously on each device is limited. v_{e_j} is the computational capacity of the edge device. After scheduling requests, the LLM deployment on an edge device for a request is $e_j = \{k_i^{j,l} | 1 \leq j \leq |\mathbf{E}|, l \in \mathbf{L}\}$, where $k_i^{j,l} = 1$ if request k_i is processed by LLM l on edge device e_j, otherwise, $k_i^{j,l} = 0$.

LLM deployments: A set of LLMs $\mathbf{L} = \{l_1, l_2, \ldots, l_{|\mathbf{L}|}\}$ are deployed as inference services on heterogeneous edge devices to process user requests, with $|\mathbf{L}|$ indicating the total number of LLMs. Deployments can be horizontally scaled by creating multiple replicas on the same devices sharing model files. The attributes of a deployment are denoted as $l_i \triangleq \{e_j, q_{l_i}^r, q_{l_i}^s, \ell_{l_i}, m_{l_i}, c_{l_i}, \mathcal{Q}_i\}$, for $1 \leq i \leq |\mathbf{L}|$, where e_j, $1 \leq j \leq |\mathbf{E}|$ is the device on which the LLM deployed. $q_{l_i}^r$ is the number of replicas, $q_{l_i}^s$ is the selector, and ℓ_{l_i} is the maximum model length of the LLM deployment. GPU memory m_{l_i} and cores c_{l_i} are the main resource consumptions, and $\mathcal{Q}_i$ is the processing queue.

B. System Cost

1) LLM Completion Costs: The cost of completing a request involves multiple components: the computational cost incurred by utilizing edge-based LLMs $\mathcal{C}_{k_i}^{inf}(t)$, the queuing cost $\mathcal{C}_{k_i}^{que}(t)$ arising from delays in the processing pipeline, and the completion cost $\mathcal{C}_{k_i}^{tsk}(t)$, which includes penalties for unmet task completion constraints. These factors collectively influence the prioritization of incoming requests. The prioritization mechanism evaluates each request based on its task completion time constraints, queuing delays, and the computational effort required, enabling efficient task allocation and scheduling.

Cost of request processing: The processing cost of requests is influenced by various factors, including resource contention among LLMs deployed on the edge device, input request length, and the processing capabilities of the LLMs being used. The processing cost of the request $k_i \in \mathbf{K}$ when served by LLM $l_j \in \mathbf{L}$ of edge device $e_j \in \mathbf{E}$ is defined as follows [30]:

$$\mathcal{C}_{k_i}^{inf}(t) = \int_{n_s^{k_i}}^{t} \frac{k_i^{e_j,l_j}(\tau) \cdot \ell_{k_i}(\tau)}{\varsigma_{l_j}} d\tau, n_s^{k_i} \leq t \leq n_d^{k_i}, \quad (1)$$

where $n_s^{k_i}$ and $n_d^{k_i}$ denote the timestamps at which the request begin processing and completes, respectively. The binary indicators variable $k_i^{e_j,l_j}(\tau)$ specifies whether request k_i is being served by LLM l_j on edge device e_j at time τ. Multiple LLMs

can be deployed on a single edge device and share GPU resources, whose effective computational capacity is represented as $\varsigma_{l_j} = v_{e_j} \cdot \frac{m_{l_j}}{m_{e_j}} \cdot \frac{c_{l_j}}{c_{e_j}}$. Requests k_i of different lengths ℓ_{k_i} are scheduled to LLM services l_j, each with varying computational capabilities.

Cost of queueing requests: The queueing cost is associated with the waiting time and delay experienced by requests before processing. It reflects the system's efficiency in handling incoming tasks. Each request k_i has a tolerance value κ_{k_i}, which influences its position in the processing queue $\mathcal{Q}_j, 1 \leq j \leq |\mathbf{L}|$ on edge device e_j. In addition, the potential for rescheduling requests across different LLM service queues should be considered. The accumulation of queueing cost for a request $k_i \in \mathbf{K}$ is defined as follows [31]:

$$\mathcal{C}_{k_i}^{que}(t) = \sum_{j=1}^{|\mathbf{L}|} \int_{n_0^{k_i}}^{t} \frac{\zeta_{k_i}(\tau)}{\alpha \cdot \kappa_{k_i}} \cdot k_i^{e_j,l_j}(\tau) d\tau, n_0^{k_i} \leq t \leq n_s^{k_i}, \quad (2)$$

where $n_0^{k_i}$ is the timestamp when the request k_i arrives at the selected LLM service, and $\zeta_{k_i}(\tau) = \tau - n_0^{k_i}$ denotes the elapsed waiting time of request k_i in the processing queue at time τ. The constant α scales the effect of the tolerance value.

Cost of request completion: Each request k_i has a completion deadline z_{k_i}, which specifies the maximum allowable end-to-end service latency. The completion cost is defined as a latency violation penalty of request k_i and is given by:

$$\mathcal{C}_{k_i}^{tsk}(t) = \max\left(0, \left(\mathcal{C}_{k_i}^{inf}(t) + \mathcal{C}_{k_i}^{que}(t)\right) - z_{k_i}\right). \quad (3)$$

Moreover, the accuracy of responses plays a crucial role in determining user satisfaction. The cost associated with inaccuracy inference for request k_i is defined as [32]:

$$\mathcal{C}_{k_i}^{iac}(t) = 1 - \frac{L(\hat{y_{k_i}}(t), y_{k_i})}{\max(|y_{k_i}|, |\hat{y_{k_i}}(t)|)}, \quad (4)$$

where y_{k_i} and $\hat{y_{k_i}}(t)$ are the actual and predicted sequence for the request k_i respectively. $|\cdot|$ represents the length of the sequence. $L(\hat{y_{k_i}}(t), y_{k_i})$ is the Levenshtein distance between the actual and predicted sequences.

2) Load-Balancing Cost: The workload cost $\mathcal{C}^{lod}$ of the cluster over a time period is defined as:

$$\mathcal{C}^{lod}(t) = \sum_{i=1}^{|\mathbf{K}|} \sum_{j=1}^{|\mathbf{L}|} \int_{n_0^{k_i}}^{t} \left(\varepsilon \cdot \rho_{l_j}(\tau) + c_{l_j}(\tau)\right) \cdot x_{k_i}^{e_j,l_j}(\tau)\, d\tau, \quad n_0^{k_i} \leq t \leq n_d^{k_i}, \quad (5)$$

where $\rho_{l_j}(\tau)$ denotes the real-time GPU utilization of LLM deployment l_j at time τ, which is captured by Prometheus monitoring system. Consequently, LLM deployments with lower GPU load are more likely to be assigned incoming requests. The scaling factor ε controls the relative importance between this scheduling preference and the initialization cost $c_{l_j}(\tau)$. $c_{l_j}(\tau)$ captures the actual overhead among the replicaset values $q_j^r(\tau), \ldots, q_{j+M-1}^r(\tau)$ of the M LLM deployments hosted on edge device e_j at time τ. This cost is measured by monitoring Kubernetes pod lifecycle events via the Watch API, reflecting the overhead of loading LLMs into memory on the servers.

C. Problem Formulation

Our objective is to minimize the total system cost $\mathcal{C}(t)$ by finding the optimal scheduling decisions for processing the incoming requests on the edge-based LLM inference system. The total cost of WPDS at time t is defined as:

$$\mathcal{C}(t) = C_{k_i}^{tsk}(t) + C_{k_i}^{iac}(t) + C^{lod}(t). \quad (6)$$

The optimization of Function (6) aims to balance the completion costs of multiple LLM services with the costs associated with load balancing. From the perspective of edge service operations, our approach enhances cost-effectiveness by optimizing per-request completion cost and achieving effective load balancing across heterogeneous edge servers. Over time, this objective will reduce the total costs of task scheduling and load balancing to a minimum, ensuring the speed of overall system inference and the efficient utilization of resources.

Constraints: Each user request has different characteristics in a dynamically complex inference system, and the nodes possess varying processing capabilities. To ensure the predictability of the system, we define the constraint of matching the length of requests with the processing capabilities of the node model as follows:

$$\sum_{l \in \mathbf{L}} k_i^{j,l} \cdot \ell_{k_i} \leq \min(v_{e_j}(t), \ell_{l_i}), \quad \forall 1 \leq i \leq |\mathbf{K}|, 1 \leq j \leq |\mathbf{E}|. \quad (7)$$

During the request scheduling process, it is essential to ensure that the selected LLM services are in a healthy state and have adequate resources and stability to handle the requests. To achieve this, the system implements an inference recovery mechanism. If a target node is evaluated as unhealthy at time t, the corresponding LLM services will be deemed unreachable, and the request will be denied. This can be denoted as:

$$k_i^{j,l} \leq H_l(t) \cdot q_{l_i}^r, \quad \forall 1 \leq i \leq |\mathbf{K}|, 1 \leq j \leq |\mathbf{E}|, l \in \mathbf{L}, \quad (8)$$

where $H_l(t)$ is the health status indicator of LLM deployment l at time t. $H_l(t) = 1$ if the LLM service is healthy, and $H_l(t) = 0$ otherwise. An user request k_i is only assigned to an LLM service l on edge device e_j if the LLM service is healthy ($H_l(t) = 1$) and has at least one active replica ($q_{l_i}^r > 0$).

Problem: (WPDS Problem) For each time t, the objective is to minimize the total system cost $\mathcal{C}(t)$ by optimizing the scheduling decisions for processing incoming requests on the edge-based LLM inference system. This optimization ensures efficient resource utilization and rapid system inference, while constraints are imposed to match request characteristics with node processing capabilities. The problem formulation is defined as follows:

Problem 1: $\min \mathcal{C} = \sum_{t \in T} \mathcal{C}(t)$,

s.t. Eqs. (7), (8),

$$k_i^{j,l} \in \{0, 1\}, \forall 1 \leq i \leq |\mathbf{K}|, \forall 1 \leq j \leq |\mathbf{E}|, \forall l \in \mathbf{L}.$$

By solving this optimization problem, we aim to achieve a balanced and efficient sechduling of requests to LLM services, ensuring both high system performance and robustness in dynamic environments.

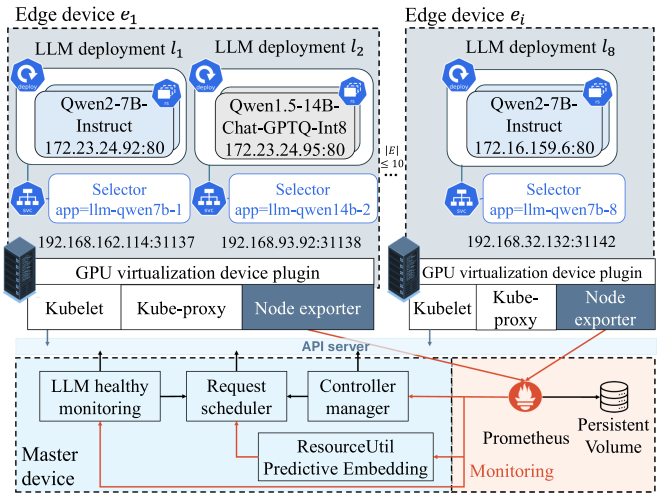


Fig. 3. An implementation of edge-based LLM inference.

IV. System Implementation

System Implementation: As shown in Figs. 3 and 2, we have implemented a prototype of the WPDS within a Kubernetes-based edge inference system. It includes edge servers, LLM deployments and services, a global load-balancing controller, a unified GPU virtualization scheduler, and a Prometheus server. The edge servers are configured as follows: an Intel i7-13700KF 16-core CPU with an NVIDIA RTX 4070Ti GPU; an Intel i9-14900KF 24-core CPU with an NVIDIA RTX 4090 GPU; and an Intel Xeon Platinum 8358 CPU with two NVIDIA A800-SXM4-80 GB GPUs.

The LLM service is provided via containers deployed through Kubernetes Deployment and discovered using selector labels. Specifically, Qwen2-7B models are deployed as native PyTorch deployments based on vLLM [16], and require approximately 16 GB of GPU memory for standard inference. Qwen1.5-14B-Int8 models use dependencies supporting INT8 operations, including the appropriate PyTorch version and the GPTQ library [33], and require around 22 GB of GPU memory during inference due to the trade-off between precision and memory usage. All model deployments are scaled using Kubernetes ReplicaSets and utilize a GPU virtualization device plugin [29] for efficient GPU resource utilization and parallel deployment. Prometheus [24] monitors the health and resource usage of nodes and deployments in the edge-based LLM inference cluster, exposing metrics via the Node Exporter.

Dataset: We construct a large-scale request dataset by combining various LLM test datasets, including three short-request datasets (MRPC [34], Commonsense QA [35], and Social IQA [36]) and five long-request datasets (Long Bench [37], Infinite Bench [38], RULER [39], L-Eval [40], and LooGLE [41]). The dataset encompasses both short and long requests across multiple domains and complexity levels, enabling evaluation in varied task complexities. To simulate request distribution patterns under both uniform and unevenly high-load conditions, we generate arrival distributions of requests using a Normal distribution (mean $=1$, standard deviation $=2$) and a Pareto distribution with intervals ranging from 3×10^{-3} to 30 seconds.

V. Algorithms

A. Algorithm Settings

In edge-based LLM inference systems, resource utilization is highly dynamic, and system states are often partially observable, making it difficult to extract informative representations from noisy, high-dimensional data. To address this issue, we formulate WPDS problem as a Partially Observable Markov Decision Process (POMDP), which captures partial observability and temporal state transitions to support informed scheduling and workload balancing decisions.

This formulation consists of the following 7-tuple components: $\{\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{T}, \mathcal{Z}, r, \gamma\}$, where $\mathcal{S}$ represents the set of states; $\mathcal{A}$ is the set of actions; $\mathcal{O}$ is the set of observations of the GPU resource utilization of the LLM deployments, collected at different time intervals; $\mathcal{T}(s'|s,a)$ is the transition model specifying the probability of moving from state s to state s' under action a; $\mathcal{Z}(o|s',a)$ is the observation model specifying the probability of receiving observation o given state s' and action a; $r(s,a)$ is the reward function defining the expected immediate reward for executing action a in state s; $\gamma \in (0,1)$ is the discount factor. The details are as follows.

Compositional state space ($\mathcal{S}$): The state space of WPDS problem is composed of the status of LLM deployments s_t^l, request characteristics s_t^k, and the resource predictive encoding representations s_t^n. The state s_t^l at time t includes the following components: the deployment IDs $\mathbf{L}$, the model lengths $\boldsymbol{\ell}_l$, the lengths of processing queues $|\mathcal{Q}_l(t)|$, the GPU resources $m_l(t)$ and $c_l(t)$, as well as the health status $H_l(t)$ and the restart count $B_l(t)$ of all LLM deployments in the environment, which is defined as:

$$\begin{aligned} s_t^l &= \{\mathbf{L}, \boldsymbol{\ell}_l, |\mathcal{Q}_l(t)|, m_l(t), c_l(t), H_l(t), B_l(t)\} \\ &= \big\{l_1, \ldots, l_{|\mathbf{L}|}, \ell_{l_1}, \ldots, \ell_{l_{|\mathbf{L}|}}, |\mathcal{Q}_{l_1}(t)|, \ldots, |\mathcal{Q}_{l_{|\mathbf{L}|}}(t)|, \\ &\quad m_{l_1}, \ldots, m_{l_{|\mathbf{L}|}}, c_{l_1}, \ldots, c_{l_{|\mathbf{L}|}}, H_{l_1}, \ldots, H_{l_{|\mathbf{L}|}}, \\ &\quad B_{l_1}, \ldots, B_{l_{|\mathbf{L}|}}\big\}, \end{aligned} \tag{9}$$

where $H_l(t)$ represents the health status of the LLM deployment monitored through the Kubernetes watching function. This health status can be categorized into two types: "running," which indicates that the service is operating normally, and "failed," which indicates that the LLM deployment has encountered a fault that needs to be addressed. $B_l(t)$ denotes the number of restarts that the LLM deployment has undergone during its operation.

Moreover, the state of the current processing request s_t^k encompasses the characteristics and completion costs associated with the request, which can be denoted as:

$$s_t^k = \left\{k_i, h_{k_i}, z_{k_i}, \mathcal{C}_{k_i}^{iac}(t), \mathcal{C}_{k_i}^{que}(t), \mathcal{C}_{k_i}^{inf}(t), \mathcal{C}_{k_i}^{tsk}(t), a_{l_j}\right\}, \tag{10}$$

where k_i represent the ID of request and $a_{l_j}, 1 \leq j \leq |\mathbf{L}|$ is the assigned LLM deployment.

To sum up, the total state at time t can be denoted as :

$$s_t = \left\{s_t^l \cup s_t^k \cup s_t^n\right\} \in \mathcal{S}. \tag{11}$$

Based on the dynamic state, a feature extraction and concatenation network is used to capture the intricate relationships in request scheduling for subsequent decision-making processes.

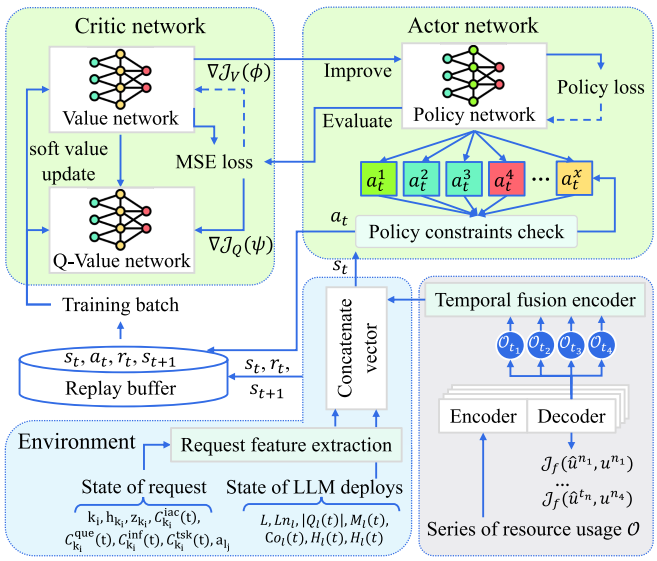


Fig. 4. The overview of WPDS algorithm.

Action space ($\mathcal{A}$): The action space $\mathcal{A}$ is defined by the set $\mathbf{L}$, where each action $a_t \in \mathcal{A}$ corresponds to selecting an LLM deployment from set $\mathbf{L}$ at time t. Formally, an action a_t is represented as a vector in a discrete space $\mathbb{R}^{|\mathbf{L}|}$, where each element corresponds to an LLM deployment $l \in \mathbf{L}$ to which a request k_i is assigned for scheduling.

Observation space ($\mathcal{O}$): The observed data of resource utilizing states for four different time intervals is denoted as $\mathcal{O} \in \mathbb{R}^{|\mathbf{L}| \times N \times 4}$, where $|\mathbf{L}|$ is the number of edge-based LLM deployments which capturing the dynamic temporal utilization. N is the length of each sequence. The dimension 4 corresponds to the time intervals of 30 seconds, 1 minute, 5 minutes, and 30 minutes. The 30-second interval aims to capture the characteristics of short-term LLM deployment initialization costs or restarts, while the one and five-minute intervals aim to reflect the average load characteristics of the time taken to process incoming requests. The 30-minute interval represents the overall load characteristics of the system from a longer perspective.

Reward function (r): The goal of the agent is to maximize the reward function $r_t \sim r(s_t, a_t)$, where the reward depends on the current state s_t and the action taken a_t. The reward can be expressed as $r_t = -\mathcal{C}(t)$, indicating that it is derived from the negative value of the cost $\mathcal{C}(t)$ at time t.

B. Workload Predictive Encoding

To capture the dynamic relationship between resource utilization and future trends in edge-based LLM deployments, we extract latent predictive features from multi-dimensional time-series data and combine them with request priority scores that assess the importance and urgency of each request. This approach optimizes scheduling by improving resource efficiency and timely request processing.

To extract informative time-series resource features from partial observations and predict future resource utilization, we design a temporal fusion encoder network, illustrated in Fig. 4. This network employs a Transformer-based Encoder [42] to effectively process partial data and derive meaningful representations. Using self-supervised learning, it captures latent representations across four time dimensions for integration into the state space.

1) Multi-Dimensional Temporal Workload Merging: For multi-dimensional time-series features, feed-forward networks (FFNs) is applied to update the Transformer-based Encoder blocks for each temporal input. This allows the network that receives four-dimensional sequences $\mathcal{O} = \{\mathbf{O}_{30\,\mathrm{s}}, \mathbf{O}_{1\,\mathrm{m}}, \mathbf{O}_{5\,\mathrm{m}}, \mathbf{O}_{30\,\mathrm{m}}\}$, for each in $\mathbb{R}^{N \times |\mathbf{L}|}$, to capture the raw features of different temporal patterns. These partial observations are fed into multi-headed self-attention network to learn feature representations. For each temporal dimension input $\mathbf{O}_n$, where $n \in \{30\,\mathrm{s}, 1\,\mathrm{m}, 5\,\mathrm{m}, 30\,\mathrm{m}\}$, a set of triplet tuples consisting of queries $\mathbf{Q}_n^{(h)}$, keys $\mathbf{K}_n^{(h)}$, and values $\mathbf{V}_n^{(h)}$ that obtained through linear transformations. The self-attention value is computed as follows:

$$AT_n^{(h)} = \mathbf{O}_n^{\mathcal{L}-1} + \mathrm{softmax}\left(\frac{\mathbf{Q}_n^{(h)}(\mathbf{K}_n^{(h)})^\top}{\sqrt{d_o}}\right)\mathbf{V}_n^{(h)}. \quad (12)$$

The FFN block then independently processes the attention outputs $AT_n^{(h)}$ at each time step. This block consists of two fully connected layers with the ReLU activation function:

$$F_n^{(h)} = \mathrm{ReLU}\left(AT_n^{(h)}\mathbf{W}_1^{(h)} + b_1^{(h)}\right)\mathbf{W}_2^{(h)} + b_2^{(h)}, \quad (13)$$

where $\mathbf{W}_1^{(h)} \in \mathbb{R}^{|\mathbf{L}| \times d_f}$ and $\mathbf{W}_2^{(h)} \in \mathbb{R}^{d_f \times |\mathbf{L}|}$ are weight matrics. $b_1^{(h)}$ and $b_2^{(h)}$ are biases, and d_f is the dimension of the intermediate FFNs layer. Then, the hidden representations from different time dimensions are combined to form the final predictive state representation, which is denoted as:

$$MAT\left(\mathcal{O}_n^{\mathcal{L}-1}\right) = \left[F_{30s}^{(1)}; F_{30s}^{(2)}; \ldots; F_{30s}^{(h)}, F_{1m}^{(1)}; \ldots; F_{30m}^{(h)}\right]\mathbf{W}_o, \quad (14)$$

where $\mathbf{W}_o \in \mathbb{R}^{hd_o \times d_{out}}$, and d_{out} is the output dimension. The MAT layer aggregates the hidden features across multiple time scales.

2) Predictive Encoder Representations: We utilize inputs from four-dimensional temporal series to obtain the current representations of the predictive encoder layer, denoted as:

$$\hat{\mathbf{U}}_n = \mathrm{ReLU}\left(\mathbf{F}_n\mathbf{W}_n^p + b_n^p\right), \quad (15)$$

where $\mathbf{F}_n = [F_n^{(1)}, \ldots, F_n^{(h)}]\mathbf{W}_f$ represents the concatenation of all attention heads followed by a linear transformation layer for integration. $\mathbf{W}_n^p \in \mathbb{R}^{d_{out} \times |\mathbf{L}|}$ is the n-th temporal dimension weight matrix, and b_n^p is the bias term. This allows us to capture specific features at different time scales. The loss function between the predicted values $\hat{\mathbf{u}}_{l,n}$ and the real values $\mathbf{u}_{l,n}$ for each time dimension n is defined as:

$$\mathcal{J}_{f,n} = \mathcal{J}_n\left(\hat{\mathbf{u}}_{l,n}, \mathbf{u}_{l,n}\right) = \left\|\hat{\mathbf{u}}_{l,n} - \mathbf{u}_{l,n}\right\|^2 + \beta\sum_{i}^{|\mathbf{L}|}\sum_{j}^{|\mathbf{L}|}\max\left(0, \left(\hat{u}_{l_i,n} - \hat{u}_{l_j,n}\right)\left(u_{l_i,n} - u_{l_j,n}\right)\right), \quad (16)$$

where β is a hyperparameter to balance the weight of regression loss and the LLM deployment ranking loss. The aim for the predicted resource utilization rates is to accurately reflect the relative utilization levels among different LLM deployments.

Algorithm 1: The WPDS Algorithm.

Input : Request queue $\mathcal{Q}_{1,2,\cdots,|\mathbf{L}|} = \emptyset$ associate with $|\mathbf{L}|$ threads, π_θ
Output: a_t

Initialize running time t_0
Get the first request k_0 from $\mathbf{K}$
Get the fiiltered valid set of edge server $\mathbf{E}_i(t_0)$
for $k_i \in \{0, 1, 2, \ldots, |\mathbf{K}|\}$ **do**
  Get the state features $s_t^{k_i}$ of k_i by Equation 10
  Get the running time t_i of the current k_i
  Calculate the tolerance value tol_i: $(\mathcal{C}_{k_i}^{inf}(t) + \mathcal{C}_{k_i}^{que}(t)) - z_{k_i}$ of k_i
  Selected the target LLM $l_i \in \mathbf{L}$, based on $\pi_\theta(t)$ and $\mathbf{E}_i(t)$
  Put (-tol_i, t_i, k_i) to $\mathcal{Q}_i$
end for
/* Parallel Task Processing */
while $\mathbf{K}$ *is not empty* **do**
  for $\mathcal{Q}_{1,2,\cdots,|\mathbf{L}|}$ *is not empty* **do in parallel**
    Check the state of LLM pod l_j
    if $restart\text{-}flag$ **then**
      Get the current time t_j
      Update the running time t_j of the current k_j
      Put k_j to $\mathcal{Q}$
      Restart the current LLM pod l_j
      Update the environment
    end if
    Fetch the current processing request $k_{j,i}$ from $\mathcal{Q}_j, 1 \le j \le |\mathbf{L}|$
    Execute the request $k_{j,i}$
    Calculate the completion time $C_{k_{j,i}}^{tsk}$
  end forpar
  Return a_t
end while
end

3) Policy Training: The WPDS algorithm is based on soft actor-critic policy optimization of the reinforcement learning (RL) agent [23]. This approach enables the agent to adaptively learn scheduling policies to maximize long-term rewards, such as resource utilization and request features. To achieve this, the RL agent integrates multi-dimensional encoder representations with the state of incoming requests and LLM deployments into a unified state space. The integrated latent state, denoted as $s_t = \text{Concat}(s_t^l, s_t^k, s_t^n)$, where $s_t^n = MAT(\mathcal{O}_n^{\mathcal{L}-1})$, is fed into actor and critic networks to facilitate the decision-making process. The critic network learns the soft Q-function $Q_\psi(s, a)$ by minimizing the soft Bellman residual, enabling gradient propagation and joint training of predictive coding and policy learning. The critic loss function is obtained as:

$$\min_\psi \mathbb{E}_{(s_t, a_t, r_t, s_t') \sim \mathcal{D}} \left[\frac{1}{2} \left(Q_\psi(s_t, a_t) - \widehat{Q}(s_t, a_t) \right)^2 \right], \quad (17)$$

where the calculation of expectation based on a mini-batch that samples from the replay buffer $\mathcal{D}$, and the target value $\widehat{Q}(s_t, a_t)$

Algorithm 2: Training of WPDS Algorithm.

Input : State s_t
Output: a_t

Initialize trajectories ν_0,
for *epoch* = $0, 1, 2, \ldots$ **do**
  Get the state features of k_i and l_i
  while $\mathcal{U} \neq \emptyset$ **do**
    Run policy π_θ
    Store a set of trajectories ν in $\mathcal{D}$
  end while
  if $|\mathcal{D}| \bmod batch\_size = 0$ **then**
    Sample a set of trajectories ν
    Compute target soft value function $\widehat{Q}(s_t, a_t)$
    Compute loss function $\mathcal{J}(\theta)$
    Optimize network and update parameters
  end if
end for
Return a_t
end

is defined as:

$$\widehat{Q}(s_t, a_t) = r_t + \gamma \mathbb{E}_{a \sim \pi_\theta} \left(Q_{\psi'}(s_{t+1}, a_{t+1}) - \epsilon \log \pi_\theta(a_{t+1} \mid s_{t+1}) \right), \quad (18)$$

where θ and ψ represent the parameters of the actor network and critic network, respectively. $\epsilon > 0$ is the temperature parameter that balances reward maximization and policy entropy. For updating the policy network with parameter θ, the loss function is defined as:

$$\min_\theta \mathbb{E}_{s \sim \mathcal{D}, a \sim \pi_\theta} \left[Q_\psi(s_t, a_t) - \epsilon \log \pi_\theta(a_t \mid s_t) \right]. \quad (19)$$

The WPDS training process optimizes the policy π_θ for selecting the best edge server for each arriving request, as illustrated in Fig. 4, with further details provided in Algorithm 2. It begins by initializing trajectories ν_0. During each epoch, it first collects the state features of both the requests $k_i \in \mathbf{K}$ and the LLM $l_i \in \mathbf{L}$. With the current policy running, the resulting trajectories are stored in the replay buffer $\mathcal{D}$. When stored trajectories reach a multiple of the batch size, a batch of trajectories is sampled and used to compute the loss and update the parameters θ. This process iteratively improves the policy until it converges to an optimal policy.

C. Request Rescheduling and Inference Recovery

In the WPDS systems, maintaining reliable inference services is critical for ensuring consistent user experiences and system efficiency. The Request Rescheduling and Inference Recovery (RRIR) mechanism, as shown in Algorithm 3, plays a pivotal role in managing unresponsive or overloaded deployments. The mechanism iterates through all LLM deployments, assessing their health status at the current timestamp. If a failure is detected, the affected deployment undergoes recovery, which includes rescaling its capacity. This ensures that the deployment can resume serving requests as soon as possible. Meanwhile, the mechanism retrieves the request queue for the failed deployment, marks the affected requests as failed, recalculates their queuing and running times, and updates their status. The affected requests

Algorithm 3: The RRIR Mechanism.

Input : LLM deployments $l_{1,2,\cdots,|\mathbf{L}|}$
Output: Updated request set $\mathbf{K}$

Get the current time t_c
for $l_i \in \{l_0, l_1, l_2, \ldots, l_{|\mathbf{L}|}\}$ **do**
  Get the status of LLM deployment $H_{l_i}(t_c)$
  if $H_{l_i}(t_c)$ *is failed* **then**
    /* Rescale Failed LLM Deployment */
    Update the restart times $B_{l_i}(t_c)$
    Rescale the LLM deployment $q^r_{l_i}$
    /* Parallel Request Rescheduling */
    Get the request queue $\mathcal{Q}_i$
    Get the current request k_j from $\mathcal{Q}_i$
    Mark the process status k_j as failed
    Update the running time $t_c - t_j$ of k_j
    Update the queuing time $\mathcal{C}^{que}_{k_i}(t_c) + t_c - n_t^{k_j}$
    Put k_j to $\mathbf{K}$
  end if
end for
Return $\mathbf{K}$
end

are promptly evicted and rescheduled with updated queue priorities and arrival times, then processed by Algorithm 1. This process not only ensures fairness by recalibrating computing times but also minimizes the impact of failures on the overall system throughput.

D. Computational Complexity Analysis

The analysis of computational complexity primarily involves three operations. As shown in Algorithm 1, the request queue $\mathcal{Q}_{1,2,\ldots,|\mathbf{L}|}$ is initialized with $|\mathbf{K}|$ requests. The complexity of the state for requests obtained by (10) is $O(|\mathbf{L}||\mathbf{K}|)$. The policy π_θ is queried for action selection, which has a constant complexity O_t. Adding requests to queue $\mathcal{Q}_i$ has complexity $O(\log(|\mathcal{Q}_i|))$ for each insertion. Therefore, the sequential processing for $|\mathbf{K}|$ requests results in total complexity $O(|\mathbf{K}|(|\mathbf{L}| + O_t + \log(|\mathbf{K}|)))$. In parallel processing, each thread processes requests from its priority queue $\mathcal{Q}_j$. Fetching and executing a request, as well as updating the pod state, are $O(1)$ operations. The parallel loop runs for $|\mathbf{K}|$ iterations, resulting in total complexity $O(|\mathbf{K}|)$ for parallel execution. Hence, the WPDS algorithm operates in polynomial time.

For Algorithm 2, the training process involves storing trajectories and updating the policy network. Assume the number of epochs is E, the batch size is B, and the number of requests is $|\mathbf{K}|$. Generating trajectories by running the policy π_θ over requests in $\mathbf{K}$ has complexity $O(E|\mathbf{K}|O_t)$, where O_t is the policy evaluation complexity. For each batch of size B, computing the advantage function has a complexity of $O(B)$. The policy network update depends on the number of weights and dimensions of the network. Using FLOPs as the metric, the complexity of a fully connected layer is $O(D_{in}D_{out})$ for each layer. The total complexity for updating the policy and value networks in each batch is $O(BD_{in}D_{out})$. Thus, the total training complexity of WPDS algorithm over E epochs and $|\mathbf{K}|$ requests is $O(E|\mathbf{K}|O_t + E(B + BD_{in}D_{out}))$.

As shown in Algorithm 3, the RRIR mechanism focuses on monitoring and rescaling failed LLM deployments. Assume the number of deployments is $|\mathbf{L}|$ and the number of requests is $|\mathbf{K}|$. Checking the status of all deployments $\mathbf{H}_{l_i}(t)$ has a complexity of $O(|\mathbf{L}|)$. For each failed deployment, requests are fetched from the queue $\mathcal{Q}_i$ and updated with a complexity of $O(1)$. Assuming all requests are rescheduled, this contributes $O(|\mathbf{K}|)$ to the complexity. The RRIR mechanism has a total complexity of $O(|\mathbf{L}| + |\mathbf{K}|)$ per iteration, which is linear in the number of deployments and requests.

VI. Performance Evaluation

A. Experimental Settings

We developed a distributed edge-based LLM inference system using the Kubernetes scheduling framework [43], deploying the WPDS algorithm on the control plane node. The *API Server* receives user requests and assigns them to edge servers selected by WPDS scheduling strategies through *Kubelet*. Each edge server processes its assigned requests and maintains network communication through *Kube-proxy*. If an LLM deployment on an edge server fails, the *Controller Manager* notifies the *Scheduler* with an unprocessed request ID, prompting rescheduling through a new scheduling decision. The system persistently stores the state of nodes, LLM deployments, resource usage, and scheduling data in etcd, and monitors runtime resource and performance metrics via the *Prometheus API*, providing real-time data to optimize WPDS scheduling.

We train the WPDS algorithm using between 500 and 10,000 requests, implemented in Python. The cost scaling constants α and ε are set to 2 and 0.01, respectivelty. The value α amplifies queueing cost in the reward function to prioritize reducing request waiting time. The parameter ε scales the GPU utilization metric obtained from Prometheus as a percentage to align its magnitude with other time-based cost terms, ensuring balanced gradient contributions during training. Learning rates for policy and value networks are 1×10^{-4}, the discount factor γ is 0.9, and the soft update value is 0.1. Our model includes four multi-head attention layers with a dimension of 128. Both the actor and critic networks have two fully connected layers: the first with 64 units and the second with 256 units. We update the network parameters at the end of each epoch, using a batch size of 256. We compare the performance of our WPDS algorithm against several baselines.

- *RR [18]:* The Round Robin (RR) algorithm is a default load-balancing algorithm on the Kubernetes platform that sequentially assigns each incoming request to LLM services equally.
- *RA [11]:* The Random Algorithm (RA) is a load-balancing method that randomly distributes incoming requests to available LLM service instances.
- *DRL [44]:* The Deep Reinforcement Learning (DRL) baseline uses a traditional actor-critic algorithm without incorporating resource utilization predictions.
- *SIA [45]:* The Sequential Inference Algorithm (SIA) processes requests on a single instance of the SLM deployment without distributed execution and model RRIR mechanism. It represents a naive, non-scalable baseline for request handling.

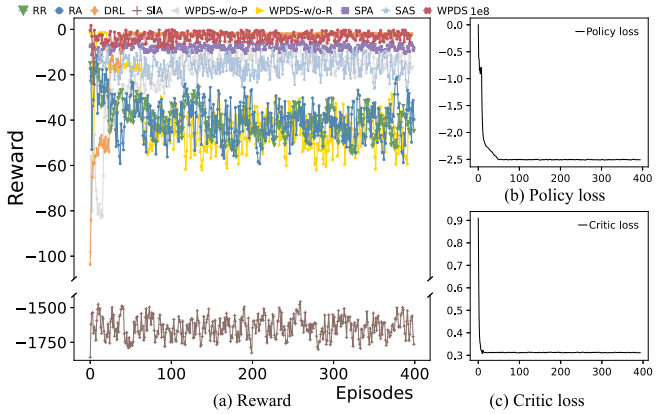


Fig. 5. Training performance of rewards and losses.

- *SPA [20]:* The Split Phase Algorithm (SPA) segments long requests into smaller chunks processed sequentially while distributing short requests in parallel across multiple LLM services.
- *SAS:* The Static-Aware Scheduling (SAS) adopts a simplified scheduling policy from [19], where requests are assigned to nodes with lower resource utilization solely based on their priority.

B. Experimental Results

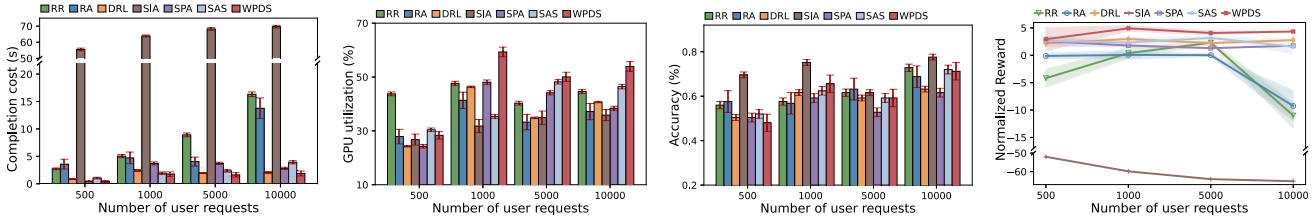
Performance of the WPDS algorithm: To evaluate the effectiveness of the WPDS algorithm, we conduct training experiments comparing it to several baseline algorithms (Fig. 5). The average training performance ranks as WPDS > DRL > SPA > SAS > RR > RA > SIA shown in Fig. 5(a), with WPDS achieving the highest and most stable rewards after convergence. The WPDS, DRL, and SIA algorithms undergo a training phase consisting of 400 episodes. Initially, the WPDS algorithm goes through an exploration phase where actions are selected randomly, leading to fluctuating rewards. As a result, during this initial stage, the rewards for the remaining algorithms except the SIA algorithm are relatively similar. As training progresses, the performance of RL-based algorithms like WPDS, DRL, and SPA algorithms significantly improves. Their rewards increase steadily and eventually converge, with WPDS achieving higher and more stable rewards than the other algorithms. This indicates that WPDS effectively learns and optimizes the scheduling policies that balance workloads across edge devices to maximize rewards. In contrast, the SAS algorithm relies solely on local queue priorities and real-time resource utilization, without exploiting global and historical system states, which limits its scheduling performance. RR and RA algorithms follow predefined rules without learning from request characteristics and the environment. This lack of adaptability means they cannot optimize for varying request patterns or resource utilization, leading to consistently lower rewards over time. The SIA algorithm, due to its inability to support distributed inference, achieves the lowest system rewards. This reflects the lower bound of the overall inference system's performance. Despite their simplicity, these heuristic algorithms serve as important baselines to highlight the superiority of WPDS in terms of adaptability and performance.

Fig. 5(b) and (c) further support the convergence of the WPDS algorithm. In Fig. 5(b), the policy loss rapidly decreases within the first 100 episodes. This reduction indicates that the algorithm efficiently learns to optimize its policy parameters. Similarly, Fig. 5(c) shows the convergence of the critic loss. As training progresses, the critic loss steadily decreases, indicating that WPDS effectively refines its ability to assess the value of different actions. Overall, the rapid and stable reduction in both losses confirms that WPDS efficiently learns and optimizes its scheduling strategy, leading to superior performance compared to other algorithms.

Ablation study on key components: To quantify the contribution of each core component, ablation experiments of WPDS-w/o-P without the predictive network and WPDS-w/o-R without the recovery network are conducted, as shown in Fig. 5. Specifically, WPDS-w/o-P achieves only performance comparable to heuristic methods like SPA and SAS, indicating that the predictive encoder representations are essential for effective state observation. Without them, the agent can only rely on raw resource metrics and request features, limiting its ability to reflect workload dynamics. Moreover, WPDS-w/o-R exhibits a significantly lower average reward after convergence, demonstrating that the recovery mechanism is critical for handling LLM service failures. The WPDS model, by integrating both components, achieves stable training and superior long-term scheduling performance.

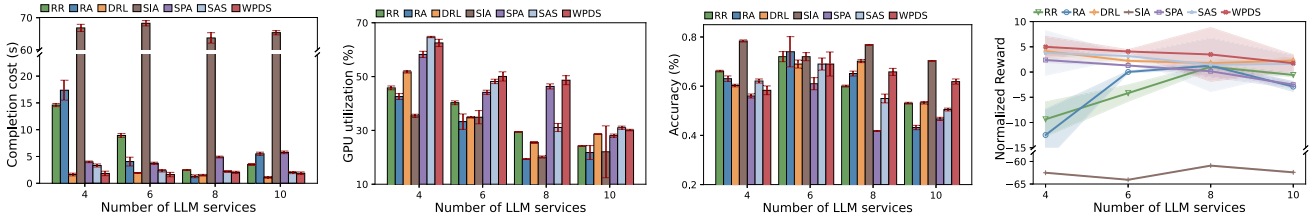
Performance of different numbers of user requests: Fig. 6 show the performance of each algorithm across different numbers of user requests. Fig. 6(a) illustrates the completion delays associated with request processing across different algorithms. The WPDS algorithm maintains low completion delays as request volume increases, significantly outperforming RR, RA, and SIA algorithms. These heuristic algorithms fail to achieve fine-grained, request-aware scheduling and effective load balancing across edge-based LLM services, resulting in higher average completion delays. In contrast, RL-based algorithms dynamically adapt to the edge inference environment and consistently achieve lower request completion delays. Notably, the SPA algorithm performs best under low request concurrency due to its hybrid strategy that segments long requests and parallelizes short ones. However, its adaptability degrades as request volume increases. In contrast, WPDS jointly considers dynamic edge LLM load and request characteristics, enabling assignment to the most suitable LLM services for faster inference. As a result, WPDS maintains relatively stable and low inference delays under higher loads and outperforms SPA in more congested scenarios. Compared to DRL, SPA, and SAS algorithms, WPDS reduces completion delay by 20.3%, 46.3%, and 37.9%, respectively, demonstrating its effectiveness in handling increasing request volumes while maintaining lower inference overhead.

As shown in Fig. 6(b), this figure shows GPU utilization on edge devices under different algorithms as request volume increases. High utilization indicates efficient LLM inference, which is critical given the heavy computational demands of LLMs. Methods such as RR, RA, SAS, and SIA ignore real-time and historical GPU usage and rely on fixed rules, resulting in suboptimal utilization. In contrast, RL-based algorithms (WPDS, DRL, and SPA) adapt to system dynamics and achieve higher GPU utilization. Among them, WPDS performs best by more effectively balancing workloads and allocating GPU resource usage. This high utilization is a result of WPDS's ability to intelligently allocate resources and balance the workload across available GPUs.



(a) Completion costs. (b) GPU utilizations. (c) Accuracy scores. (d) Rewards.

Fig. 6. Performance metrics for different numbers of user requests.



(a) Completion costs. (b) GPU utilizations. (c) Accuracy scores. (d) Rewards.

Fig. 7. Performance metrics for different numbers of LLM services.

Despite variations in request scheduling and inference models, Fig. 6(c) and (d) demonstrate that WPDS maintains accuracy consistently close to top-performing baselines, while enabling significant gains in inference efficiency. This minimal accuracy trade-off underscores WPDS's practical balance between precision and system performance in real-world deployments. Furthermore, Fig. 6(d) highlights the stability of WPDS's performance as the number of requests increases. Baseline algorithms such as RR, RA, and SIA exhibit a notable decline in system reward under higher loads. This degradation primarily stems from their inability to adapt to dynamic resource availability and heterogeneous request characteristics. These methods often lead to node overloading or inefficient request allocation. In contrast, WPDS leverages workload prediction and an RL-based framework to proactively balance edge resources, consistently achieving high system rewards even under heavy workloads. As a result, WPDS consistently maintains high system reward and robust performance even under increasing workloads, demonstrating its effectiveness in managing complex and dynamic edge inference environments.

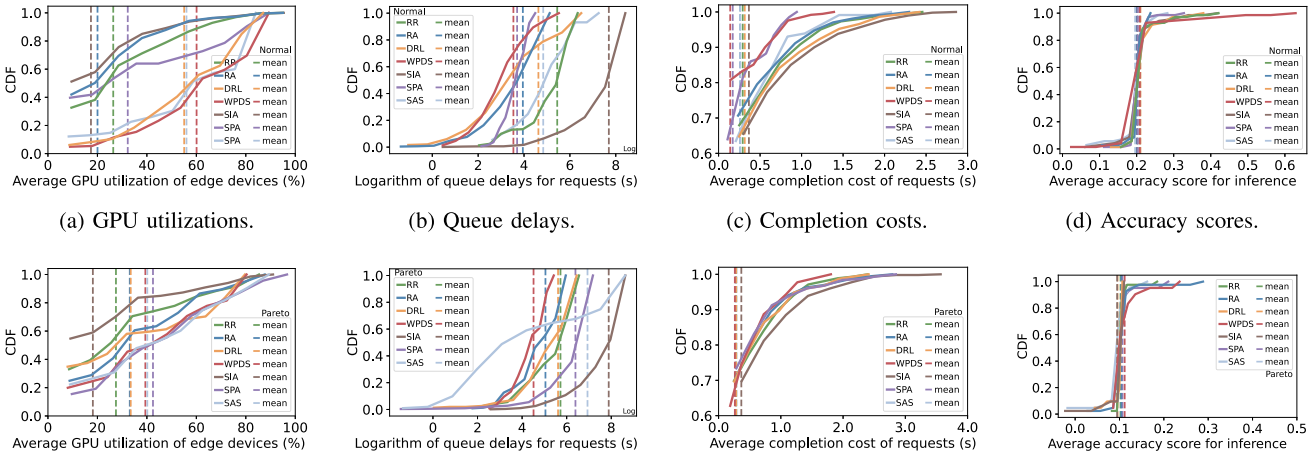
Performance of different numbers of LLM services: Fig. 7 provide a performance comparison across different numbers of LLM services. As the number of LLM services increases, the complexity of managing these services and their associated requests increases significantly. Fig. 7(a) shows the completion delays associated with varying numbers of LLM services. The RL-based algorithm achieves substantially lower completion delays compared to heuristic algorithms as LLM service count grows. The WPDS algorithm consistently maintains acceptable completion delays in a dynamically expanding edge environment, demonstrating strong scalability and robustness in environments with fluctuating demand and system capacity. The ability to scale effectively while maintaining performance is crucial for ensuring reliable and efficient operation.

Fig. 7(b) and (c) indicate that the WPDS algorithm achieves high GPU utilization while preserving stable accuracy scores, showcasing its capability to manage workloads across LLM services effectively. High GPU utilization signifies that WPDS efficiently allocates user requests, ensuring that each LLM service operates at optimal performance levels. This effective workload distribution prevents resource idling and maximizes the utilization of computational power. Moreover, despite increased system complexity from scaling LLM services, WPDS maintains stable accuracy scores, highlighting its ability to balance scheduling efficiency and service quality under varying demands. Overall, WPDS demonstrates robustness in adapting to varying demands while optimizing performance across the system.

Fig. 7(d) presents the overall reward rankings for different algorithms across various metrics. The results show that the WPDS algorithm outperforms all baseline algorithms, achieving the highest rewards. Specifically, the reward rankings are: WPDS > DRL > SAS > SPA > RR > RA > SIA.These findings demonstrate WPDS's superior adaptability and performance, effectively balancing multiple objectives.

Performance of different request distributions: Fig. 8 illustrates an overview of the GPU utilization, queue delays, completion costs, and accuracy scores under different request distributions. Specifically, Fig. 8(a) and (e) focus on GPU utilization performance of different algorithms. The WPDS algorithm consistently maintains high GPU utilization across workloads, demonstrating its efficiency in scheduling user requests under Normal or Pareto distributions. By dynamically adjusting to varying request patterns, WPDS ensures that GPU resources are fully leveraged. In contrast, other algorithms exhibit lower resource utilization, especially under the Pareto distribution, where the variability in request-arriving frequencies poses a greater challenge. WPDS's ability to adapt and optimize resource utilization under diverse conditions underscores its efficiency in GPU usage.

Fig. 8(b) and (f) indicate the queue delays experienced by different algorithms. The WPDS algorithm stands out for its



(a) GPU utilizations. (b) Queue delays. (c) Completion costs. (d) Accuracy scores.

(e) GPU utilizations. (f) Queue delays. (g) Completion costs. (h) Accuracy scores.

Fig. 8. Performance metrics for different workload distributions.

TABLE II
COMPUTATION RESOURCES FOR EACH DECISION-MAKING

Algorithm	RAM	VRAM	Execution Time (s)
RR	-	-	6.48×10^{-4}
RA	-	-	7.03×10^{-4}
DRL	236Kb	242Kb	3.68×10^{-3}
SIA	-	-	7.25×10^{-4}
SPA	236Kb	244Kb	1.17×10^{-3}
SAS	-	-	1.23×10^{-3}
WPDS	236Kb	607Kb	2.40×10^{-2}

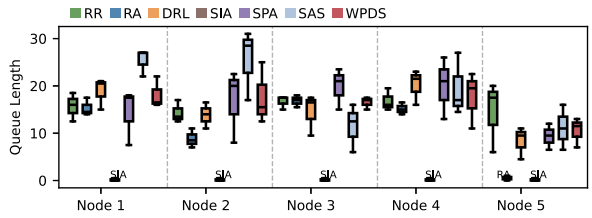


Fig. 9. Queue length distribution across edge servers.

effectiveness in reducing queue delays. Compared to DRL, SPA, and RA, WPDS achieves reductions of 2%, 16%, and 11%, respectively. Shorter delays enable faster request processing and improved system performance. WPDS achieves this by intelligently distributing tasks across LLM services, ensuring that requests are promptly processed without overloading any single service.

Fig. 8(c) and (g) show completion costs across different algorithms. While DRL, SPA, and SAS achieve lower completion costs compared to heuristic algorithms, they still fall short of WPDS's performance. WPDS consistently demonstrates the lowest completion costs, reflecting its adaptive scheduling and balanced workload distribution. Finally, Fig. 8(d) and (h) examine accuracy scores of different algorithms. Despite variations in request patterns, all algorithms largely maintain similar accuracy scores. However, WPDS's consistently better performance across multiple metrics translates into a more reliable and efficient service experience.

Computational complexity and execution resources: As shown in Table II, we use *torch.profiler* [46] to measure Random Access Memory (RAM), Video RAM (VRAM), and execution time for each algorithm during decision-making. RR, RA, and SIA algorithms achieve the shortest execution times due to their simple rule-based policies without modeling real-time system states. Compared to other deep learning-based methods, WPDS incurs only a moderate increase in execution time (6.5x) while enabling significantly better scheduling decisions. This overhead is justified by the significant performance gains, which outweigh the millisecond-level latency. Moreover, WPDS consumes negligible computational resources on edge servers and can be further accelerated via model quantization or inference optimization.

Load distribution performance: To provide a more comprehensive understanding of load distribution across edge servers, queue length distributions across five edge nodes at a fixed point in time under different scheduling algorithms are presented in Fig. 9. The RR algorithm serves as a baseline for fair load distribution, indicating its ability to achieve relatively balanced task distribution. SIA, as a single-instance algorithm, assigns most tasks to one node, causing extreme imbalance despite low absolute queue lengths. In contrast, algorithms such as RA, SPA, and SAS show higher variability in queue lengths, suggesting uneven task allocation and potential bottlenecks. The WPDS algorithm achieves low median queue lengths across all nodes, demonstrating load-balancing capability. This indicates that WPDS not only reduces overall system latency but also effectively prevents any single node from becoming overloaded.

VII. CONCLUSION

In this paper, we propose WPDS, a workload prediction and dynamic request scheduling algorithm that improves LLM inference on edge devices under large-scale user requests. By leveraging a Transformer-based predictive network, WPDS captures temporal patterns and high-volume request characteristics typical of edge environments. It then employs a soft actor-critic reinforcement learning-based framework to learn an adaptive scheduling policy that reduces inference latency and balances

workload across edge nodes. We evaluated WPDS's performance on a real-world edge system, and experiments show that it outperforms baseline algorithms by up to 16.6% under the tested conditions. However, our current implementation assumes homogeneous LLM models and a moderate-scale edge cluster. Scalability to heterogeneous LLM systems with thousands of nodes remains an open challenge. Future work will explore stateful request scheduling and caching strategies to further enhance inference efficiency, with a focus on extending WPDS to more heterogeneous and large-scale edge deployments.

REFERENCES

[1] J. Wei et al., "Emergent abilities of large language models," *Trans. Mach. Learn. Res.*, 2022. [Online]. Available: https://openreview.net/forum?id=yzkSU5zdwD

[2] Y. Shen et al., "Large language models empowered autonomous edge AI for connected intelligence," *IEEE Commun. Mag.*, vol. 62, no. 10, pp. 140–146, Sep. 2024.

[3] Y. Wang, C. Yang, S. Lan, L. Zhu, and Y. Zhang, "End-edge-cloud collaborative computing for deep learning: A comprehensive survey," *IEEE Commun. Surv. Tut.*, vol. 26, no. 4, pp. 2647–2683, Fourth Quarter, 2024.

[4] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han, "SmoothQuant: Accurate and efficient post-training quantization for large language models," in *Proc. Int. Conf. Mach. Learn.*, 2023, pp. 38087–38099.

[5] Y. Sheng et al., "FlexGen: High-throughput generative inference of large language models with a single GPU," in *Proc. Int. Conf. Mach. Learn.*, 2023, pp. 31094–31116.

[6] Y. Liu et al., "CacheGen: KV cache compression and streaming for fast large language model serving," in *Proc. ACM SIGCOMM Conf.*, 2024, pp. 38–56.

[7] R. Ma et al., "HPipe: Large language model pipeline parallelism for long context on heterogeneous cost-effective devices," in *Proc. 2024 Conf. North Amer. Chapter Assoc. Comput. Linguistics: Hum. Lang. Technol.*, 2024, pp. 1–9.

[8] B. Sun et al., "Llumnix: Dynamic scheduling for large language model serving," in *Proc. 18th USENIX Symp. Operating Syst. Des. Implementation*, Santa Clara, CA: USENIX Association, 2024, pp. 173–191.

[9] T. Dreibholz and S. Mazumdar, "Towards a lightweight task scheduling framework for cloud and edge platform," *Internet Things*, vol. 21, 2023, Art. no. 100651.

[10] J. Dogani, R. Namvar, and F. Khunjush, "Auto-scaling techniques in container-based cloud and edge/fog computing: Taxonomy and survey," *Comput. Commun.*, vol. 209, pp. 120–150, 2023.

[11] Z. Tang, J. Lou, and W. Jia, "Layer dependency-aware learning scheduling algorithms for containers in mobile edge computing," *IEEE Trans. Mobile Comput.*, vol. 22, no. 6, pp. 3444–3459, Jun. 2023.

[12] J. Chen et al., "Edgeinfinite: A memory-efficient infinite-context transformer for edge devices," 2025, *arXiv:2503.22196*.

[13] D. Zhang, N. Vance, Y. Zhang, M. T. Rashid, and D. Wang, "EdgeBatch: Towards AI-empowered optimal task batching in intelligent edge systems," in *Proc. IEEE Real-Time Syst. Symp.*, 2019, pp. 366–379.

[14] A. Borzunov et al., "Distributed inference and fine-tuning of large language models over the Internet," in *Proc. Adv. Neural Inf. Process. Syst.*, 2024, pp. 12312–12331.

[15] Z. Yu et al., "EDGE-LLM: Enabling efficient large language model adaptation on edge devices via unified compression and adaptive layer voting," in *Proc. 61st ACM/IEEE Des. Automat. Conf.*, 2024, pp. 1–6.

[16] W. Kwon et al., "Efficient memory management for large language model serving with paged Attention," in *Proc. 29th Symp. Operating Syst. Princ.*, 2023, pp. 611–626.

[17] E. J. Ghomi, A. M. Rahmani, and N. N. Qader, "Load-balancing algorithms in cloud computing: A survey," *J. Netw. Comput. Appl.*, vol. 88, pp. 50–71, 2017.

[18] J. Chen, D. Wang, and W. Zhao, "A task scheduling algorithm for hadoop platform," *J. Comput.*, vol. 8, no. 4, pp. 929–936, 2013.

[19] X. Miao et al., "Spotserve: Serving generative large language models on preemptible instances," in *Proc. 29th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2024, pp. 1112–1127.

[20] P. Patel et al., "Splitwise: Efficient generative LLM inference using phase splitting," in *Proc. ACM/IEEE 51st Annu. Int. Symp. Comput. Archit.*, 2024, pp. 118–132.

[21] S. Liu et al., "Dependent task scheduling and offloading for minimizing deadline violation ratio in mobile edge computing networks," *IEEE J. Sel. Areas Commun.*, vol. 41, no. 2, pp. 538–554, Feb. 2023.

[22] S. Huang, Z. Wang, H. Zhang, X. Wang, C. Zhang, and W. Wang, "One for all: Unified workload prediction for dynamic multi-tenant edge cloud platforms," in *Proc. 29th ACM SIGKDD Conf. Knowl. Discov. Data Mining*, 2023, pp. 788–797.

[23] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 1861–1870.

[24] J. Turnbull, *Monitoring With Prometheus*. London. U.K.: Turnbull Press, 2018.

[25] J. Lin, S. Q. Zhang, and A. Leon-Garcia, "sLLM: Accelerating LLM inference using semantic load balancing with shared memory data structures," in *Proc. 25th Int. Symp. Qual. Electron. Des.*, 2024, pp. 1–6.

[26] D. Crankshaw et al., "InferLine: Latency-aware provisioning and scaling for prediction serving pipelines," in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 477–491.

[27] H. Fang, D. Zhang, C. Tan, P. Yu, Y. Wang, and W. Li, "Large language model enhanced autonomous agents for proactive fault-tolerant edge networks," in *Proc. IEEE Conf. Comput. Commun. Workshops*, 2024, pp. 1–2.

[28] S. Choi, S. Lee, Y. Kim, J. Park, Y. Kwon, and J. Huh, "Serving heterogeneous machine learning models on multi-GPU servers with spatio-temporal sharing," in *Proc. USENIX Annu. Tech. Conf.*, 2022, pp. 199–216.

[29] Hami, 2024. [Online]. Available: https://github.com/Project-HAMi/HAMi

[30] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing opencl kernels for high performance computing with FPGAs," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, pp. 409–420.

[31] S. Wang, X. Li, Q. Z. Sheng, R. Ruiz, J. Zhang, and A. Beheshti, "Multi-queue request scheduling for profit maximization in IaaS clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 11, pp. 2838–2851, Nov. 2021.

[32] T.-Y. Chang, J. Thomason, and R. Jia, "Do localization methods actually localize memorized data in LLMs? A tale of two benchmarks," in *Proc. 2024 Conf. North Amer. Chapter Assoc. Comput. Linguistics: Hum. Lang. Technol.*, 2024, pp. 3190–3211.

[33] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "OPTQ: Accurate quantization for generative pre-trained transformers," in *Proc. 11th Int. Conf. Learn. Representations*, 2023, pp. 1–16.

[34] B. Dolan and C. Brockett, "Automatically constructing a corpus of sentential paraphrases," in *Proc. 3rd Int. Workshop Paraphrasing*, 2005, pp. 1–8.

[35] A. Talmor, J. Herzig, N. Lourie, and J. Berant, "Commonsenseqa: A question answering challenge targeting commonsense knowledge," in *Proc. 2019 Conf. North Amer. Chapter Assoc. Comput. Linguistics: Hum. Lang. Technol.*, 2019, pp. 4149–4158.

[36] M. Sap, H. Rashkin, D. Chen, R. Le Bras, and Y. Choi, "SocialIQA: Commonsense reasoning about social interactions," in *Proc. 2019 Conf. Empirical Methods Natural Lang. Process. 9th Int. Joint Conf. Natural Lang. Process.*, 2019, pp. 4463–4473.

[37] Y. Bai et al., "LongBench: A bilingual, multitask benchmark for long context understanding," 2023, *arXiv:2308.14508*.

[38] X. Zhang et al., "∞ bench: Extending long context evaluation beyond 100k tokens," 2024, *arXiv:2402.13718*.

[39] C.-P. Hsieh et al., "RULER: What's the real context size of your long-context language models?," 2024, *arXiv:2404.06654*.

[40] C. An et al., "L-Eval: Instituting standardized evaluation for long context language models," in *Proc. 12th Int. Conf. Learn. Representations*, 2024, pp. 14388–14411.

[41] J. Li, M. Wang, Z. Zheng, and M. Zhang, "Loogle: Can long-context language models understand long contexts?," 2023, *arXiv:2311.04939*.

[42] L. Chen et al., "Decision transformer: Reinforcement learning via sequence modeling," in *Proc. Adv. Neural Inf. Process. Syst.*, 2021, pp. 15084–15097.

[43] "Kubernetes documentation," 2026. [Online]. Available: https://kubernetes.io/docs/home/

[44] A. Pradhan, S. K. Bisoy, and M. Sain, "Action-based load balancing technique in cloud network using actor-critic-swarm optimization," *Wireless Commun. Mobile Comput.*, vol. 2022, no. 1, 2022, Art. no. 6456242.

[45] G. Ottoni and D. August, "Global multi-threaded instruction scheduling," in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2007, pp. 56–68.

[46] "PyTorch documentation," (n.d.). [Online]. Available: https://pytorch.org/docs/

Fangyi Mou received the MSc degree from the Department of Computer Science, University of Macau, China, in 2020. She is currently working toward the PhD degree with Beijing Normal-Hong Kong Baptist University, and is a research assistant with the Advanced Institute of Natural Science, Beijing Normal University, China. Her research interests include edge computing, resource allocation, and container scheduling.

Zhiqing Tang (Member, IEEE) received the BS degree from the School of Communication and Information Engineering, University of Electronic Science and Technology of China, China, in 2015 and the PhD degree from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China, in 2022. He is currently an associate professor with the Advanced Institute of Natural Sciences, Beijing Normal University, China. His current research interests include edge computing, resource scheduling, and reinforcement learning.

Weijia Jia (Fellow, IEEE) is currently the director with the Institute of Artificial Intelligence and Future Networking, director with Super Intelligent Computer Center, Beijing Normal University at Zhuhai (BNU), and also the chair professor with UIC, China. From 2020 to 2024, he was the VP for research with UIC. Prior joining BNU, he was the deputy director with the State Key Laboratory of Internet of Things for Smart City, University of Macau and the Zhiyuan chair professor with Shanghai Jiaotong University, China. From 1995 to 2013, he was with the City University of Hong Kong, as a professor. His contributions have been recoganized for the research of edge AI, optimal network routing and deployment, vertex cover, anycast and multicast protocols, and sensors networking. He has more than 700 publications in international journals/conferences and research books and book chapters. He was the recipient of the 1st Prize of Scientific Research awards from the Ministry of Education of China in 2017 (list 2), and top 2% World Scientists in Stanford-list during 2020–2024, and many provincial science and Tech awards. He was the area editor of various prestige international journals, chair and PC member/keynote speaker for many top international conferences. He is also the distinguished member of CCF.

Wei Zhao (Fellow, IEEE) received the undergraduate degree in physics from Shaanxi Normal University, China, in 1977, and the MSc and PhD degrees in computer and information sciences from the University of Massachusetts at Amherst, in 1983 and 1986, respectively. He has served important leadership roles in academic including the chief research officer with the American University of Sharjah, the chair of academic council with CAS Shenzhen Institute of Advanced Technology, the eighth Rector of the University of Macau, the dean of science with Rensselaer Polytechnic Institute, the director for the Division of Computer and Network Systems in the U.S. National Science Foundation, and the senior associate vice president for research with Texas A&M University. He has made significant contributions to cyber-physical systems, distributed computing, real-time systems, and computer networks. He led the effort to define the research agenda of and to create the very first funding program for cyber-physical systems in 2006. His research results have been adopted in the standard of Survivable Adaptable Fiber Optic Embedded Network. He was awarded the Lifelong Achievement Award by the Chinese Association of Science and Technology in 2005.