Enhancing LLM QoS Through Cloud-Edge Collaboration: A Diffusion-Based Multi-Agent Reinforcement Learning Approach

Zhi Yao, Zhiqing Tang, *Member, IEEE*, Wenmian Yang, *Member, IEEE*, and Weijia Jia, *Fellow, IEEE*

***Abstract*—Large Language Models (LLMs) are widely used across various domains, but deploying them in cloud data centers often leads to significant response delays and high costs, undermining Quality of Service (QoS) at the network edge. Although caching LLM request results at the edge using vector databases can greatly reduce response times and costs for similar requests, this approach has been overlooked in prior research. To address this, we propose a novel Vector database-assisted cloud-Edge collaborative LLM QoS Optimization (VELO) framework that caches LLM request results at the edge using vector databases, thereby reducing response times for subsequent similar requests. Unlike methods that modify LLMs directly, VELO leaves the LLM's internal structure intact and is applicable to various LLMs. Building on VELO, we formulate the QoS optimization problem as a Markov Decision Process (MDP) and design an algorithm based on Multi-Agent Reinforcement Learning (MARL). Our algorithm employs a diffusion-based policy network to extract the LLM request features, determining whether to request the LLM in the cloud or retrieve results from the edge's vector database. Implemented in a real edge system, our experimental results demonstrate that VELO significantly enhances user satisfaction by simultaneously reducing delays and resource consumption for edge users of LLMs. Our DLRS algorithm improves performance by 15.0% on average for similar requests and by 14.6% for new requests compared to the baselines.**

***Index Terms*—Edge computing, vector database, diffusion model, multi-agent reinforcement learning, request scheduling.**

Received 14 November 2024; revised 5 March 2025; accepted 29 March 2025. Date of publication 18 April 2025; date of current version 12 June 2025. 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, Beijing Normal-Hong Kong Baptist University, Zhuhai under 2023-2024 Grants sponsored by Guangdong Provincial Department of Education, in part by Institute of Artificial Intelligence and Future Networks and Engineering Center of AI and Future Education, Guangdong Provincial Department of Science and Technology, China, in part by Zhuhai Science-Tech Innovation Bureau under Grant 2320004002772, and in part by the Interdisciplinary Intelligence Super Computer Center of Beijing Normal University at Zhuhai. *(Corresponding authors: Zhiqing Tang; Weijia Jia.)*

Zhi Yao is with the School of Artificial Intelligence, Beijing Normal University, Beijing 100875, China, and also with the Institute of Artificial Intelligence and Future Networks, Beijing Normal University, Zhuhai 519087, China (e-mail: yaozhi@mail.bnu.edu.cn).

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

Weijia Jia is with the Institute of Artificial Intelligence and Future Networks, Beijing Normal University, Zhuhai 519087, China, and also with 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).

Digital Object Identifier 10.1109/TSC.2025.3562362

I. Introduction

The Large Language Models (LLMs), as the latest achievement in the field of generative artificial intelligence, can be widely used in production and daily life by achieving accurate dialogue service through reasonable prompt text [1]. LLMs can provide satisfactory answers to users through reasoning, but its extensive parameters demand substantial computational resources, thus prolonging the total time required to generate a comprehensive response for users [2]. Additionally, LLMs relying on traditional cloud computing frameworks introduce additional data transfer delay and network traffic stress [3]. Conversely, edge computing provides both significant computing power and low delay by enabling collaboration between the edge and the cloud [4]. Moreover, it decreases reliance on central cloud data centers, enhancing privacy protection and reducing the risk of sensitive data transmission over the network.

Existing related studies primarily focus on optimizing the challenges of large model sizes and high computational delay by constructing lightweight models [2]. Techniques like model quantization and compression optimize models by reducing parameter counts while minimizing performance loss [5], [6], [7]. Other approaches, such as knowledge distillation and model pruning, enable models of different scales to collaboratively fulfill LLM requests through cloud-edge collaboration [8]. However, these methods involve invasive alterations to the model structure, significantly limiting versatility. Moreover, LLM requests remain computationally intensive, consuming substantial resources, and high delay persists. Therefore, optimizing the Quality of Service (QoS) of LLMs via edge servers remains a worthwhile research problem [9], [10], [11].

Vector databases can cache historical Questions and Answers (QA) as vectors, reducing LLM inference by reusing them when similar requests recur, or enhancing requests through prompt expansion. As a non-invasive LLM optimization technology, it effectively minimizes request completion delay and conserves computational resources while ensuring satisfactory request fulfillment [12], [13], [14]. The main costs of the vector database come from CPU and memory consumption when calculating the similarity between different vectors. As shown in Table I, we have tested the delay required to directly request LLM to return answers and the delay of directly returning answers through database queries at the edge [15]. Even if the vector database stores 11.34×10^6 vectors, the memory required is

TABLE I
COMPARISON BETWEEN LLM AND VECTOR DATABASE

The amount of cached vector (10^6)	Loading memory (GB)	LLM request completion delay (s)		
		Vector Database		Cloud LLM: Qwen14b
		Edge	Cloud	
3.64	4	0.82	1.05	3.34
9.62	10.4	0.84	1.08	3.34
10.60	11.4	0.83	1.03	3.34
11.34	12.2	0.81	1.05	3.34

only 12.2 GB. Additionally, the query delay is still very small when the vector database is deployed on the edge server [16], [17]. Therefore, compared with the large amount of GPU resources and high request delay required to directly deploy LLM on the edge server, deploying a vector database at the edge and storing LLM request results is a very promising method to improve the QoS for edge users.

We propose a novel Vector database-assisted cloud-Edge collaborative LLM QoS Optimization (VELO) framework. In the VELO framework, we deploy the vector database on edge servers and cache some results returned by the LLM. The edge server decides how to process new LLM requests based on request features and vector similarity in the edge vector database, selecting one of the following methods to return the answer: 1) Query the request directly from the edge vector database and return the answer. 2) Utilize similar vectors in the edge vector database to enhance the user request, and then request the LLM from the cloud to return the answer. 3) Directly request the LLM from the cloud to return the answer.

However, several challenges remain unresolved when determining whether LLM requests should be processed by the edge or the cloud. First, the correlation between LLM requests is significant, and there are new features in this scenario, such as one question for multiple answers, multiple queries corresponding to one answer, and timeliness of request results [18], [19]. Second, traditional scheduling methods base decisions on analyzing the similarity between newly arrived requests and the cached vectors. However, different LLM requests exhibit varying sensitivities to the similarity between requests, reflected in the diverse forms and descriptions of language requests [20]. Additionally, the features of LLM requests are discrete, which can pose challenges in early exploration and lead to data wastage in related training models [21]. While Reinforcement Learning (RL) can enhance LLM request scheduling decisions to achieve higher long-term returns through learned policy networks and reward design [22], it suffers from low sample efficiency and limited flexibility in policy updates. Conditional diffusion models address these shortcomings by enabling smoother and more stable policy updates, effectively modeling uncertainties in decision-making, better analyzing the relationship between input conditions and decisions, and making more robust decisions [23].

We present a Diffusion based LLM Request Scheduling (DLRS) algorithm utilizing Multi-Agent Reinforcement Learning (MARL) to enhance the scheduling process of LLM requests [24], [25]. The RL agent is placed on each edge server to determine LLM request scheduling. First, to address the challenges of feature extraction and similarity analysis of diverse requests, we introduce a request feature extraction network built on the Transformer encoder [26], [27]. A diffusion model-based policy network is further proposed to determine the scheduling decisions of requests using extracted features and vector query results from the edge vector database [28], [29]. Second, to address the discrete nature of LLM requests, we suggest a policy network training approach based on expert demonstrations [30]. The network is updated with the support of similar decision-making agents to achieve superior performance. The integration of expert demonstrations effectively resolves concerns regarding poor vector richness and challenges in model fitting due to early action sampling.

In this extension of our previous work [31], we focus on enhancing the algorithm's effectiveness and robustness. Our main improvements over prior work are as follows: 1) A related work section is added to clarify our contributions to LLM services in cloud-edge scenarios and distinguish our approach from existing researches. 2) The algorithm is significantly enhanced by integrating a diffusion-based policy network that uses input states for progressive denoising to derive action probabilities. This innovative approach yields superior scheduling decisions for LLM requests, improving satisfaction and reducing delay. 3) The computational complexity of the algorithm is analyzed, demonstrating that performance improvements are achieved without a significant increase in complexity, confirming its practical feasibility. 4) Large-scale experiments demonstrate the effectiveness and scalability of the proposed algorithm, achieving up to 15% performance improvement over baselines. 5) The performance is evaluated with new requests after freezing parameters and the vector database cache. Additionally, we verify its executability by measuring execution times, showing up to 14.59% performance improvement over the baselines.

The main contributions of this paper are summarized as follows:

1) We introduce the vector database-assisted cloud-edge collaborative LLM QoS optimization framework, VELO. In VELO, vector databases are deployed on edge servers to store LLM request processing results. This framework is highly versatile, maintaining the structure of LLMs and applicable across various LLM implementations.
2) We propose the DLRS algorithm based on a diffusion-based policy network, which utilizes request features and vector query results to denoise and restore the decision of LLM requests. Additionally, to enhance feature extraction and convergence performance, we introduce a feature extraction network and include expert demonstrations during training.
3) We have implemented the VELO framework and DLRS algorithm in a real edge system, complemented by larger-scale simulations using virtual machines. Experimental results indicate the efficacy of our algorithms in enhancing the QoS of edge users when requesting LLMs, leading to higher satisfaction and lower delay.

II. Related Work

1) Quality of Services in LLM: LLMs, primarily deployed in cloud data centers for AI services, face challenges such as response delays and high operational costs. Existing optimization techniques have somewhat mitigated these issues. For example, Lin et al. [35] reduce model size through weight quantization, which lowers computational and memory requirements but may compromise QoS. The advent of edge computing presents a promising optimization strategy for LLM deployment. Chen et al. [36] propose an edge computing framework that segments LLM layers to improve privacy and computational efficiency, but it faces challenges with data consistency and noise interference. Patel et al. [37] distribute computation-heavy prompt calculations and memory-heavy token generation across various servers to enhance throughput. However, these techniques involve human intervention during the model or inference stages, which may limit adaptability and complicate direct deployment in varied scenarios. Rao et al. [38] introduce a method for automatically generating and applying code via natural language instructions, enabling dynamic task deployment between the edge and cloud. However, they overlook task characteristics, causing resource wastage and reduced service quality due to repeated calculations.

2) Decision Algorithms for LLM Request Scheduling: Previous research focuses mainly on RL and Imitation Learning (IL). Fujimoto et al. [39] propose an offline RL algorithm that incorporates a behavior cloning term and data normalization to reduce computational overhead. Gulcehre et al. [30] propose an agent that uses demonstrations to address challenging exploration problems in partially observable environments. In RL, researchers have investigated the use of diffusion models as policy actors within actor-critic frameworks [40]. Wang et al. [41] propose diffusion Q-learning, which leverages expressive diffusion models to represent policies, effectively coupling behavior cloning and policy improvement. Inspired by the above works, we apply diffusion models to Multi-Agent Proximal Policy Optimization (MAPPO), leveraging parallel learning to share the computational load. This is a straightforward approach that can effectively enhance the learning efficiency of diffusion models in large-scale environments. This approach mitigates elevated service delay in the cluster resulting from concurrent cloud LLM requests across all edge servers. We also introduce a feature extraction module and leverage expert demonstrations to address data scarcity and the challenges of random exploration in real-world scenarios.

3) LLM Inference Optimization: With the rising demand for efficient AI services, accelerating LLM inference is a key research focus [42]. Researchers enhance performance by optimizing LLM inference through intelligent task scheduling. He et al. [43] propose an efficient method for large-scale LLM inference services, significantly reducing inference delay through resource analysis, batch scheduling, and model deployment. Fu et al. [44] present a novel scheduler that enhances LLM inference by learning relative output length rankings, thereby reducing delay. Additionally, some studies utilize Retrieval-Augmented Generation (RAG) techniques to lower inference delay and enhance the quality of LLM services. Zhao et al. [45] optimize the illusion, delay, and accuracy of LLMs in data analysis by combining domain knowledge, vector databases, and task decomposition strategies. Hernandez-Salinas et al. [46] integrate RAG into the field of intelligent driving, thereby enhancing its ability to respond to user queries in real time.

The aforementioned studies have optimized LLM services through edge computing and RAG-assisted reasoning while

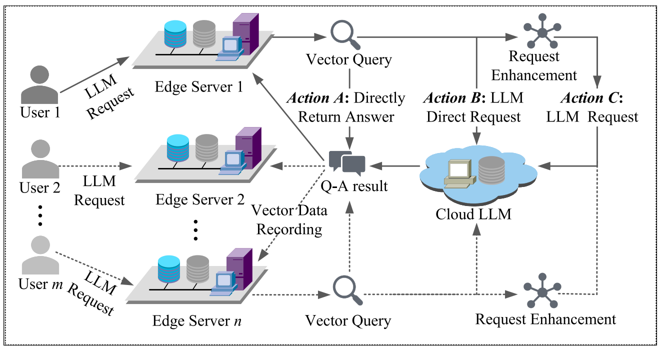


Fig. 1. VELO framework overview.

scheduling tasks based on resource availability. However, they overlook request relationships and lack effective management strategies for similar requests. Furthermore, they do not fully leverage vector databases for data caching.

In contrast to these studies, this paper presents several key differences: 1) We consider the performance of servers equipped with vector databases when faced with similar and new requests, aligning more closely with real-world usage scenarios. 2) We explore the relationship between request features and scheduling decisions using a diffusion model-based policy network, a factor overlooked in previous works [31], [46]. 3) We deploy the prototype system of VELO in a real environment to validate the effectiveness of the DLRS algorithm.

III. VELO Framework and Problem Formulation

The VELO framework, illustrated in Fig. 1, comprises users, edge servers, vector databases, and the cloud LLM. Various users dispatch distinct LLM requests to nearby edge servers. Upon receiving a user request, the edge server decides how to handle it based on the request's content and subsequently provides the result. Using Edge Server 1 as an illustration, upon receiving a request from User 1, it initially queries the local vector database and then assesses request features alongside vector query outcomes. Based on this analysis, it selects LLM request scheduling actions from the following options: *Action A* - returning the answer directly from the vector database, *Action B* - directly requesting the cloud LLM and returning the answer, and *Action C* - augmenting the user request with the vector database query results and requesting the LLM in the cloud to return the answer. Consequently, edge servers can offer high QoS request scheduling decisions through cloud-edge collaboration, predominantly assessed by completion satisfaction and delay. The specifics are elaborated as follows.

Users and edge servers: There exists a set of mobile users $\mathbf{M}$ and a set of edge servers $\mathbf{N}$. At each time slot t, the LLM request is generated by user $m \in \mathbf{M}$ and offloaded to edge server $n \in \mathbf{N}$, which can be represented as $x_{m,n}(t)$. Subsequently, the LLM request is embedded as a vector $\boldsymbol{f}_m(t)$ with a dimension of H. The experience samples generated by the server processing the LLM request is E_n, with a quantity of l_n. The experience generated by all agents can be represented as $\mathbf{E}_N$. In addition,

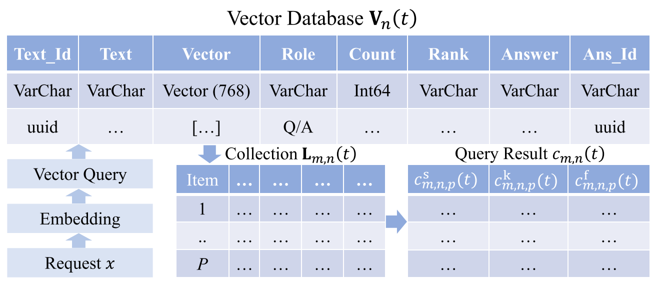

Vector Database $\mathbf{V}_n(t)$
Text_Id | Text | Vector | Role | Count | Rank | Answer | Ans_Id
VarChar | VarChar | Vector (768) | VarChar | Int64 | VarChar | VarChar | VarChar
uuid | … | […] | Q/A | … | … | … | uuid
Vector Query
Embedding
Request x
Collection $\mathbf{L}_{m,n}(t)$
Item | … | … | … | …
1
..
P
Query Result $c_{m,n}(t)$
$c^{\mathrm{s}}_{m,n,p}(t)$ | $c^{\mathrm{k}}_{m,n,p}(t)$ | $c^{\mathrm{f}}_{m,n,p}(t)$

Fig. 2. Vector database structure and data formats.

there are expert demonstrations E_{g}, with a quantity of l_{g}, cached in advance on the server.

Vector database: Each edge server n has a vector database $\mathbf{V}_n(t)$ to cache request completion records, with a current data volume of $k(t)$ [28]. A vector query result of an LLM request in $\mathbf{V}_n(t)$ is a vector data collection $\mathbf{L}_{m,n}(t)$ of P items with the highest similarity to the request. The p^{th} item of data in $\mathbf{L}_{m,n}(t)$ can be described as $\boldsymbol{l}_{m,n,p}(t)$. Vector cache value $c^{\mathrm{r}}_{n,p}(t)$ is cached in the $\mathbf{V}_n(t)$, with an average value of $\bar{c^{\mathrm{r}}_n}(t)$.

The correlation between request $\boldsymbol{f}_m(t)$ and vector query data $\mathbf{L}_{m,n}(t)$ can be expressed as $\boldsymbol{c}_{m,n}(t)$. Then, the correlation between $\boldsymbol{f}_m(t)$ and one of the vector query data $\boldsymbol{l}_{m,n,p}(t)$ can be expressed as $c_{m,n,p}(t) = \{c^{\mathrm{s}}_{m,n,p}(t), c^{\mathrm{k}}_{m,n,p}(t), c^{\mathrm{f}}_{m,n,p}(t)\}$. In which, $c^{\mathrm{s}}_{m,n,p}(t)$ denotes the value of similarity between $\boldsymbol{f}_m(t)$ and $\boldsymbol{l}_{m,n,p}(t)$, $c^{\mathrm{f}}_{m,n,p}(t)$ is the number of times vector data $\boldsymbol{l}_{m,n,p}(t)$ has been used, and $c^{\mathrm{k}}_{m,n,p}(t) \in \{1, 2\}$ denotes the type of $\boldsymbol{l}_{m,n,p}(t)$ including question and answer. We further demonstrate the vector database structure and data formats in Fig. 2.

A. QoS Definition

As shown in Fig. 1, edge servers provide high QoS request scheduling decisions through cloud-edge collaboration, which is mainly measured by request completion satisfaction $q_{m,n}(t)$ and request completion delay $d_{m,n}(t)$ [47].

Request completion satisfaction: The similarity between LLM requests and vector data $\boldsymbol{l}_{m,n,p}(t)$ can be calculated based on L2 euclidean distance [15].

$$J_{m,n,p}(t) = \sqrt{\sum_{h=1}^{H} \left(f_{m,h}(t) - l_{m,n,p,h}(t)\right)^2}, \tag{1}$$

where the dimension of $\boldsymbol{f}_m(t)$ and the vector data obtained from query $\boldsymbol{l}_{m,n,p}(t)$ is H. The $q_{m,n}(t)$ is used to evaluate the satisfaction of completing LLM request at time t. When the reference answer of the LLM request is known, $q_{m,n}(t)$ is measured by the similarity between the current answer and the reference answer, which is calculated as (2).

$$q_{m,n}(t) = -\sqrt{\sum_{h=1}^{H} \left(f^{\mathrm{a}}_{m,h}(t) - f^{\mathrm{r}}_{m,h}(t)\right)^2}, \tag{2}$$

where $f^{\mathrm{a}}_m(t)$ and $f^{\mathrm{r}}_m(t)$ are the answers obtained through DLRS algorithm and reference answers.

Request completion delay: The $d_{m,n}(t)$ represents the delay in the return of request $x_{m,n}(t)$, which is determined as follows:

$$d_{m,n}(t) = \begin{cases} d^{\mathrm{e}}_{m,n}(t), & \mathit{Action\ A} \\ d^{\mathrm{c}}_{m,n}(t), & \mathit{Action\ B} \\ d^{\mathrm{e}}_{m,n}(t) + d^{\mathrm{c}}_{m,n}(t), & \mathit{Action\ C}. \end{cases} \tag{3}$$

In (3), $d^{\mathrm{e}}_{m,n}(t)$ is the delay for the system to complete LLM requests through *Action A*, $d^{\mathrm{c}}_{m,n}(t)$ is the delay for the system to complete LLM requests through *Action B*, and $d^{\mathrm{e}}_{m,n}(t) + d^{\mathrm{c}}_{m,n}(t)$ is the delay for the system to complete LLM requests through *Action C* including the time when the server acquires cache knowledge and the time when the LLM is requested.

B. Vector Database Operations

When the LLM request is resolved by *Action B*, the server embeds the QA and inserts their information including vector embedding, $c^{\mathrm{k}}_{m,n,p}(t)$, $c^{\mathrm{f}}_{m,n,p}(t)$, and vector cache value $c^{\mathrm{r}}_{n,p}(t)$ separately into the vector database. When the server processes LLM requests through *Action A* and *Action C*, vector data most relevant to the LLM request $\boldsymbol{l}_{m,n,p}(t)$ in the request query result $\mathbf{L}_{m,n}(t)$ is filtered to assist in LLM request processing. The principles of filtering can be represented as $\arg\max_{p\in P} F_{n,p}(t)$, which can be further expressed as [48]:

$$F_{n,p}(t) = \phi_1 c^{\mathrm{s}}_{m,n,p}(t) + \phi_2 c^{\mathrm{f}}_{m,n,p}(t), p \in P, \tag{4}$$

where ϕ_1 and ϕ_2 are weights used to balance the effective of these factors. In addition, the cache value of query vector is updated as:

$$c^{\mathrm{r}}_{n,p}(t) = \left(c^{\mathrm{r}}_{n,p}(t-1) + q_{m,n}(t) - d_{m,n}(t)\right)/2, \tag{5}$$

which is beneficial for analyzing and managing cached vector data. When the cached data is correctly matched to the request, $c^{\mathrm{r}}_{n,p}(t)$ will increase, and vice versa.

C. Problem Formulation

We aim to maximize the QoS of the LLM request for the system, which mainly depends on (2) and (3). The goal is to find the best policy to enhance QoS while adhering to constraints. The definition of LLM request scheduling problem is as follows:

$$\begin{aligned} \min W(t) = &\sum_{m\in M}\sum_{n\in N} \left(-\varphi_1 q_{m,n}(t) + \varphi_2 d_{m,n}(t)\right) \\ s.t.\ \ & q_{m,n}(t) < 0, \forall m \in M, \forall n \in N, \\ & d_{m,n}(t) > 0, \forall m \in M, \forall n \in N, \\ & c^{\mathrm{k}}_{m,n,p}(t) \in \{1, 2\}, \forall m \in M, \forall n \in N, \forall p \in P, \\ & c^{\mathrm{r}}_{m,n,p}(t) < 0, \forall m \in M, \forall n \in N, \forall p \in P, \\ & c^{\mathrm{s}}_{m,n,p}(t) > 0, \forall m \in M, \forall n \in N, \forall p \in P, \\ & c^{\mathrm{f}}_{m,n,p}(t) > 0, \forall m \in M, \forall n \in N, \forall p \in P. \end{aligned} \tag{6}$$

The LLM request scheduling problem is NP-hard and can only be solved heuristically. However, most heuristic algorithms make scheduling decisions based on deterministic policies and cannot consider the effects of dynamic environments and continuous decisions. For meta-heuristic algorithms, it is necessary to know all future information, but the future LLM requests are unknown. The Greedy strategy, a common strategy for heuristic algorithms, makes judgements based on the similarity between the question and the database content, but different types of requests have different sensitivities to similarity. Whereas the arrival of LLM requests and updates to the environment are memoryless, so this problem can be modeled as a Markov Decision Process (MDP) [49].

To solve the MDP problem, RL is a promosing method and has been widely adopted. By treating each server as an agent, we propose the DLRS algorithm based on MAPPO, which makes scheduling decisions using a conditional denoising model [25], [50]. The MAPPO network enhances training stability through update-clipped policy adjustments, effectively mitigating policy oscillation risks. Meanwhile, diffusion models employ iterative denoising stages to progressively refine decision distributions. The synergistic integration of MAPPO network with diffusion-based exploration enables enhanced sample efficiency, where structured policy evolution and probabilistic exploration jointly maximize information extraction from constrained data environments.

Through training, the agent considers the associated status of cached vector databases and then selects the action from a global perspective. The long-term QoS of the system for handling LLM requests can be improved by the reward function.

IV. Our Algorithms

A. Algorithm Settings

The DLRS algorithm is founded on MAPPO, with an agent deployed on each edge server to make scheduling decisions independently. Each agent maintains a local state and shares a policy. Furthermore, a global value function incorporates global information and updates the policy network, enabling multiple agents to collaborate and optimize for better long-term benefits for the system. The main settings are outlined as follows.

State: The state provides a comprehensive description of LLM request and the queried vectors. It encompasses the correlation between the LLM request and the vectors, as well as the features of the request. Therefore, the state of the agent on edge server n at time slot t can be divided as follows.

Correlation State: The correlation information between the LLM request and the edge vector database is represented by a matrix $\boldsymbol{c}_{m,n}(t)$, with a dimension of $3 \times P$:

$$\boldsymbol{c}_{m,n}(t) = \begin{bmatrix} c^{\mathrm{s}}_{m,n,1}(t) & \dots & c^{\mathrm{s}}_{m,n,p}(t) & \dots & c^{\mathrm{s}}_{m,n,P}(t) \\ c^{\mathrm{k}}_{m,n,1}(t) & \dots & c^{\mathrm{k}}_{m,n,p}(t) & \dots & c^{\mathrm{k}}_{m,n,P}(t) \\ c^{\mathrm{f}}_{m,n,1}(t) & \dots & c^{\mathrm{f}}_{m,n,p}(t) & \dots & c^{\mathrm{f}}_{m,n,P}(t) \end{bmatrix}. \tag{7}$$

Request Feature State: Considering the different sensitivity of various requests to the vector database, it is crucial to include the request features in the state representation. To accomplish this, we employ a request embedding tool based on the Transformer Encoder [51], defined as:

$$\boldsymbol{f'}_m(t) = T\left[\boldsymbol{f}_m(t)\right], \tag{8}$$

where $T[\cdot]$ represents the network layers based on the Transformer Encoder and fully connected layer used to extract the features of the initial request vector $\boldsymbol{f}_m(t)$. By (8), the request features are further compressed into D, the local state of each agent can be described as a set of dimensions $3P + D$:

$$s_{m,n}(t) = \left\{\boldsymbol{c}_{m,n}(t), \boldsymbol{f'}_m(t) | m \in \mathbf{M}, n \in \mathbf{N}\right\}. \tag{9}$$

Since each server n is treated as an agent and user request m has no specificity during the training process, we use $s_n(t)$ instead of $s_{m,n}(t)$ for subsequent expressions. The global state consists of the observed states of all edge servers, with a dimension of $|\mathbf{N}| \times (3P + D)$, which can be denoted as:

$$\boldsymbol{s}(t) = \left\{s_n(t) | n \in \mathbf{N}\right\}. \tag{10}$$

Action: Each action refers to the scheduling of the LLM request, which can be denoted as:

$$a_n(t) \in \{0, 1\}, \tag{11}$$

where $a_n(t) = 0$ indicates that the LLM request is processed by the edge vector database, with *two sub-actions:* either returning the results directly from the vector database (*Action A*) or using the vector database to enhance the request before querying the LLM in the cloud (*Action C*) [52]. We concatenate vector query content with the user request to form an enhanced request. Specifically, the template for constructing the prompt is: "There may be some relevant reference information, but it is uncertain whether it will be helpful for you to solve the problem." Conversely, $a_n(t) = 1$ signifies that the request is directly forwarded to the LLM in the cloud (*Action B*).

Reward: The objective of each agent is to maximize the reward. In the VELO framework, the aim is to enhance the QoS, encompassing increasing user satisfaction and reducing request completion delay, which can be denoted as:

$$r_n(t) = -\left(W(t) + \varphi_3 a_n(t)\right), \tag{12}$$

where $a_n(t)$ is obtained by sampling from the probability distribution of actions output by the policy network. To address the sparsity of vector database data during algorithm training and encourage more frequent use of these databases, we have introduced a penalty of φ_3 for directly requesting the LLM without utilizing vector databases.

Policy: The output of the policy is the probability distribution of the action choices given the observed environmental state, represented as $\kappa_{x,0}$ in Fig. 3. The diffusion model consists of forward and reverse diffusion processes [23]. Forward diffusion gradually adds Gaussian noise to the data, transforming the action probability distribution into one that approximates a uniform distribution. In contrast, reverse diffusion aims to restore the action probability distribution from the noisy data using state-condition information. By training the neural network, the data that meets the conditions can be progressively denoised based on the current noise and the provided state information.

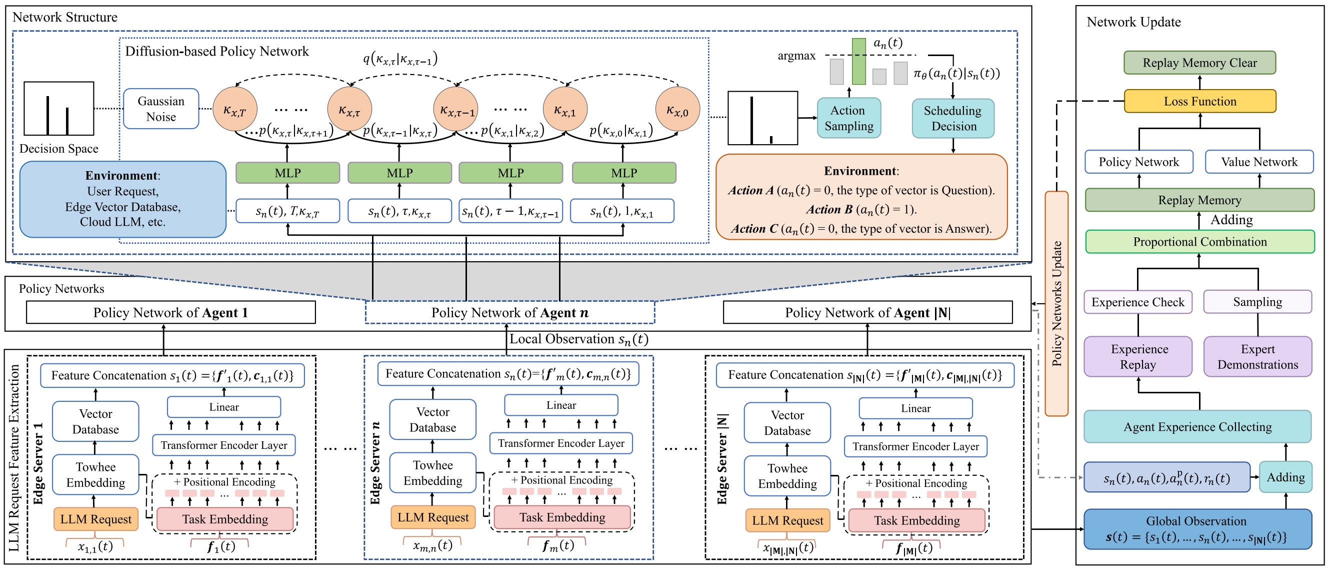

Network Structure
Diffusion-based Policy Network
Gaussian Noise
Decision Space
Environment: User Request, Edge Vector Database, Cloud LLM, etc.
MLP
Action Sampling
Scheduling Decision
argmax
Environment: Action A ($a_n(t)=0$, the type of vector is Question). Action B ($a_n(t)=1$). Action C ($a_n(t)=0$, the type of vector is Answer).
Policy Networks
Policy Network of Agent 1
Policy Network of Agent n
Policy Network of Agent |N|
Local Observation $s_n(t)$
LLM Request Feature Extraction
Edge Server 1
Edge Server n
Edge Server |N|
Feature Concatenation
Vector Database
Towhee Embedding
LLM Request
Linear
Transformer Encoder Layer
+ Positional Encoding
Task Embedding
Network Update
Replay Memory Clear
Loss Function
Policy Network
Value Network
Replay Memory
Adding
Proportional Combination
Experience Check
Sampling
Experience Replay
Expert Demonstrations
Agent Experience Collecting
Global Observation $s(t)=\{s_1(t),...,s_n(t),...,s_{|N|}(t)\}$
Policy Networks Update

Fig. 3. DLRS Algorithm Overview.

The algorithm aims to minimize the error in each denoising step, enabling the model to effectively restore data during backpropagation. We denote the noise at step τ in the forward process as $\kappa_{x,\tau}$, matching the dimension of $\kappa_{x,0}$, where $\tau = 1,\ldots,\mathcal{T}$. The forward diffusion process is modeled as a normal distribution, with a mean of $\kappa_{x,\tau-1}\sqrt{1-\beta_\tau}$ and a variance of β_τ, denoted as:

$$q\left(\kappa_{x,\tau}|\kappa_{x,\tau-1}\right) = \mathcal{N}\left(\kappa_{x,\tau};\kappa_{x,\tau-1}\sqrt{1-\beta_\tau},\beta_\tau\delta\right), \tag{13}$$

where β_τ represents the forward process variance and δ is used to describe the standard normal distribution $\varepsilon \sim (0,\delta)$ that noise follows. ε is a tensor with the same dimensionality as x_0. Based on this, the derivation relationship between $\kappa_{x,\tau}$ and $\kappa_{x,0}$ can be denoted as:

$$\kappa_{x,\tau} = \kappa_{x,0}\sqrt{\bar{\alpha}_\tau} + \varepsilon\sqrt{1-\bar{\alpha}_\tau}, \tag{14}$$

where $\alpha_\tau = 1-\beta_\tau$ and $\bar{\alpha}_\tau$ is the cumulative product of α_τ during the previous denoising step. Based on this, we can further obtain the expression of the reverse process:

$$\kappa_{x,0} = \frac{1}{\sqrt{\bar{\alpha}_\tau}}\kappa_{x,\tau} - \sqrt{\frac{1}{\bar{\alpha}_\tau}-1}\cdot\tanh\left(\varepsilon_\theta\left(\kappa_{x,\tau},\tau,s_n(t)\right)\right), \tag{15}$$

where $\varepsilon_\theta(\kappa_{x,\tau},\tau,s_n(t))$ represents a Multi-Layer Perceptron (MLP) network composed of linear layers. Since there is no pre-existing dataset of optimal scheduling decisions for various input states, and the content of the vector database is gradually populated, the policy network only employs the reverse diffusion process to transition from $\kappa_{x,\tau}$ to $\kappa_{x,0}$ to address this challenge. [25]

Using the Bayesian formula, we convert the reverse diffusion process into the forward process and express it as a Gaussian probability density function. For example, regarding a request on server n, the mean of the reverse diffusion process can be denoted as:

$$\boldsymbol{\mu}_\theta\left(\kappa_{x,\tau},\tau,s_n(t)\right) = \frac{\sqrt{\alpha_\tau}(1-\bar{\alpha}_{\tau-1})}{1-\bar{\alpha}_\tau}\kappa_{x,\tau} + \frac{\sqrt{\bar{\alpha}_{\tau-1}}\beta_\tau}{1-\bar{\alpha}_\tau}\kappa_{x,0}. \tag{16}$$

Then, through (15) and (16), it can be further expressed as:

$$\boldsymbol{\mu}_\theta(\kappa_{x,\tau},\tau,s_n(t)) = \frac{1}{\sqrt{\alpha_\tau}}\left(\kappa_{x,\tau} - \frac{\beta_\tau\tanh(\varepsilon_\theta(\kappa_{x,\tau},\tau,s_n(t)))}{\sqrt{1-\bar{\alpha}_\tau}}\right). \tag{17}$$

When sampling from a distribution, gradients cannot backpropagate through random variables. To enable gradient-based optimization, we implement the reparameterization that disentangles the stochasticity from learnable distribution parameters via differentiable transformations. Specifically, we use the following update rule:

$$\kappa_{x,\tau-1} = \boldsymbol{\mu}_\theta\left(\kappa_{x,\tau},\tau,s_n(t)\right) + \left(\tilde{\beta}_\tau/2\right)^2\odot\varepsilon, \tag{18}$$

where $\tilde{\beta}_\tau$ is a deterministic variance amplitude, which can be denoted as:

$$\tilde{\beta}_\tau = \frac{1-\bar{\alpha}_{\tau-1}}{1-\bar{\alpha}_\tau}\beta_\tau. \tag{19}$$

The inputs to this network include the reverse diffusion noise, the current reverse diffusion step, and the state $s_n(t)$ as a condition. The output is the prediction result of the $\kappa_{x,0}$ at the current step. By applying the softmax function, $\kappa_{x,0}$ is transformed into a probability distribution. The action output is obtained by sampling from the policy network. The goal of the policy network is to optimize the MLP network parameters and obtain the optimal action output by sharing the trajectories of multiple users [53]. Therefore, the cumulative discounted reward J_θ can be calculated based on (12), which can be denoted as:

$$J_\theta = \mathbb{E}_{s_n(t),a_n(t)}\left[\Sigma_t\gamma^t r(t)\right], \tag{20}$$

where γ is a discount factor.

B. *Vector Database-Assisted Feature Extraction*

The LLM Request Feature Extraction module is described in Fig. 3. By extracting the LLM request query result features from the vector database and combining the LLM request features, the high-dimensional LLM request vectors are mapped while the LLM information is fully extracted. As shown in Fig. 3, each edge server, acting as an agent, independently performs the request feature learning process.

Taking agent n as an example, we analyze the process from bottom to top. LLM request is first encoded into vector $\boldsymbol{f}_m(t)$ through the *Towhee* framework and then utilized through two pathways. The DLRS performs a vector query in the vector database to obtain the matching results between LLM and vector database content after comparing $\boldsymbol{f}_m(t)$ with vector data $\mathbf{L}_{m,n}(t)$. In addition, DLRS feeds $\boldsymbol{f}_m(t)$ to the Transformer Encoder, which consists of a position encoder and z multi-head attention modules to obtain a better representation of the request features $\boldsymbol{f}^{\mathrm{e}}_m(t)$. Then $\boldsymbol{f}^{\mathrm{e}}_m(t)$ is further compressed through a linear layer to obtain the LLM request features $\boldsymbol{f'}_m(t)$ for the inputs of policy network and value network. Finally, features $\boldsymbol{c}_{m,n}(t)$ and $\boldsymbol{f'}_m(t)$ will be connected.

C. *Training With Expert Demonstrations*

To solve the problems of sparse training data and slow convergence at the early stage of training, we add expert demonstrations during the training process, as shown in Algorithm 1. To ensure that each network update is valuable, we set the minimum number of expert demonstrations and experience required for network training and updating to $l^{\mathrm{g}}_{\min}$ and $l^{\mathrm{m}}_{\min}$, respectively. Then, we denote the number of network updates as $u(u < u_{\max})$.

Algorithm 1 is deployed on the edge server for network training. The weight of the policy network $b(t)$ obtained from network training is updated and sent down to all agents. Algorithm 1 describes the expert demonstrations assisted network training and updating process in the Multi-Agent Network Policy and Update module in Fig. 3. As shown in Fig. 3, after processing LLM requests, edge servers cache the experience locally and send it periodically to the server for network training. If the server has collected enough experience for each agent, it will train the network according to lines 7–21. The number of expert demonstration items $l^{\mathrm{u}}_{\mathrm{g}}$ used in training decreases as the number of network updates increases. When $l^{\mathrm{u}}_{\mathrm{g}}$ is above the threshold $l^{\mathrm{g}}_{\min}$, the server will sample the expert demonstrations, train and update the network according to lines 10–12. Conversely, considering that the expert demonstrations are outdated for the network, it will be trained and updated using only $\mathbf{E}_N$ according to lines 13–14. To prevent overflow of the replay buffer caused by increased training iterations, the training server completes the network weight update and clears the replay buffer after 50 accesses, as shown in line 23.

Policy optimization: We use the policy gradient method to update the network parameters. The gradient estimation of time step t for parameter θ can be calculated as

Algorithm 1: DLRS Training.

```
1  Input: $E_{\mathrm{g}}$, $\mathbf{E}_N$, $u$, $l_{\mathrm{g}}$, $l_n$
2  Output: $b(t)$
3  Initialize policy network $\pi_\theta$;
4  Initialize $u = 0$, $l_n = 0$;
5  while $u < u_{\max}$ do
6      Check $\mathbf{E}_N$ and calculate $l_n, n = (1, \ldots, N)$;
7      if $\forall l_n > l^m_{\min}, n = (1, \ldots, N)$ then
8          $l^{\mathrm{u}}_{\mathrm{g}} = l_{\mathrm{g}}/u$;
9          for $n = 1, 2, \ldots, N$ do
10             if $l^u_g > l^g_{\min}$ then
11                 Sample $l^{\mathrm{u}}_{\mathrm{g}}$ expert demonstrations from
                   $E_{\mathrm{g}}$ to get $E'_g$;
12                 Get replay memory by $\{E'_g, \mathbf{E}_N\}$;
13             else
14                 Get replay memory by $\mathbf{E}_N$;
15             end if
16             Compute $\hat{A}(1), \ldots, \hat{A}(T)$ by Eq. (22);
17             Compute $L(\theta)$ by Eq. (24);
18             Optimize the network and get $b(t)$;
19             $u = u + 1$;
20             Update weights $\pi_{\mathrm{old}} \leftarrow \pi_\theta$;
21         end for
22         Send $b(t)$ to all agents;
23         Drop $\mathbf{E}_N$ on the edge server;
24     end if
25 end while
```

$$\hat{g}_t(\theta) = \frac{1}{N} \sum_{n=1}^{N} \mathbb{E}_{\tau_n} \left[\sum_{t=0}^{T_n} \nabla_\theta \log \pi_\theta(a_n(t)|s_n(t)) \hat{A}(t) \right], \tag{21}$$

where $\mathbb{E}_{\tau_n}$ represents the expectation for the trajectory of agent n, $\pi_\theta(a_n(t)|s_n(t))$ is the policy function of agent n, and $\hat{A}_n(t)$ is an adjusted advantage function introduced in the DLRS algorithm to handle the mutual influence between agents in multi-agent environments, which can be denoted as:

$$\hat{A}(t) = \delta(t) + \gamma\lambda\delta(t+1) + \cdots + (\lambda\delta)^{T-t+1} \lambda(T-1), \tag{22}$$

where $\delta(t) = r(t) + \gamma V^\pi(\boldsymbol{s}(t+1)) - V^\pi(\boldsymbol{s}(t))$, and if the agent starts in state $s(t)$ and takes action according to policy π, the value function $V^\pi(\boldsymbol{s}(t))$ gives the expected return, which can be denoted as:

$$V^\pi(\boldsymbol{s}(t)) = \mathbb{E}_{\tau_{\mathrm{traj}} \sim \pi} \left[R(\tau_{\mathrm{traj}}) | \boldsymbol{s} = \boldsymbol{s}(t) \right], \tag{23}$$

where $R(\tau_{\mathrm{traj}})$ is the cumulative reward function and τ_{traj} is a trajectory generated based on policy π. The loss function is constrained to ensure that the difference between new and old parameters is not too large, which can be expressed as [54]:

$$L(\theta) = \hat{\mathbb{E}} \left[L_{\mathrm{clip}}(\theta) - c_1 L_{\mathrm{E}}(\theta) + c_2 C_{\mathrm{e}}[\pi_\theta](s_n(t)) \right], \tag{24}$$

where $L_{\mathrm{E}}(\theta)$ is the mean square error of the reward and the corresponding state value, $C_{\mathrm{e}}[\pi_\theta](s_n(t))$ is the cross entropy of the action probability distribution, c_1 and c_2 are hyperparameters, with values of -0.5 and 0.01, respectively. The $L_{\text{clip}}(\theta)$ is denoted as:

$$L_{\text{clip}}(\theta) = \hat{\mathbb{E}}\left[\min\left(\psi_t(\theta)\hat{A}(t), \text{clip}\left(\psi_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}(t)\right)\right], \tag{25}$$

where $\psi_t(\theta)$ is the probability ratio of actions under two policies, clip$(\psi_t(\theta), 1-\epsilon, 1+\epsilon)$ is used to clip the $\psi_t(\theta)$ between $(1-\epsilon, 1+\epsilon)$, and ϵ is a hyper-parameter, e.g., $\epsilon = 0.2$, which can make the value of actions lower or higher than the average amplitude between $(1-\epsilon, 1+\epsilon)$.

Subsequently, we use the gradient ascent to update the policy parameters.

$$\theta \leftarrow \theta + \alpha_{\text{g}} \nabla_\theta L_{\text{clip}}(\theta), \tag{26}$$

where α_{g} is the learning rate.

D. Diffusion-Based LLM Request Scheduling

Algorithm 2 describes the DLRS algorithm, whose output is the reward value $r_n(t)$ that represents the QoS of agent n. The reception time of policy network weight on edge server n is $b_n^{\text{T}}(t)$ and the periodicity of vector database content checking and updating is ξ. First, agent n creates a collection of vector databases and obtains the update time of the local network weights, as shown in lines 4–5. Prior to processing each request, edge servers must verify whether the local policy network weights have been loaded with the most recent version, as shown in lines 8–11. Then, the server processes the arriving LLM requests and stores experience, as shown in lines 12–18. In each time slot, the policy network initializes input noise, as shown in line 12. It then employs the state input as a condition for denoising, iteratively calculating the log probability of action selection $a_n^{\text{p}}(t)$. Then the agent n samples actions, as shown in lines 13–16. In addition, the server periodically executes the vector data eviction policy, as shown in lines 19–26.

When the experience accumulated by agents is sufficient, it is sent to the training server, as shown in lines 27–30. Then, after completing the network update, new weights are sent to all agents.

Vector data eviction policy: Low-quality data records resulting from poor policies need to be updated and processed promptly. Therefore, in addition to data insertion and update operations, we also introduces checks and updates for vector data quality. For each elapsed time ξ, data in the vector database will be dropped when they satisfy $c_{n,p}^{\text{r}}(t) < c_n^{\bar{\text{r}}}(t)$, as these data usually exhibit characteristics such as training failures, mismatches, and outdated answers.

Algorithm 2: The DLRS Algorithm.

```
1  Input: $T$, $b_n^{\text{T}}(t)$, $u$, $x_{m,n}(t)$, $l_n$, $\xi$, $c_{n,p}^{\text{r}}(t)$, $k(t)$, $\mathcal{T}$, $\kappa_{x,\mathcal{T}}$
2  Output: $r_n(t)$
3  Initialize $u = 0, t = 0$;
4  Create vector data collection;
5  $b_n^{\text{i}} = b_n^{\text{T}}(0)$;
6  Reset environment, get $x_{m,n}(0)$ and $s_n(0)$ ;
7  while $u < u_{\max}$ and $t < T$ do
8      if $b_n^T(t) \neq b_n^i$ then
9          Update networks by calling Algorithm 1;
10         $b_n^{\text{i}} = b_n^{\text{T}}(t)$;
11     end if
12     Initialize $\kappa_{x,\mathcal{T}}$ based on normal distribution;
13     for $\tau = \mathcal{T}-1, \mathcal{T}-2, \ldots, 1$ do
14         Get $\kappa_{x,\tau}$ by $s_n(t)$ and $\kappa_{x,\tau+1}$;
15     end for
16     Get $a_n(t), a_n^{\text{p}}(t)$ by $\kappa_{x,0}$;
17     Execute $a_n(t)$ by Eq. (4) and get $r_n(t)$ by Eq. (12);
18     Store $\{s_n(t),\ a_n(t),\ a_n^{\text{p}}(t),\ r_n(t)\}$ in replay memory;
19     if $t\%\xi = 0$ then // Vector data eviction policy
20         $c_n^{\bar{\text{r}}}(t) = \frac{1}{k(t)}\sum_{p=1}^{k(t)} c_{n,p}^{\text{r}}(t)$;
21         for $p = 1, 2, \ldots, k(t)$ do
22             if $c_{n,p}^r(t) < c_n^{\bar{r}}(t)$ then
23                 Drop the vector data;
24             end if
25         end for
26     end if
27     if $t\%l_{\min}^m = 0$ then
28         Send experience to the training server;
29         $l_n = 0$;
30     end if
31     Generate new request $x_{m,n}(t+1)$; Get $s_n(t+1)$ based on $x_{m,n}(t+1)$;
32     $s_n(t) = s_n(t+1)$;
33     $x_{m,n}(t) = x_{m,n}(t+1)$;
34     $u = u + 1$;
35     $t = t + 1$;
36 end while
```

E. Computational Complexity Analysis

The DLRS algorithm is mainly composed of $\mathbf{N}$ policy networks and a value network. The decision-making process can be mainly divided into three parts: state observation, action selection and reward calculation. The computational complexity of each section is analyzed below. First, the state is shown in (10), and the complexity of this part is determined by request feature extraction calculation and vector query. The computational complexity of the Transformer Encoder module is primarily influenced by its multi-head self-attention operation, which is one of its key components. This complexity is determined by the embedding dimension H and the number of patches C, which can be expressed as $O(H^2C + HC^2)$ [55]. Assuming that the Transformer Encoder module contains L_1 layers, its overall complexity can be calculated as $O(L_1 \times (H^2C + HC^2))$. Second, $\kappa_{x,0}$ is obtained through reverse diffusion in the policy network. $\varepsilon_\theta(\kappa_{x,\tau}, \tau, s_n(t))$ is an MLP network with L_1 hidden layers and G_1 neurons in each layer. The complexity of reverse diffusion calculations can be expressed as $O(\mathcal{T} \times ((3P+D) \times G_1 + L_2 \times G_1^2))$ [56]. The complexity of action selection and reward calculation for all servers can be expressed as $O(|\mathbf{N}|)$.

In network updates, the total number of experiences used is constant, but the ratio of expert experiences to agent experiences varies. The value network is an MLP network with L_3 hidden layers and G_2 neurons in each layer. The complexity of network updating can be expressed as $O(|\mathbf{N}| \times (3P + D) \times G_2 + L_3 \times G_2^2)$. The other operations in the DLRS have a relatively small impact on computational complexity analysis. Therefore, the complexity of the DLRS algorithm is $O(L_1 \times (H^2C + HC^2) + \mathcal{T} \times ((3P + D) \times G_1 + L_2 \times G_1^2) + |\mathbf{N}| \times (3P + D) \times G_2 + L_3 \times G_2^2)$.

V. System Implementation and Experiments

A. System Implementation

We have implemented a VELO prototype system and deployed the DLRS algorithm into the system using Python. The VELO system consists of edge servers and cloud LLM, where the edge mainly consists of *Towhee* Service and *Milvus* Service [15], [57], all of which are deployed through containers. The experimental platform consists of three desktops featuring an Intel i9-10900 K 10-Core CPU and NVIDIA RTX 2070 Super GPU. Data from edges is transmitted to a desktop for network training, which is equipped with an Intel i7-13700KF 16-Core CPU and an NVIDIA RTX 4070ti GPU.

The Qwen7b quantized by int8 is deployed as the cloud LLM, based on FastAPI and Uvicorn servers [32]. Uvicorn is an Asynchronous Server Gateway Interface (ASGI) server that supports asynchronous programming. FastAPI is a high-performance web framework built on Starlette. Therefore, we can achieve streaming output from the LLM by gradually sending data to the client through the server, and expose the LLM service via host and port configuration. A high-performance workstation with an Intel i9-14900KF 24-Core CPU, an NVIDIA RTX 4090 GPU, and 64 GB RAM is used as the cloud. Given the slow processing speed for parallel LLM requests and the limited experimental devices, some experiments are conducted in virtual machines (VMs). Each VM has a minimum of four CPU cores, 50 GB storage, and 8 GB RAM. The system overview we have implemented is illustrated in Fig. 4.

LLM requests from users can be offloaded to the nearest server and then vectorized through *Towhee* service and input into a vector database to query relevant vectors. The *DLRS Algorithm Layer* can determine the scheduling location of LLM requests, including directly sending results to users through *Action A* at the edge and sending LLM requests to the cloud through *Action B* or *Action C*. The cloud server outputs the answer to the LLM request after instantiating FastAPI. Each server sends experience to the edge server for training through the TCP protocol and obtains the latest network weights through the FTP protocol. In Fig. 4, directional arrows denote both data flow pathways and inter-module functional dependencies within the system architecture.

Milvus service: Milvus service is an open-source vector database that enables vector similarity search [15]. The *Milvus* data service can be deployed and managed using distributed containers, allowing for higher data throughput and more convenient data indexing and querying. Vector data is usually stored

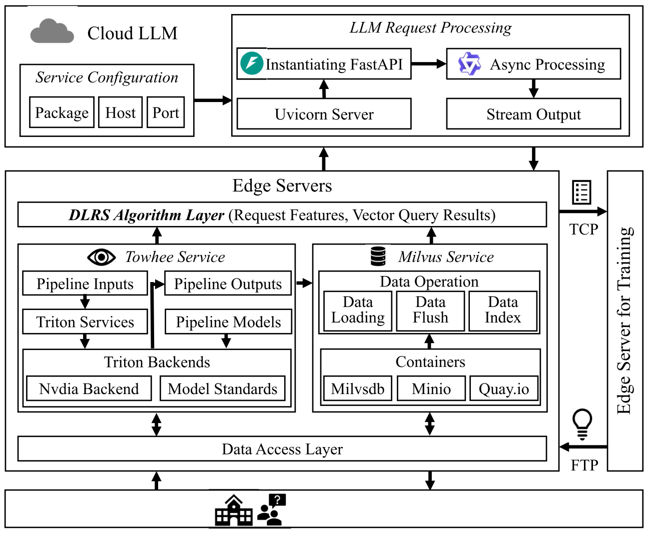


Fig. 4. System implementation details.

on the hard drive through data flush and loaded into memory to accelerate queries as they are used. The type of collection index is *IVF_FLAT*, which is an index type based on inverted files. Based on this, we create 128 inverted lists for the inverted files. When performing a vector query, the 10 closest candidate items are searched from the inverted lists for precise distance calculation [15].

Towhee service: We use the open-source *Towhee* framework and adopt *gpt-neo-1.3B* as the embedding model [51], [57]. This is a trained GPT-style language model, which supports multilingual embedding requests. Furthermore, the Towhee framework also supports Triton acceleration, making it particularly suitable for edge deployment. We have deployed a *Triton* acceleration framework based on the NVIDIA plugin on the server to improve embedding speed. The LLM requests into the Towhee service first enters a pipeline, where it is accelerated by pre-defined models and the Triton backend to obtain vector embeddings. Through the pipeline cached embedding model, LLM requests can be embedded in vectors with a dimension of 768. These embeddings can be used for vector querying as well as for the task feature extraction module in the DLRS algorithm.

B. Experimental Settings

Data preprocessing: We use the multilingual open dialogue dataset *oasst1* as LLM requests [33], [34]. Due to the multilingual and diverse nature of the LLM request, the highest-ranked dialogue from the *oasst1* is chosen as the training set. Each item in the training set is expanded into five versions: English, Spanish, German, Chinese, and Russian using Qwen14b. The test set is obtained by restating the Question in QA pairs through Qwen14b.

During training, edge servers periodically transmit the experiences to the server for training. Updated network weights are subsequently propagated back to all edge servers through FTP protocol. For testing, both network parameters and vector

TABLE II
HYPERPARAMETER SETTINGS

Type	Hyperparameter	Value
Policy Netowrk	Input dimension	$3P + D = 60$
	Time embedding	16
	Hidden layers	2 Full connection (256)
	Activation function	Mish
	Output dimension	2
	Denoising steps	$\mathcal{T} = 10$
	Learning rate	3e-4
Value Netowrk	Input dimension	$\lvert\mathbf{N}\rvert \times (3P + D)$
	Hidden layers	2 Full connection (384)
	Activation function	Tanh
	Learning rate	1e-3
	Loss Function	MSELoss
Feature Extraction	Input dimension	$H = 768$
	Input layer	6-head transformer
	Linear layers	5 Full connection
	Activation function	Tanh
	Output dimension	$D = 30$
Other	Query results number	$P = 10$
	Discount factor	0.99
	Clipping threshold	0.2
	Reply buffer accesses	50
	Optimizer	Adam

database entries remain frozen. The test requests are sent simultaneously to all edge servers to evaluate the cache contents of each edge server and the decision-making ability of agents. Before the experiment, we populate the edge vector database with some random data sourced from non-experimental data in *oasst1*. This initialization ensures that all possible actions are available and allows the DLRS algorithm to accumulate negative trajectory samples.

Parameter settings: For a training dataset of 3000 samples, 13500 rounds are used for training and 500 rounds for testing. As the dataset size increases, the number of training rounds is adjusted accordingly. Results are recorded every 300 rounds during the experiment. The learning rates for the policy and value networks are set to 0.0003 and 0.001, respectively. The discount factor is set to 0.99. The DLRS algorithm adjusts the noise intensity in a uniformly increasing manner at each step, requiring 10 denoising steps to restore the action decision. The hyperparameters are listed in Table II.

Baselines: The following baselines are conducted.

1) *Greedy-0.1, Greedy-0.3, Greedy-0.5 [58]:* These algorithms make scheduling decisions by judging the vector query results and the fixed threshold. When the vector query results is higher than the threshold, the server directly requests the LLM; otherwise, it use the vector database to enhance the request. Experiments are conducted with thresholds of 0.1, 0.3, and 0.5.
2) *Greedy-LLM:* The algorithm completes LLM requests with an initial estimated reward, updated periodically based on directly requesting LLM completion satisfaction. Specifically, the algorithm records the similarity between the answers obtained by directly requesting LLM and the reference answers, periodically updating this average as the evaluation reward value. When the vector query result is better than the reward, the vector database is used to complete requests; otherwise, the server requests the LLM directly.
3) *MAPPO [59]:* This algorithm is a multi-agent proximal policy optimization algorithm based on deep RL.
4) *G-MAPPO:* This algorithm is a MAPPO algorithm with expert demonstrations.
5) *T-MAPPO:* This algorithm is a MAPPO algorithm with an external Transformer Encoder, which takes the connection of the extracted request vector features and the vector cache comparison results as input. The T-MAPPO algorithm is used to verify the effectiveness of the feature extraction module in Fig. 3.
6) *LRS [31]:* This algorithm is a MAPPO algorithm with an external Transformer Encoder, which introduces expert demonstrations during the training process.
7) *DLRS-L5, DLRS-L15 [25]:* The DLRS algorithm adjusts the noise intensity in a uniformly increasing manner at each denoising step, making adding or removing noise smoother and more stable. In experiments, tests are conducted using 5 and 15 denoising steps.
8) *DLRS-V5, DLRS-V10, DLRS-V15 [23]:* The forward process variance of DLRS algorithms is controlled by the Variational Posterior (VP) scheduler [60], with experiments conducted using 5, 10, and 15 denoising steps.
9) *Random:* Edge servers complete LLM requests by randomly selecting actions.

Evaluation methods: Each edge server receives different requests, accumulating various content in the vector database. This aligns better with real-world service scenarios and helps us collect more diverse agent experiences, thereby improving the training efficiency of the algorithm. Because of the differing quality of cached data for each server during training, we suggest two evaluation methods, which can be outlined as follows:

- LLM requests are offloaded to the nearest server.
- LLM requests are offloaded to all servers, and the fastest response is returned as the answer to the request. While distributed servers collaborate in training, disparities in their processing capabilities arise from variations in vector databases.

C. Experimental Results

To demonstrate the effectiveness of the DLRS algorithm, we carry out a more detailed evaluation within the system [31]. After every 300 LLM request is fulfilled, we document an average outcome in the figures. Within the figures, LLM Direct Request Frequency represents the proportion of direct processing requests via LLM out of the total processed requests. The weight ratio is computed as $w = \varphi_2/\varphi_1$. The reward value in the figures represents the average value achieved by all servers that complete LLM requests. To enhance the fitting ability of the network, we have increased the reward value in (12) by a factor of 10.

Training trajectories: From Fig. 5, it is evident that the DLRS algorithm outperforms others when the number of

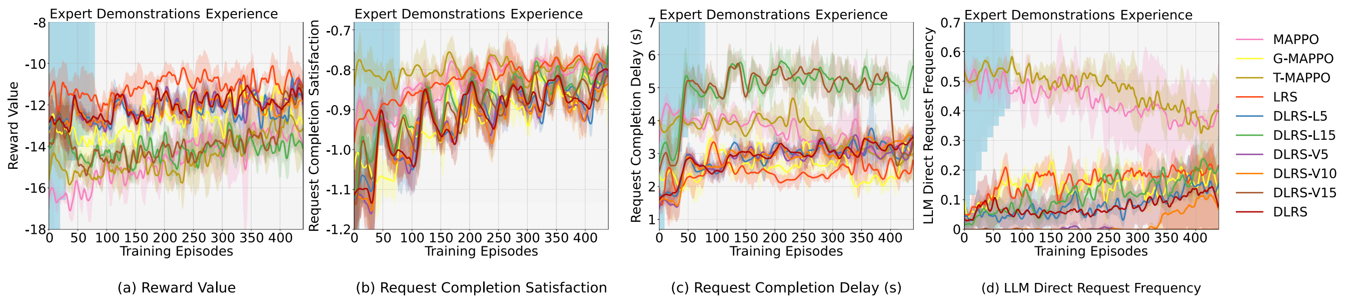
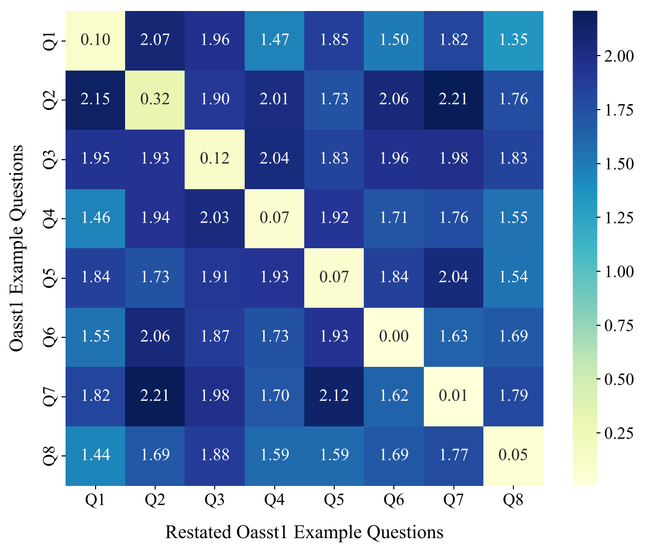


Fig. 5. Performance of DLRS during training episodes with a weight ratio $\omega = 0.1$.



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Q1	0.10	2.07	1.96	1.47	1.85	1.50	1.82	1.35
Q2	2.15	0.32	1.90	2.01	1.73	2.06	2.21	1.76
Q3	1.95	1.93	0.12	2.04	1.83	1.96	1.98	1.83
Q4	1.46	1.94	2.03	0.07	1.92	1.71	1.76	1.55
Q5	1.84	1.73	1.91	1.93	0.07	1.84	2.04	1.54
Q6	1.55	2.06	1.87	1.73	1.93	0.00	1.63	1.69
Q7	1.82	2.21	1.98	1.70	2.12	1.62	0.01	1.79
Q8	1.44	1.69	1.88	1.59	1.59	1.69	1.77	0.05

Fig. 6. Vector similarity metrics between requests.

TABLE III
REQUEST COMPLETION RESULTS WITH REFERENCE ANSWER ASSISTANCE

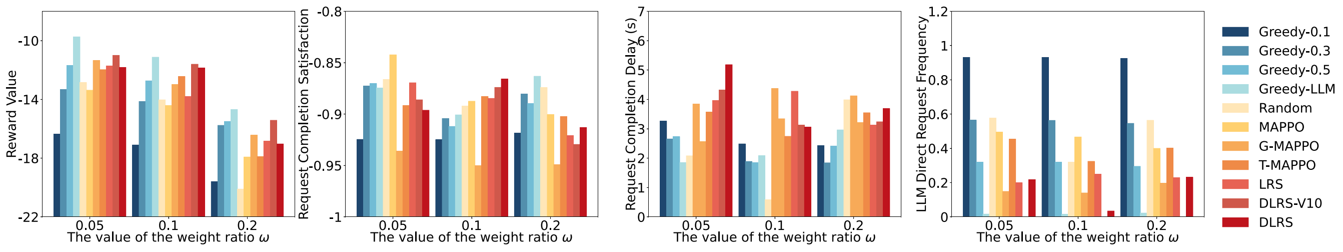
Action	Performance (Completion satisfaction/delay(s))					
	EN	CH	RU	DE	ES	Restated
Action A	-0.51	-0.56	-0.58	-0.59	-0.35	-0.55
	0.10	0.05	0.02	0.04	0.10	0.03
Action B	-0.51	-0.56	-0.58	-0.59	-0.35	-0.55
	6.00	7.56	6.01	11.19	7.79	7.81
Action C	-0.45	-0.21	-0.20	-0.13	-0.14	-0.37
	10.79	10.41	11.34	12.49	12.72	11.06

denoising steps is set to 10, significantly enhancing request completion satisfaction. Additionally, a comparison between the G-MAPPO algorithm and other algorithms reveals that the former achieves higher initial reward values and better vector database utilization when guided by expert demonstrations. In contrast to the LRS algorithm, the DLRS algorithm requires a longer training duration due to its involvement of multiple time steps. At each time step, the algorithm must learn how to recover scheduling decisions from conditional noise data, which requires multiple iterations. Although the completion delay for the DLRS algorithm during training is not the lowest, the quality of the resulting vector database is superior, leading to higher utilization. This is illustrated in Figs. 5(d) and 7, where Fig. 5 shows that the shaded area under the DLRS algorithm curve is relatively small when the number of denoising steps is 10, indicating a stronger capacity for simultaneous optimization of multiple agents and greater algorithm stability. Therefore, in subsequent experiments, we will focus exclusively on the DLRS algorithm with 10 denoising steps.

Analysis of vector similarity calculation results: To illustrate the utility of vector similarity calculations in analyzing the similarity between requests, we sample eight requests from *oasst1* and rephrased them. The calculation of vector similarity between these questions follows (1). Specifically, the sampling requests are as follows, where the second and sixth requests are in German, the seventh is in Chinese, and the eighth is in Russian.

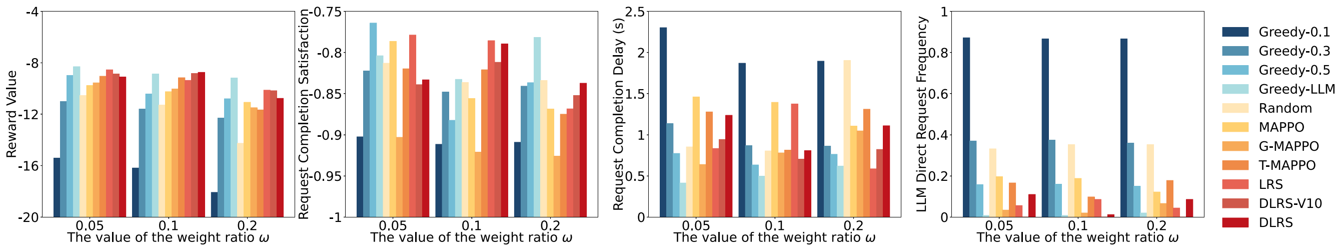
1) 'What is a fun science project I can do with my son?'
2) 'What does Christianity think about this?'
3) 'Is TypeScript better than JavaScript?'
4) 'What would be the three best items to bring to a desert island?'
5) 'Why do humans need to sleep?'
6) 'Can you tell me something about protein simulation based on coarse glass mirror techniques?'
7) 'I have a Pandas dataset containing three columns: 'x', 'y', and 'z'. I want to group by 'x' and calculate the average occurrence of the value 'a' in the 'y' column for each group. Additionally, I only want to consider the occurrence frequency when the values in the 'z' column are not equal to 'b'.'
8) 'I want to convince my mother that giving a rabbit as a gift for a six-year-old girl is not a good idea. Could you give me some reasons why giving a rabbit to a six-year-old girl is not a good idea?'

As shown in Fig. 6, there is a clear indication of low similarity between different LLM requests, while the similarity between the original questions and their rephrased questions is significantly higher. In Table III, we demonstrate the execution results of the seventh request under high-similarity vector queries. We expand the linguistic expressions of the request into English, Chinese, Russian, German, and Spanish versions. Additionally, we rephrase the original request to create variants as "Given a pandas DataFrame containing columns 'x', 'y', and 'z', first exclude all rows where column 'z' equals 'b'. Then, for each unique category in column 'x', compute the mean proportion of



(a) Reward Value (b) Request Completion Satisfaction (c) Request Completion Delay (s) (d) LLM Direct Request Frequency

Fig. 7. Performance with different weight ratio ω when LLM requests are offloaded to the nearest edge server.



(a) Reward Value (b) Request Completion Satisfaction (c) Request Completion Delay (s) (d) LLM Direct Request Frequency

Fig. 8. Performance with different weight ratio ω when LLM requests are offloaded to all servers.

entries in column 'y' that hold the value 'a' within the filtered subset."

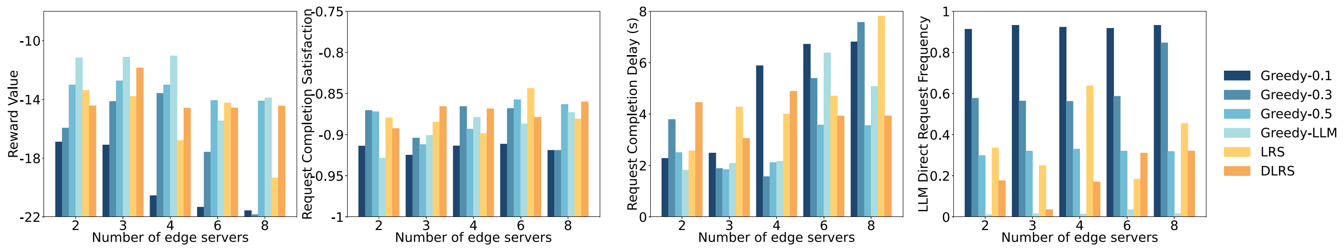
Action B is directly processing the request using an LLM. After caching these results in the vector database, *Action A* simply retrieves and returns the cached answers, achieving significant delay reduction. In contrast, *Action C* incorporates retrieved reference answers as vector-based knowledge to enhance request execution. The similarity metrics between the query and cached reference answers are 0.59, 0.68, 0.58, 0.54, 0.63, and 0.64, respectively—significantly lower than the inter-query similarity values observed in Fig. 6 for non-similar requests. Experimental results show that high-quality cached answers significantly enhance the efficiency of processing related requests.

Performance with different reward weights: The reward function in (12) primarily considers the main factors affecting user Quality of Experience (QoE), namely completion satisfaction and delay. Our goal is to minimize the completion delay while ensuring the completion satisfaction. Considering that the reward value directly impacts the learning effectiveness of RL, we have pre-determined a reasonable range for the weight factors through experiments based on the content of the *oasst1* dataset and the inference speed of the LLM. It can be seen from Figs. 7(b) and 8(b) that the DLRS algorithm has a significant advantage with $w = 0.1$. Compared with the LRS algorithm, it can significantly reduce request completion delay without affecting request completion satisfaction. The performance of DLRS algorithm is generally superior when LLM requests are offloaded to the nearest edge server. RL algorithms perform suboptimally at other weights because changes in weight influence reward definitions, which in turn affect the learning effectiveness of the policy network. Unreasonable weight settings can decrease request completion satisfaction and increase delays, preventing the achievement of a dynamic optimal state. As a result, we set w to 0.1in subsequent experiments.

To encourage algorithms to use vector data related to requests, we have imposed corresponding penalties for directly requesting LLM without using vector databases. However, this does not imply that every request during the testing phase must rely on the vector database. While the test request is a variant of the training data, not all vector databases on the server may have cached relevant content. As shown in Figs. 7 and 8, although the Greedy-LLM and DLRS-V10 algorithms achieve high reward values, the satisfaction with request completion remains low.

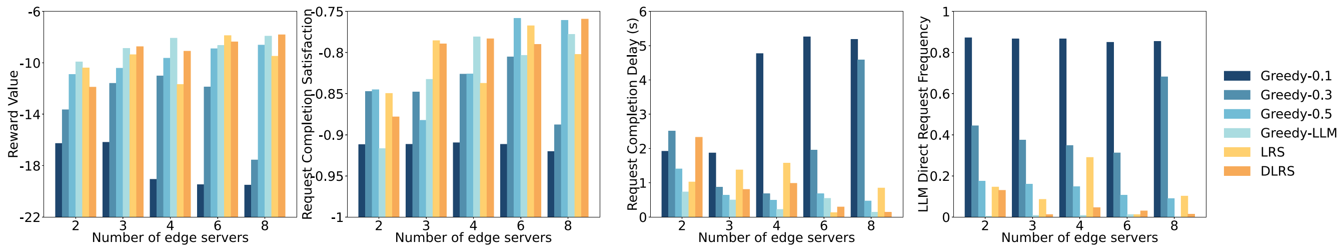
Due to request scheduling decisions that minimize the use of *Action B* during training, the Greedy-LLM and DLRS-V10 algorithms incorrectly match vector knowledge data with LLM requests. This mismatch leads to decreased request completion satisfaction and low-quality caching in the vector database. Clearly, when the quality of vector data is poor, errors occur in completing requests through Action A. The training process of the DLRS-V10 algorithm is shown in Fig. 5(b) and (d), and subsequent experimental results further confirm this.

From Figs. 7(a) and 8(a), the reward obtained with the Greedy-LLM algorithm is higher. Distributing each LLM request across multiple servers is effective because each server caches different vector data during training, leading to varying request processing capabilities. Combining multiple servers for a single request can significantly reduce completion delays. With w at 0.1, DLRS algorithm outperforms in all metrics, significantly reducing completion delay compared to LRS while preserving completion satisfaction. Fig. 8(b) and (d) demonstrate that the DLRS algorithm offers faster feedback to requests due to superior vector data cache content.



(a) Reward Value (b) Request Completion Satisfaction (c) Request Completion Delay (s) (d) LLM Direct Request Frequency

Fig. 9. Performance with different number of edge servers when LLM requests are offloaded to the nearest edge server.



(a) Reward Value (b) Request Completion Satisfaction (c) Request Completion Delay (s) (d) LLM Direct Request Frequency

Fig. 10. Performance with different number of edge servers when LLM requests are offloaded to all servers.

The experimental results show that the DLRS algorithm can enhance the overall reward value by up to 10.47% with $w = 0.1$ when offloading LLM requests to the nearest edge server. Additionally, the system's request completion satisfaction improves by an average of 6.40%, 4.26%, 5%, 3.88%, 3%, 2.44%, 8.89%, 2%, 2.14%, and 1% compared to the Greedy-0.1, Greedy-0.3, Greedy-0.5, Greedy-LLM, Random, MAPPO, G-MAPPO, T-MAPPO, LRS, and DLRS-V10 algorithms, respectively.

Furthermore, when the DLRS algorithm offloads LLM requests to all servers with $w = 0.1$, it can boost the overall reward value by up to 15% . The request completion satisfaction increases by an average of 13.38%, 6.89%, 10.52%, 5.16%, 5.59%, 7.74%, 14.28%, 3.82%, and 2.74% compared to the Greedy-0.1, Greedy-0.3, Greedy-0.5, Greedy-LLM, Random, MAPPO, G-MAPPO, T-MAPPO, and DLRS-V10 algorithms, respectively. DLRS algorithm reduces request completion delay by 41.39% while maintaining a request completion satisfaction level comparable to the LRS algorithm.

Performance with different number of servers: We also evaluate the performance of algorithms with varying numbers of servers, as depicted in Figs. 9 and 10. Since DLRS algorithm significantly outperforms basic MARL and random algorithms in previous experiments, we exclude them here. As shown in Fig. 10(b), the performance of the DLRS algorithm improves with more servers due to its collaborative training approach, which enables multiple agents to share experiences. Because DLRS requires multiple iterations for scheduling decisions, incorporating more learning experiences reduces noise in the learning process. Conversely, the LRS algorithm, based on fully connected structures, performs better with fewer servers due to its simpler decision-making process.

As the cluster scales, requests to individual servers become sparser, leading to decreased cache density in the vector database. In such sparse scenarios, the Greedy-0.5 algorithm sustains higher database utilization, thereby outperforming alternatives when processing similar requests. The LRS algorithm has a higher proportion of directly returning cached answers through *Action A*, thereby reducing delay. However, when handling new requests, the algorithm's effectiveness may degrade due to retrieving excessive semantically irrelevant knowledge segments. It is evident that the DLRS algorithm demonstrates better adaptability.

The experimental results show that the DLRS algorithm can increase the overall reward value by up to 22% when there are at least 3 servers and LLM requests are offloaded to all of them. Specifically, the request completion satisfaction of the system is increased by 14.53%, 7.27%, 3.27%, 2.27%, and 2.2% on average compared with Greedy-0.1, Greedy-0.3, Greedy-0.5, Greed-LLM, and LRS algorithms, respectively.

Performance with different training set sizes: After obtaining parameters with performance advantages, we further verify the performance of algorithms on different training set sizes. Figs. 11 and 12 show the evaluation results on different training set sizes. From Fig. 11(a), it can be seen that the DLRS algorithm can achieve high reward values after training with different training set sizes. However, the addition of new data leads to a decrease in request completion satisfaction for the DLRS algorithm when the network structure remains unchanged. This occurs because, in conditional denoising, noise guides the model to progressively generate decision scheduling outputs that align with a given state condition input. When the network structure is fixed, the increase in training data complicates the exploration of noise during the generation process. Hence, to achieve optimal performance with varying training set sizes, adjustments to the DLRS network structure are necessary. We increase the hidden layer of the policy network to 384. Then, we extend

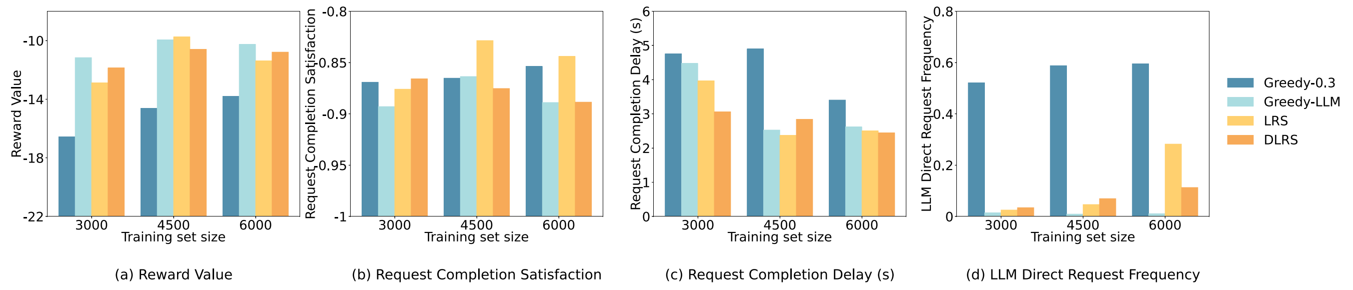


Fig. 11. Performance with different training set sizes when LLM requests are offloaded to the nearest edge server.

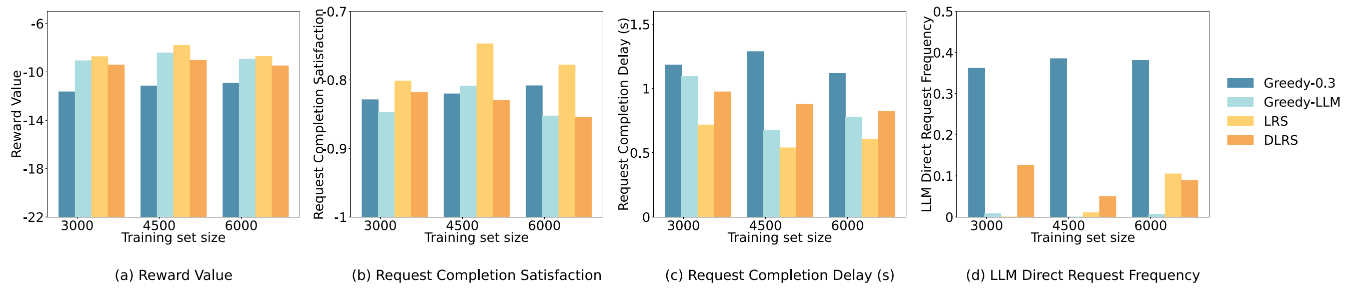


Fig. 12. Performance with different training set sizes when LLM requests are offloaded to all servers.

TABLE IV
PERFORMANCE OF DLRS ALGORITHM AFTER ADJUSTING THE NETWORK STRUCTURE

Dataset size	Configuration	Completion Satisfaction	Completion Delay (s)	Request Frequency
4500	DLRS(256)	-0.83	0.88	0.05
	DLRS(384)	$\mathbf{-0.81}$	1.19	0.07
6000	DLRS(256)	-0.85	0.82	0.09
	DLRS(384)	$\mathbf{-0.82}$	1.50	0.04

TABLE V
PERFORMANCE WITH DIFFERENT LLMS

LLMs	Reward Value	Completion Satisfaction	Completion Delay (s)	Request Frequency
Qwen-72B	-12.55	-0.85	2.98	0.18
DeepSeek-q	-16.07	$\mathbf{-0.73}$	7.79	0.17
DeepSeek-d	-10.87	-0.87	1.92	0.05
GPT-3.5	$\mathbf{-9.07}$	-0.77	**1.32**	0.01

the training by 3000 time slots without updating the network weights, yielding the experimental results shown in Table IV. The experimental results show that while generating minimal completion delays, the satisfaction with request fulfillment can be further enhanced.

Performance with different LLMs:. We have integrated more mainstream LLMs into the VELO framework for experimentation. We utilize Azure's ChatGPT-3.5-turbo and deploy the open-source DeepSeek-R1-Distill-Qwen25-32B and Qwen72B on an A800 GPU server. Due to the longer inference time of DeepSeek-R1 and the higher delay of the official API, we conduct our experimental analysis using DeepSeek-R1-Distill-Qwen25-32B.

TABLE VI
PERFORMANCE OF DLRS IN HANDLING NEW REQUESTS

Algorithms	Reward Value	Completion Satisfaction	Completion Delay (s)	Request Frequency
Greedy-0.3	-17.30	-0.84	4.92	0.65
Greedy-LLM	-13.35	-0.91	3.72	0.08
LRS	-13.75	-0.87	4.00	0.38
DLRS-V10	-11.62	-0.87	2.97	0
DLRS	-11.72	-0.85	3.23	0.03

As shown in Table V, the inference results of the DeepSeek model differ from previous experimental results with dialogue models. Despite the assistance of a vector database, the pursuit of low-latency response times leads to a noticeable decline in satisfaction. It is evident that the DeepSeek model can achieve the highest completion satisfaction. In contrast, ChatGPT-3.5-turbo delivers more balanced experimental outcomes. Our approach is applicable to various mainstream commercial LLMs and yields greater benefits with the improvement in LLM performance, demonstrating the practicality of the method.

Robustness and adaptability of our DLRS algorithm: We sample 500 diverse requests to evaluate the vector database quality and the algorithm's decision-making capabilities after training. As shown in Table VI, the DLRS algorithm outperforms others, making it ideal for open dialogue scenarios. While the DLRS algorithm makes fewer direct LLM requests with new inputs, it maintains high task satisfaction and minimal delay. This efficiency stems from its ability to build a high-quality vector database during training, enabling effective reference to knowledge when using *Action C*, thereby reducing delays linked to lengthy prompts. We also assess the execution times of the RL algorithms, which measured: LRS at 24.1 ms, DLRS-V10 at 25.5 ms, and DLRS at 26.2 ms. These results confirm that the time cost of the DLRS algorithm is reasonable. While some completion delay stems from vector retrieval, most is due to LLM inference.

Overhead of DLRS algorithm: The DLRS algorithm has a training time of 16.04 ms. The average GPU memory requirement is 421 MB, with a memory demand of 3021 MB, further confirming its resource efficiency. We have tested and found that the data transfer rate of TCP protocol between servers is about 5.0 KB/s, and the data transfer rate of FTP is about 21213.9 KB/s. Therefore, our VELO framework and DLRS algorithm are practical for real-world applications.

VI. Conclusion

In this paper, we presented a framework for optimizing QoS in LLMs through a collaborative cloud-edge approach assisted by vector databases. We comprehensively modeled the LLM request scheduling problem, considering both the satisfaction of the LLM request and the completion delay. We proposed a feature extraction method for LLM requests using the Transformer Encoder, and combined these features with the query result features of LLM requests to fully describe the request properties and their relevance to local vector data. Additionally, we introduced training and updating algorithms based on expert demonstrations to optimize the sparse LLM request features and address policy exploration challenges. We introduced the DLRS algorithm, which schedules requests through a diffusion-based policy network and enhances the QoS of LLM requests by utilizing cloud-edge collaboration. The system was deployed in a physical environment and used an open-source QA dataset to evaluate the performance of algorithms against similar and new requests. The experimental results indicated that the DLRS algorithm improves performance by 15% and 14.59% compared to the baseline algorithms when handling similar and new requests.

Future work will focus on further improving the success rate of matching vector databases with requests, as well as addressing potential parallel inference issues in request scheduling. Specifically, we will improve the vector database structure by implementing content-aware partitioning to enhance query speed and reduce irrelevant results. In addition, we plan to implement a decision-checking mechanism to analyze the relevance of the prompt content provided to the LLM, thereby reducing potential decision-making errors in the algorithm. Addressing parallel inference issues from request scheduling requires load balancing, which can be achieved by creating and allocating pods in Kubernetes. However, this comes with many challenges in system implementation, guiding our future work.

References

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

[2] M. Xu et al., “Unleashing the power of edge-cloud generative AI in mobile networks: A survey of AIGC services,” *IEEE Commun. Surv. Tut.*, vol. 26, no. 2, pp. 1127–1170, Second Quarter, 2024.

[3] C. Ding, Z. Lu, F. Juefei-Xu, V. N. Boddeti, Y. Li, and J. Cao, “Towards transmission-friendly and robust CNN models over cloud and device,” *IEEE Trans. Mobile Comput.*, vol. 22, no. 10, pp. 6176–6189, Oct. 2023.

[4] Y.-C. Wang, J. Xue, C. Wei, and C. C. J. Kuo, “An overview on generative AI at scale with edge–cloud computing,” *IEEE Open J. Commun. Soc.*, vol. 4, pp. 2952–2971, 2023.

[5] J. Kim et al., “Memory-efficient fine-tuning of compressed large language models via sub-4-bit integer quantization,” in *Proc. Adv. Neural Inf. Process. Syst.*, Curran Associates, Inc., 2024, pp. 36 187–36 207.

[6] 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.*, PMLR, 2023, pp. 38 087–38 099.

[7] C.-Y. Hsieh et al., “Distilling step-by-step! Outperforming larger language models with less training data and smaller model sizes,” in *Proc. Annu. Meeting Assoc. Comput. Linguistics*, 2023, pp. 8003–8017.

[8] M. Lin, L. Cao, Y. Zhang, L. Shao, C.-W. Lin, and R. Ji, “Pruning networks with cross-layer ranking & K-reciprocal nearest filters,” *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 34, no. 11, pp. 9139–9148, Nov. 2023.

[9] Y. Chen et al., “NetGPT: A native-AI network architecture beyond provisioning personalized generative services,” 2023, *arXiv:2307.06148*.

[10] X. Huang et al., “Federated learning-empowered AI-generated content in wireless networks,” *IEEE Netw.*, vol. 38, no. 5, pp. 304–313, Sep. 2024.

[11] H. Wang, J. Hong, and J. Luo, “Service matching based on group preference and service representation learning for edge caching,” in *Proc. 2023 IEEE Int. Conf. Web Serv.*, 2023, pp. 1–6.

[12] Q. Cao, P. Khanna, N. D. Lane, and A. Balasubramanian, “MobiVQA: Efficient on-device visual question answering,” in *Proc. ACM Interactive, Mobile, Wearable Ubiquitous Technol.*, vol. 6, no. 2, pp. 1–23, Jul. 2022.

[13] A. Nematallah, C. H. Park, and D. Black-Schaffer, “Exploring the latency sensitivity of cache replacement policies,” *IEEE Comput. Archit. Lett.*, vol. 22, no. 2, pp. 93–96, Jul./Dec. 2023.

[14] B. Cao, Q. Peng, X. Xie, Z. Peng, J. Liu, and Z. Zheng, “Web service recommendation via combining topic-aware heterogeneous graph representation and interactive semantic enhancement,” *IEEE Trans. Serv. Comput.*, vol. 17, no. 6, pp. 4451–4466, Nov./Dec. 2024, doi: 10.1109/TSC.2024.3418328.

[15] J. Wang et al., “Milvus: A purpose-built vector data management system,” in *Proc. 2021 ACM Int. Conf. Manage. Data*, 2021, pp. 2614–2627.

[16] H. Jiang, Q. Wu, C.-Y. Lin, Y. Yang, and L. Qiu, “LLMLingua: Compressing prompts for accelerated inference of large language models,” in *Proc. 2023 Conf. Empirical Methods Natural Lang. Process.*, 2023, pp. 13 358–13 376.

[17] J. J. Pan, J. Wang, and G. Li, “Survey of vector database management systems,” *VLDB J.*, vol. 33, no. 5, pp. 1591–1615, Jul. 2024.

[18] O. Topsakal and T. C. Akinci, “Creating large language model applications utilizing LangChain: A primer on developing LLM apps fast,” in *Proc. Int. Conf. Appl. Eng. Natural Sci.*, 2023, pp. 1050–1056.

[19] D. Driess et al., “PaLM-E: An embodied multimodal language model,” in *Proc. 40th Int. Conf. Mach. Learn.*, PMLR, 2023, pp. 8469–8488.

[20] T. Ahmed and P. Devanbu, “Better patching using LLM prompting, via self-consistency,” in *Proc. 38th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2023, pp. 1742–1746.

[21] Z. Tang, W. Jia, X. Zhou, W. Yang, and Y. You, “Representation and reinforcement learning for task scheduling in edge computing,” *IEEE Trans. Big Data*, vol. 8, no. 3, pp. 795–808, Jun. 2022.

[22] J. K. Gupta, M. Egorov, and M. Kochenderfer, “Cooperative multi-agent control using deep reinforcement learning,” in *Proc. Auton. Agents Multiagent Syst.*, Springer, 2017, pp. 66–83.

[23] J. Ho, A. Jain, and P. Abbeel, “Denoising diffusion probabilistic models,” in *Proc. Adv. Neural Inf. Process. Syst.*, Curran Associates, Inc., 2020, pp. 6840–6851.

[24] H. Zhu, X. Li, L. Chen, and R. Ruiz, “Smart offloading computation-intensive & delay-intensive tasks of real-time workflows in mobile edge computing,” in *Proc. 2023 IEEE Int. Conf. Web Serv.*, 2023, pp. 695–697.

[25] H. Du et al., “Diffusion-based reinforcement learning for edge-enabled AI-generated content services,” *IEEE Trans. Mobile Comput.*, vol. 23, no. 9, pp. 8902–8918, Sep. 2024.

[26] A. Vaswani et al., “Attention is all you need,” in *Proc. Adv. Neural Inf. Process. Syst.*, Curran Associates, Inc., 2017, pp. 5998–6008.

[27] Z. Lu, C. Ding, F. Juefei-Xu, V. N. Boddeti, S. Wang, and Y. Yang, “TFormer: A transmission-friendly ViT model for IoT devices,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 2, pp. 598–610, Feb. 2023.

[28] R. Guo et al., “Manu: A cloud native vector database management system,” in *Proc. VLDB Endowment*, vol. 15, no. 12, pp. 3548–3561, Aug. 2022.

[29] L. Huang, X. Feng, C. Zhang, L. Qian, and Y. Wu, "Deep reinforcement learning-based joint task offloading and bandwidth allocation for multi-user mobile edge computing," *Digit. Commun. Netw.*, vol. 5, no. 1, pp. 10–17, Feb. 2019.
[30] C. Gulcehre et al., "Making efficient use of demonstrations to solve hard exploration problems," in *Proc. Int. Conf. Learn. Representations*, OpenReview.net, 2020, pp. 1–20.
[31] Z. Yao, Z. Tang, J. Lou, P. Shen, and W. Jia, "VELO: A vector database-assisted cloud-edge collaborative LLM QoS optimization framework," in *Proc. 2024 IEEE Int. Conf. Web Serv.*, 2024, pp. 865–876.
[32] J. Bai et al., "Qwen technical report," 2023, *arXiv:2309.16609*.
[33] A. Köpf et al., "Openassistant conversations-democratizing large language model alignment," in *Proc. Adv. Neural Inf. Process. Syst.*, Curran Associates, Inc., 2024, pp. 47 669–47 681.
[34] OASST1. 2023. [Online]. Available: https://huggingface.co/datasets/OpenAssistant/oasst1
[35] J. Lin et al., "AWQ: Activation-aware weight quantization for on-device LLM compression and acceleration," in *Proc. Mach. Learn. Syst.*, 2024, pp. 87–100.
[36] S. Zhang, M. Xu, W. Y. Bryan Lim, and D. Niyato, "Sustainable AIGC workload scheduling of geo-distributed data centers: A multi-agent reinforcement learning approach," in *Proc. 2023 IEEE Glob. Commun. Conf.*, 2023, pp. 3500–3505.
[37] P. Patel, E. Choukse, C. Zhang, and A. Shah, "Splitwise: Efficient generative LLM inference using phase splitting," in *Proc. 2024 ACM/IEEE 51st Annu. Int. Symp. Comput. Archit.*, 2024, pp. 118–132.
[38] K. Rao, G. Coviello, P. Benedetti, C. G. De Vita, G. Mellone, and S. Chakradhar, "ECO-LLM: LLM-based edge cloud optimization," in *Proc. 2024 ACM Workshop AI Syst.*, 2024, pp. 7–12.
[39] S. Fujimoto and S. S. Gu, "A minimalist approach to offline reinforcement learning," in *Proc. Adv. Neural Inf. Process. Syst.*, Curran Associates, Inc., 2021, pp. 20 132–20 145.
[40] Z. Dong et al., "CleanDiffuser: An easy-to-use modularized library for diffusion models in decision making," 2024, *arXiv:2406.09509*.
[41] Z. Wang, J. J. Hunt, and M. Zhou, "Diffusion policies as an expressive policy class for offline reinforcement learning," in *Proc. Int. Conf. Learn. Representations*, 2023, pp. 1–17.
[42] D. Yu et al., "MoESys: A distributed and efficient mixture-of-experts training and inference system for internet services," *IEEE Trans. Serv. Comput.*, vol. 17, no. 5, pp. 2626–2639, Sep./Oct. 2024.
[43] Y. He, M. Xu, J. Wu, W. Zheng, K. Ye, and C. Xu, "UELLM: A unified and efficient approach for LLM inference serving," 2024, *arXiv:2409.14961*.
[44] Y. Fu, S. Zhu, R. Su, A. Qiao, I. Stoica, and H. Zhang, "Efficient LLM scheduling by learning to rank," 2024, *arXiv:2408.15792*.
[45] X. Zhao, X. Zhou, and G. Li, "Chat2Data: An interactive data analysis system with RAG, vector databases and LLMs," in *Proc. VLDB Endowment*, vol. 17, pp. 4481–4484, 2024.
[46] L.-B. Hernandez-Salinas et al., "IDAS: Intelligent driving assistance system using RAG," *IEEE Open J. Veh. Technol.*, vol. 5, pp. 1139–1165, 2024.
[47] Z. Ye et al., "Deep learning workload scheduling in GPU datacenters: A survey," *ACM Comput. Surv.*, vol. 56, no. 6, pp. 1–38, Jan. 2024.
[48] P. Wu, J. Li, L. Shi, M. Ding, K. Cai, and F. Yang, "Dynamic content update for wireless edge caching via deep reinforcement learning," *IEEE Commun. Lett.*, vol. 23, no. 10, pp. 1773–1777, Oct. 2019.
[49] Z. Han, H. Tan, G. Chen, R. Wang, Y. Chen, and F. C. Lau, "Dynamic virtual machine management via approximate Markov decision process," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun.*, 2016, pp. 1–9.
[50] Z. Tang, F. Mou, J. Lou, W. Jia, Y. Wu, and W. Zhao, "Multi-user layer-aware online container migration in edge-assisted vehicular networks," *IEEE/ACM Trans. Netw.*, vol. 32, no. 2, pp. 1807–1822, Apr. 2024.
[51] S. Black et al., "GPT-NeoX-20B: An open-source autoregressive language model," in *Proc. ACL Workshop Challenges Perspectives Creating Large Lang. Models*, 2022, pp. 95–136.
[52] H. Xiong et al., "When search engine services meet large language models: Visions and challenges," *IEEE Trans. Serv. Comput.*, vol. 17, no. 6, pp. 4558–4577, Nov./Dec. 2024, doi: 10.1109/TSC.2024.3451185.
[53] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods fomoesysr reinforcement learning with function approximation," in *Proc. Adv. Neural Inf. Process. Syst.*, MIT Press, 1999, pp. 1057–1063.
[54] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017, *arXiv: 1707.06347*.
[55] K. Han, A. Xiao, E. Wu, J. Guo, C. XU, and Y. Wang, "Transformer in transformer," in *Proc. Adv. Neural Inf. Process. Syst.*, Curran Associates, Inc., 2021, pp. 15 908–15 919.
[56] I. Sarkar, M. Adhikari, S. Kumar, and V. G. Menon, "Deep reinforcement learning for intelligent service provisioning in software-defined industrial fog networks," *IEEE Internet Things J.*, vol. 9, no. 18, pp. 16 953–16 961, Sep. 2022.
[57] L. Gao et al., "The pile: An 800GB dataset of diverse text for language modeling," 2020, *arXiv:2101.00027*.
[58] FastGPT. 2023. [Online]. Available: https://github.com/labring/FastGPT
[59] C. Yu et al., "The surprising effectiveness of PPO in cooperative multi-agent games," in *Proc. Adv. Neural Inf. Process. Syst.*, Curran Associates, Inc., 2022, pp. 24 611–24 624.
[60] C. Luo, "Understanding diffusion models: A unified perspective," 2022, *arXiv:2208.11970*.

Zhi Yao received the BS degree from the College of Electronic and Information Engineering, Shandong University of Science and Technology, China, in 2020, and the MS degree from the South China Academy of Advanced Optoelectronics, South China Normal University, China, in 2023. He is currently working toward the PhD degree with the School of Artificial Intelligence, Beijing Normal University, China. His current research interests include mobile edge computing, vector database, LLM request scheduling, and reinforcement learning.

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 assistant professor with the Advanced Institute of Natural Sciences, Beijing Normal University, China. His current research interests include edge computing, resource scheduling, and reinforcement learning.

Wenmian Yang (Member, IEEE) received the BS degree from the Department of Electronic Information and Electrical Engineering, Dalian University of Technology, China, in 2015, and the PhD degree from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China, in 2020. Currently, he is a research associate professor with Beijing Normal University (Zhuhai). His current research interests include natural language processing, time-series data, and diffusion models.

Weijia Jia (Fellow, IEEE) is currently the director with the Institute of Artificial Intelligence and Future Networking, and the director of Super Intelligent Computer Center, Beijing Normal University at Zhuhai, also a chair professor with UIC, Zhuhai, Guangdong, China. He has more than 700 publications in the prestige international journals/conferences and research books and book chapters. He has served as area editor for various prestige international journals, chair and PC member/keynote speaker for many top international conferences. He is the distinguished member of CCF.